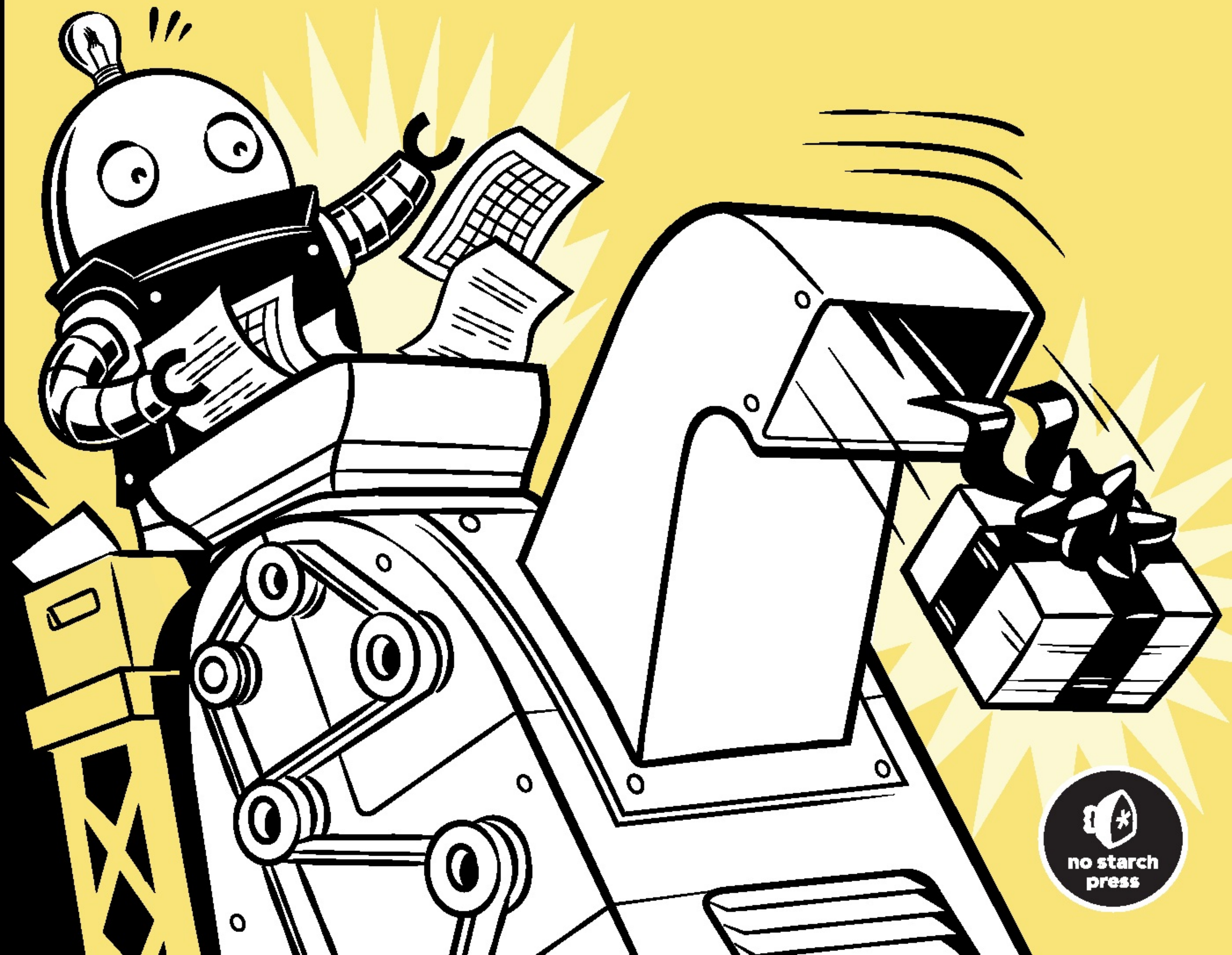
SECOND EDITION

PRACTICAL SQL

A BEGINNER'S GUIDE TO STORYTELLING WITH DATA

ANTHONY DeBARROS



no starch press

CONTENTS IN DETAIL

INDEX

PRACTICAL SQL

2nd Edition

A Beginner's Guide to Storytelling with Data

by Anthony DeBarros

no starch press

San Francisco

Printed in the United States of America

First printing

25 24 23 22 21   1 2 3 4 5 6 7 8 9

ISBN-13: 978-1-7185-0106-5 (print)
ISBN-13: 978-1-7185-0107-2 (ebook)

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Rachel Monaghan
Production Editors: Jennifer Kepler and Paula Williamson
Developmental Editor: Liz Chadwick
Cover Illustrator: Josh Ellingson
Interior Design: Octopod Studios
Technical Reviewer: Stephen Frost
Copyeditor: Kim Wimpsett

Compositor: Maureen Forys, Happenstance Type-O-Rama
Proofreader: Liz Wheeler

For information on book distributors or translations, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; info@nostarch.com
www.nostarch.com

The Library of Congress has catalogued the first edition as follows:
Names: DeBarros, Anthony, author.
Title: Practical SQL : a beginner's guide to storytelling with data / Anthony DeBarros.
Description: San Francisco : No Starch Press, 2018. | Includes index.
Identifiers: LCCN 2018000030 (print) | LCCN 2017043947 (ebook) | ISBN 9781593278458 (epub) | ISBN
1593278454 (epub) | ISBN 9781593278274 (paperback) | ISBN 1593278276 (paperback) | ISBN 9781593278458
(ebook)
Subjects: LCSH: SQL (Computer program language) | Database design. | BISAC: COMPUTERS / Programming
Languages / SQL. | COMPUTERS / Database Management / General. | COMPUTERS / Database Management
/ Data Mining.
Classification: LCC QA76.73.S67 (print) | LCC QA76.73.S67 D44 2018 (ebook) | DDC 005.75/6--dc23
LC record available at https://lccn.loc.gov/2018000030

About the Author

Anthony DeBarros is a longtime journalist and early adopter of “data journalism,” the use of spreadsheets, databases, and code to find news in data. He’s currently a data editor for the *Wall Street Journal*, where he covers topics including the economy, trade, demographics, and the Covid-19 pandemic. Previously, he worked for the Gannett company at *USA Today* and the *Poughkeepsie Journal* and held product development and content strategy roles for Questex and DocumentCloud.

About the Technical Reviewer

Stephen Frost is the chief technology officer at Crunchy Data. He has been working with PostgreSQL since 2003 and general database technology since before then. Stephen began contributing to PostgreSQL development in 2004 and has been involved in the development of the role system, column-level privileges, row-level security, GSSAPI encryption, and the predefined roles system. He has also served on the board of the United States PostgreSQL Association and Software in the Public Interest, regularly speaks at PostgreSQL Community conferences and events, and works as a member of various PostgreSQL community teams.

PREFACE TO THE SECOND EDITION

Since the publication of the first edition of *Practical SQL*, I've received kind notes about the book from readers around the world. One happy reader said it helped him ace SQL questions on a job interview. Another, a teacher, wrote to say that his students remarked favorably about having the book assigned for class. Others just wanted to say thanks because they found the book helpful and a good read, two pieces of feedback that will warm the heart of most any author.

I also sometimes heard from readers who hit a roadblock while working through an exercise or who had trouble with software or data files. I paid close attention to those emails, especially when the same question seemed to crop up more than once. Meanwhile, during my own journey of learning SQL—I use it every day at work—I'd often discover a technique and wish that I'd included it in the book.

With all that in mind, I approached the team at No Starch Press with the idea of updating and expanding *Practical SQL* into a second edition. I'm thankful they said yes. This new version of the book is more complete, offers stronger guidance for readers related to software and code, and clarifies information that wasn't as clear or presented as accurately as it could have been. The book has been thoroughly enjoyable to revisit, and I've learned much along the way.

This second edition includes numerous updates, expansions, and clarifications in every chapter. Throughout, I've been careful to note when code syntax adheres to the SQL standard—meaning you can generally use it across database systems—or when the syntax is specific to the database used in the book, PostgreSQL.

The following are among the most substantial changes:

Two chapters are new. Chapter 1, "Setting Up Your Coding Environment," details how to install PostgreSQL, pgAdmin, and additional PostgreSQL components on multiple operating systems. It also shows how to obtain the code listings and data from GitHub. In the first edition, this information was located in the introduction and occasionally missed by readers. Chapter 16, "Working with JSON Data," covers PostgreSQL's support for the JavaScript Object Notation data format, using datasets about movies and earthquakes.

In Chapter 4 on data types, I've added a section on `IDENTITY`, the ANSI SQL standard implementation for auto-incrementing integer columns. Throughout the book, `IDENTITY` replaces the PostgreSQL-specific `serial` auto-incrementing integer type so that code examples more closely reflect the SQL standard.

Chapter 5 on importing and exporting data now includes a section about using the `WHERE` keyword with the `COPY` command to filter which rows are imported from a source file to a table.

I've removed the user-created `median()` function from Chapter 6 on basic math in favor of focusing exclusively on the SQL standard `percentile_cont()` function for calculating medians.

In Chapter 7 on table joins, I've added a section covering the set operators `UNION`, `UNION ALL`, `INTERSECT`, and `EXCEPT`. Additionally, I've added a section covering the `USING` clause in joins to reduce redundant output and simplify query syntax.

Chapter 10 on inspecting and modifying data includes a new section on using the `RETURNING` keyword in an `UPDATE` statement to display the data that the statement modified. I've also added a section that describes how to use the `TRUNCATE` command to remove all rows from a table and restart an `IDENTITY` sequence.

In Chapter 11 on statistical functions, a new section demonstrates how to create a rolling average to smooth uneven data to get a better sense of trends over time. I've also added information on functions for calculating standard deviation and variance.

Chapter 13 on advanced query techniques now shows how to use the `LATERAL` keyword with subqueries. One benefit is that, by combining `LATERAL` with `JOIN`, you get functionality similar to a *for loop* in a programming language.

In Chapter 15 on analyzing spatial data, I demonstrate how to use the Geometry Viewer in pgAdmin to see geographies placed on a map. This feature was added to pgAdmin after publication of the first edition.

In Chapter 17 on views, functions, and triggers, I've added information about materialized views and showed how their behavior differs from standard views. I also cover procedures, which PostgreSQL now supports in addition to functions.

Finally, where practical, datasets have been updated to the most recent available at the time of writing. This primarily applies to US Census population statistics but also includes the text of presidential speeches and library usage statistics.

Thank you for reading *Practical SQL*! If you have any questions or feedback, please get in touch by emailing *practicalsqlbook@gmail.com*.

ACKNOWLEDGMENTS

This second edition of *Practical SQL* is the work of many hands. My thanks, first, go to the team at No Starch Press. Thanks to Bill Pollock for capturing the vision and sharpening the initial concept for the book—and for agreeing to let me have another go at it. Special thanks and appreciation to senior editor Liz Chadwick, who improved each chapter with her insightful suggestions and deft editing, and to copyeditor Kim Wimpsett and the production team of Paula Williamson and Jennifer Kepler.

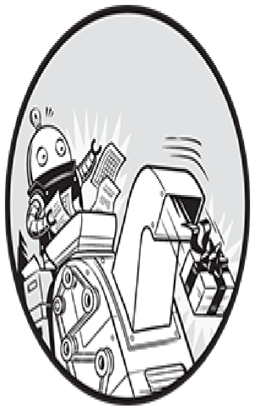
Stephen Frost, chief technology officer at Crunchy Data and longtime contributor to the PostgreSQL community, served as the technical reviewer for this edition. I deeply appreciate the time Stephen took to explain the inner workings of PostgreSQL and SQL concepts. This book is better, more thorough, and more accurate thanks to his detailed eye. I'd also like to acknowledge Josh Berkus, whose many contributions as technical reviewer for the first edition persist in this new version.

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

Many of my colleagues have not only imparted memorable lessons on data analysis, they've also made my workdays brighter. Special thanks to Paul Overberg for sharing his vast knowledge of demographics and the US Census, to Lou Schilling for many technical lessons, to Christopher Schnaars for his SQL expertise, to Peter Matseykanets for his encouragement, and to Chad Day, John West, and Maureen Linke and the WSJ DC visuals team for continual inspiration.

My deepest appreciation goes to my dear wife, Elizabeth, and our sons. You are the brightest lights in my day. As we are fond of saying, "To the journey . . ."

INTRODUCTION



Shortly after joining the staff of *USA Today*, I received a dataset that 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 titles based on confidential sales data. Not only did the list produce an endless stream of story ideas to pitch, it also captured the zeitgeist of America in a singular way.

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? Every week, the book list editor and I pored over the sales figures and book genres, ranking the data in search of a new 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 had become a perennial gift for new graduates.

My technical companion in that 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 discover the stories hidden in the database, which contained sales data related to titles, authors, genres, and the codes that defined the publishing world.

SQL has been useful to me ever since, whether my role was in product development, in content strategy, or, lately, as a data editor for the *Wall Street Journal*. In each case, SQL has helped me find interesting stories in data—and that's exactly what you'll learn to do using this book.

What Is SQL?

SQL is a widely used programming language for managing data and database systems. 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 collect, modify, explore, and summarize data.

Because SQL is a mature language that's been around for decades, it's 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, SQL still ranks as one of the most-used computer languages in the world, and that's unlikely to change soon.

Each database system, such as PostgreSQL, MySQL or Microsoft SQL Server, implements its own variant of SQL, so you'll notice subtle—or sometimes significant—differences in syntax if you jump from one system to another. There are several reasons behind this. The American National Standards Institute (ANSI) adopted a standard for SQL in 1986, followed by the International Organization for Standardization (ISO) in 1987. But the standard doesn't cover all aspects of SQL that are required for a database implementation—for example, it has no entry for creating indexes. That leaves each database system maker to choose how to implement features the standard doesn't cover—and no database maker currently claims to conform to the entire standard.

Meanwhile, business considerations can lead commercial database vendors to create nonstandard SQL features for both competitive advantage and as a way to keep users in their ecosystem. For example, Microsoft's SQL Server uses the proprietary Transact-SQL (T-SQL) that includes a number of features not in the SQL standard, such as its syntax for declaring local variables. Migrating code written using T-SQL to another database system may not be trivial, therefore.

In this book, the examples and code use the PostgreSQL database system. PostgreSQL, or simply Postgres, is a robust application that can handle 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 aims to closely follow the SQL standard.

It’s widely used, so finding help online is easy.

Its geospatial extension, PostGIS, lets you analyze geometric data and perform mapping functions and is often used with mapping software such as QGIS.

It’s available in cloud computing environments such as Amazon Web Services and Google Cloud.

It’s a common choice as a data store for web applications, including those powered by the popular web framework Django.

The good news is that the fundamental concepts and much of the core SQL syntactical conventions of PostgreSQL will work across databases. So, if you’re using MySQL at work, you can employ much of what you learn here—or easily find parallel code concepts. When syntax is PostgreSQL-specific, I make sure to point that out. If you need to learn the SQL syntax of a system with features that deviate from the standard, such as Microsoft SQL Server’s T-SQL, you may want to further explore a resource focusing on that system.

Why SQL?

SQL certainly isn’t the only option for crunching data. 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 some versions of Microsoft Office, which has a graphical query interface that makes it easy to get work done. So why learn SQL?

One reason is that Excel and Access have their limits. Excel currently allows 1,048,576 rows maximum per worksheet. Access limits database size to two gigabytes and limits columns to 255 per table. It’s not uncommon for datasets to surpass those limits. 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 fine-grained 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.

Finally, SQL is useful 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, the ability to manage data and databases with SQL will come in very handy.

Who Is This Book For?

Practical SQL is for people who encounter data in their everyday lives and want to learn how to analyze, manage, and transform it. With that in mind, we cover real-world data and scenarios, such as US Census demographics, crime reports, and data about taxi rides in New York City. We aim to understand not only how SQL works but how we can use it to find valuable insights.

This book was written with people new to programming in mind, so the early chapters cover key basics about databases, data, and SQL syntax. Readers with some SQL experience should benefit from later chapters that cover more advanced topics, such as Geographical Information Systems (GIS). I assume that you know your way around your computer, including how to install programs, navigate your hard drive, and download files from the internet, but I don't assume you have any experience with programming or data analysis.

What You'll Learn

Practical SQL starts with a chapter on setting up your system and getting the code and data examples and then moves through the basics of databases, queries, tables, and data that are common to SQL across many database systems. Chapters 14 to 19 cover topics more specific to PostgreSQL, such as full-text search, functions, and GIS. Although many chapters in this book can stand alone, you should work through the book sequentially to build on the fundamentals. Datasets presented in early chapters often reappear later, so following the book in order will help you stay on track.

The following summary provides more detail about each chapter:

Chapter 1: Setting Up Your Coding Environment walks through setting up PostgreSQL, the pgAdmin user interface, and a text editor, plus how to download example code and data.

Chapter 2: Creating Your First Database and Table provides step-by-step instructions for the process of loading a simple dataset about teachers into a new database.

Chapter 3: Beginning Data Exploration with SELECT explores basic SQL query syntax, including how to sort and filter data.

Chapter 4: 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 5: 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 US Census population data that you’ll use throughout the book.

Chapter 6: Basic Math and Stats with SQL covers arithmetic operations and introduces aggregate functions for finding sums, averages, and medians.

Chapter 7: 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 8: 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 9: Extracting Information by Grouping and Summarizing explains how to use aggregate functions to find trends in US library usage based on annual surveys.

Chapter 10: 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 11: Statistical Functions in SQL introduces correlation, regression, ranking, and other functions to help you derive more meaning from datasets.

Chapter 12: Working with Dates and Times explains how to create, manipulate, and query dates and times in your database, including working with time zones and with data about New York City taxi trips and Amtrak train schedules.

Chapter 13: Advanced Query Techniques explains how to use more complex SQL operations such as subqueries and cross tabulations, plus the `CASE` statement, to reclassify values in a dataset on temperature readings.

Chapter 14: 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 police reports and a collection of speeches by US presidents as examples.

Chapter 15: Analyzing Spatial Data with PostGIS introduces data types and queries related to spatial objects, which will let you analyze geographical features such as counties, roads, and rivers.

Chapter 16: Working with JSON Data introduces the JavaScript Object Notation (JSON) data format and uses data about movies and earthquakes to explore PostgreSQL JSON support.

Chapter 17: Saving Time with Views, Functions, and Triggers explains how to automate database tasks so you can avoid repeating routine work.

Chapter 18: 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 19: Maintaining Your Database provides tips and procedures for tracking the size of your database, customizing settings, and backing up data.

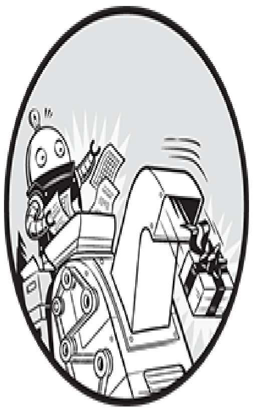
Chapter 20: Telling Your Data's Story provides guidelines for generating ideas for analysis, vetting data, drawing sound conclusions, and presenting your findings clearly.

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.

Ready? Let's begin with Chapter 1, "Setting Up Your Coding Environment."

1
SETTING UP YOUR CODING ENVIRONMENT



Let's begin by installing the resources you'll need to complete the exercises in the book. In this chapter, you'll install a text editor, download the example code and data, and then install the PostgreSQL database system and its companion graphical user interface, pgAdmin. I'll also tell you how to get help if you need it. When you're finished, your computer will have a robust environment for you to learn how to analyze data with SQL.

Avoid the temptation to skip ahead to the next chapter. My high school teacher (clearly a fan of alliteration) used to tell us that "proper planning prevents poor performance." If you follow all the steps in this chapter, you'll avoid headaches later.

Our first task is to set up a text editor suitable for working with data.

Installing a Text Editor

The source data you'll add to a SQL database is typically stored in multiple *text files*, often in a format called *comma-separated values (CSV)*. You'll learn more about the CSV format in Chapter 5, in the section "Working with Delimited Text Files," but for now let's make sure you have a text editor that will let you open those files without inadvertently harming the data.

Common business applications—word processors and spreadsheet programs—tend to introduce styles or hidden characters into files without asking, and that makes using them for data work problematic, as data software expects data in precise formats. For example, if you open a CSV file with Microsoft Excel, the program will automatically alter some data to make it more human-readable; it will assume, for example, that an item code of `3-09` is a date and format it as `9-Mar`. Text editors deal exclusively with plain text with no embellishments such as formatting, and for that reason programmers use them to edit files that hold source code, data, and software configurations—all cases where you want your text to be treated as text, and nothing more.

Any text editor should work for the book's purposes, so if you have a favorite, feel free to use it. Here are some I have used and recommend. Except where noted, they are free and available for macOS, Windows, and Linux.

Visual Studio Code by Microsoft: *https://code.visualstudio.com/*

Atom by GitHub: *https://atom.io/*

Sublime Text by Sublime HQ (free to evaluate but requires purchase for continued use): *https://www.sublimetext.com/*

Notepad++ by author Don Ho (Windows only): *https://notepad-plus-plus.org/* (note that this is a different application than *Notepad.exe*, which comes with Windows)

More advanced users who prefer to work in the command line may want to use one of these two text editors, which are installed by default in macOS and Linux:

`vim` by author Bram Moolenaar and the open source community: *https://www.vim.org/*

GNU `nano` by author Chris Allegretta and the open source community: *https://www.nano-editor.org/*

If you don't have a text editor, download and install one and get familiar with the basics of opening folders and working with files.

Next, let's get the book's example code and data.

Downloading Code and Data from GitHub

All of the code and data you'll need for working through the book's exercises are available for download. To get it, follow these steps:

- Visit the book's page on the No Starch Press website at *https://nostarch.com/practical-sql-2nd-edition/*.
- On the page, click **Download the code from GitHub** to visit the repository on *https://github.com/* that holds the material.
- On the Practical SQL 2nd Edition page at GitHub, you should see a **Code** button. Click it, and then select **Download ZIP** to save the ZIP file to your computer. Place it in a location where you can easily find it, such as your desktop. (If you're a GitHub user, you can also clone or fork the repository.)
- Unzip the file. You should then see a folder named *practical-sql-2-master* that contains the various files and subfolders for the book. Again, place this folder where you can easily find it.

NOTE

Windows users will need to provide permission for the PostgreSQL database you will install to read and write to the contents of the practical-sql-2-master *folder. To do so, right-click the folder, click* ***Properties****, and click the* ***Security*** *tab. Click* ***Edit*** *and 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****.*

Inside the *practical-sql-2-master* folder, for each chapter you'll find a subfolder named *Chapter_XX* (*XX* is the chapter number). Inside subfolders, each chapter that includes code examples will also have a file named *Chapter_XX* that ends with a *.sql* extension. This is a SQL code file that you can open with your text editor or with the PostgreSQL administrative tool you'll install later in this chapter. 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--` in the listing.

The chapter folders also contain the public data you'll use in the exercises, stored in CSV and other text-based files. As noted, it's fine to view CSV files with a true text editor, but don't open these files with Excel or a word processor.

Now, with the prerequisites complete, let's load the database software.

Installing PostgreSQL and pgAdmin

In this section, you'll install both the PostgreSQL database system and a companion graphical administrative tool, pgAdmin. Think of pgAdmin as a helpful visual workspace for managing your PostgreSQL database. Its interface lets you see your database objects, manage settings, import and export data, and write queries, which is the code that retrieves data from your database.

One benefit of using PostgreSQL is that the open source community has provided excellent guidelines that make it easy to get PostgreSQL up and running. The following sections outline installation for Windows, macOS, and Linux as of this writing, but the steps might change as new versions of the software or operating systems are released. Check the documentation noted in each section as well as the GitHub repository with the book's resources; I'll maintain files there with updates and answers to frequently asked questions.

NOTE

I recommend you 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 11.0 or later.

Windows Installation

For Windows, I recommend using the installer provided by the company EDB (formerly EnterpriseDB), which offers support and services for PostgreSQL users. When you download the PostgreSQL package bundle from EDB, you also get pgAdmin and Stack Builder, which includes a few other tools you'll use in this book and throughout your SQL career.

To get the software, visit *https://www.postgresql.org/download/windows/* and click **Download the installer** in the EDB section. This should lead to a downloads page on the EDB site. Select the latest available 64-bit Windows version of PostgreSQL unless you're using an older PC with 32-bit Windows.

NOTE

The following section covers installation for Windows 10. If you're using Windows 11, please check the GitHub repository with the book's resources for notes about any adjustments to these instructions.

After you download the installer, follow these steps to install PostgreSQL, pgAdmin, and additional components:

. 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.

. Choose your installation directory, accepting the default.

- On the Select Components screen, select the boxes to install PostgreSQL Server, the pgAdmin tool, Stack Builder, and the command line tools.
- Choose the location to store data. You can choose the default, which is in a *data* subdirectory in the PostgreSQL directory.
- Choose a password. PostgreSQL is robust with security and permissions. This password is for the default initial database superuser account, which is called `postgres`.
- Select the default port number where the server will listen. Unless you have another database or application using it, use the default, which should be `5432`. You can substitute `5433` or another number if you already have an application using the default port.
- Select your locale. Using the default is fine. Then click through the summary screen to begin the installation, which will take several minutes.
- When the installation is done, you'll be asked whether you want to launch EnterpriseDB's Stack Builder to obtain additional packages. Make sure the box is checked and click **Finish**.
- When Stack Builder launches, choose the PostgreSQL installation on the drop-down menu and click **Next**. A list of additional applications should download.
- Expand the Spatial Extensions menu and select the PostGIS Bundle for the version of PostgreSQL you installed. You may see more than one listed; if so, choose the newest version. 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.

  When installation files have been downloaded, click **Next** to install the language and PostGIS 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 install location, and click **Next** again.
- Enter your database password when prompted and continue through the prompts to install PostGIS.

. Answer **Yes** when asked to register the `PROJ_LIB` and `GDAL_DATA` environment variables. Also, answer **Yes** to the questions about setting `POSTGIS_ENABLED_DRIVERS` and enabling the `POSTGIS_ENABLE_OUTDB_RASTERS` environment variable. Finally, click through the final Finish steps to complete the installation and exit the installers. Depending on the version, you may be prompted to restart your computer.

When finished, you should have two new folders in your Windows Start menu: one for PostgreSQL and another for PostGIS.

If you'd like to get started right away, you can skip ahead to the section "Working with pgAdmin." Otherwise, follow the steps in the next section to set environment variables for optional Python language support. We cover using Python with PostgreSQL in Chapter 17; you can wait until then to set up Python if you'd like to move ahead now.

Configuring Python Language Support

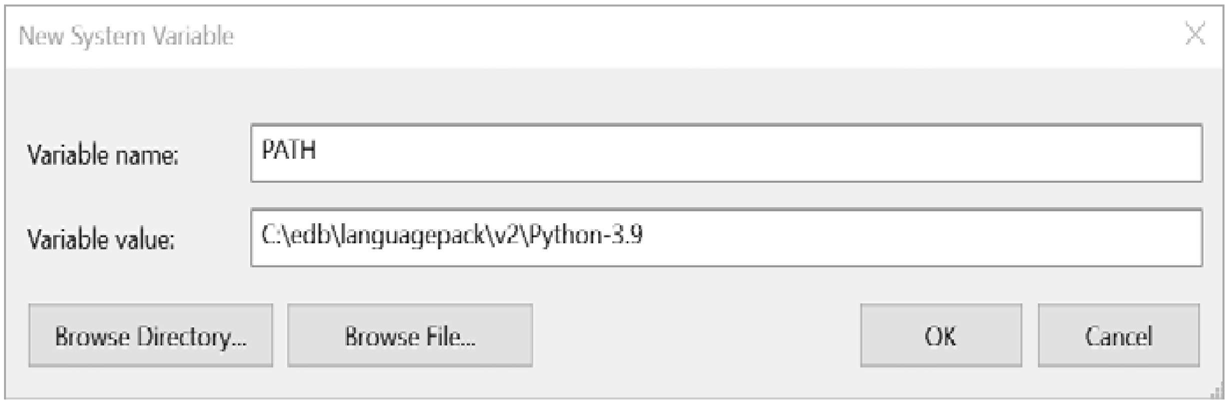
In Chapter 17, you'll learn how to use the Python programming language with PostgreSQL. In the previous section, you installed the EDB Language Pack, which provides Python support. Follow these steps to add the location of the Language Pack files to your Windows system's environment variables:

. Open the Windows Control Panel by clicking the **Search** icon on the Windows taskbar, entering **Control Panel**, and then clicking the **Control Panel** icon.

. In the Control Panel app, enter **Environment** in the search box. In the list of search results displayed, click **Edit the System Environment Variables**. A System Properties dialog will appear.

. In the System Properties dialog, on the Advanced tab, click **Environment Variables**. The dialog that opens has two sections: User variables and System variables. In the System variables section, if you don't see a `PATH` variable, continue to step a to create a new one. If you do see an existing `PATH` variable, continue to step b to modify it.

If you don't see PATH in the System variables section, click **New** to open a New System Variable dialog, shown in *Figure 1-1*.



Variable name: PATH

Variable value: C:\edb\languagepack\v2\Python-3.9

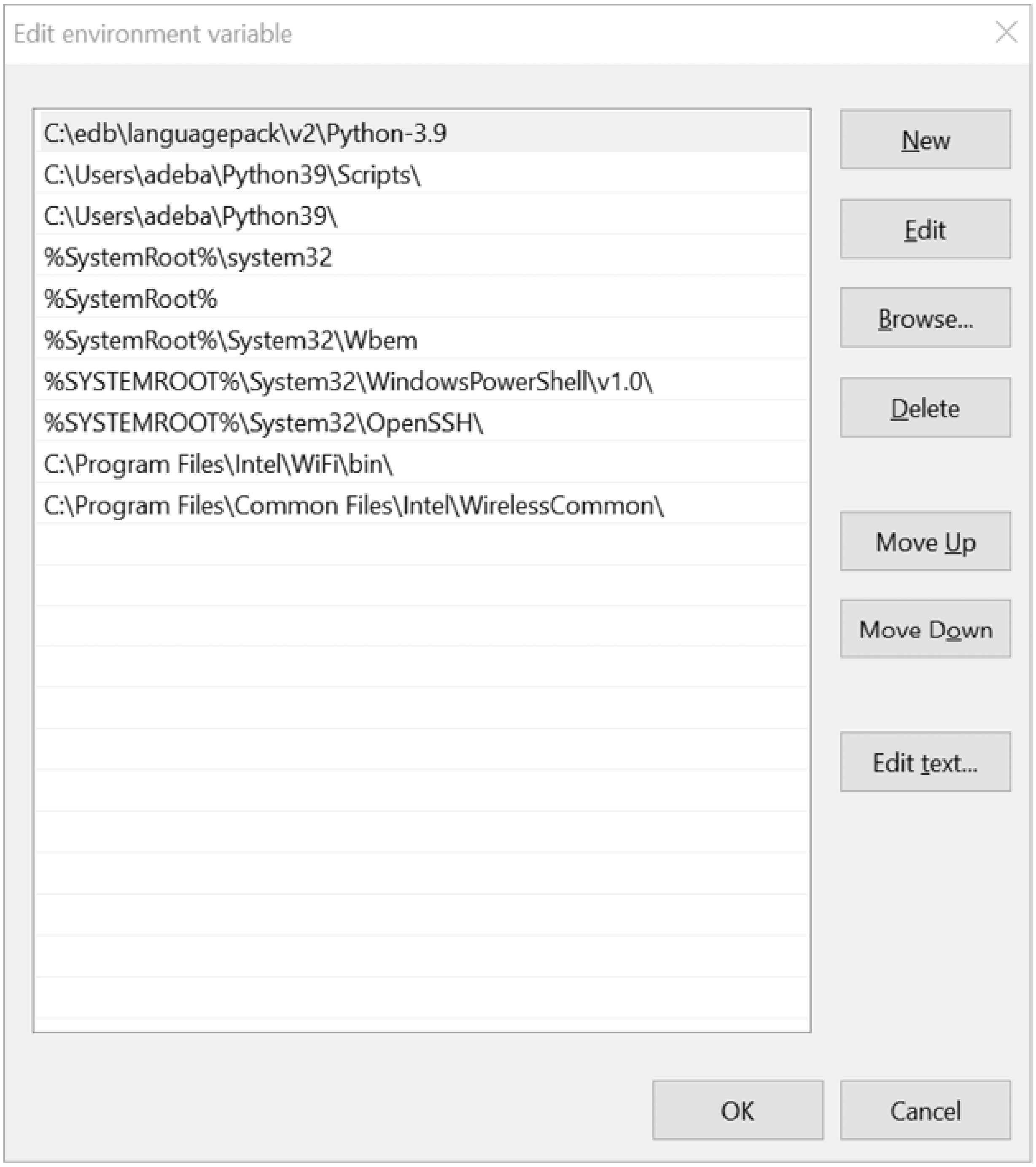
Figure 1-1: Creating a new PATH environment variable in Windows 10

In the Variable name box, enter **PATH**. In the Variable value box, enter **C:\edb\languagepack\v2\Python-3.9**. (Instead of typing, you can click **Browse Directory** and navigate to the directory in the Browse For Folder dialog.) When you've either entered the path manually or browsed to it, click **OK** on the dialog to close it.

If you do see an existing PATH variable in the System variables section, highlight it and click **Edit**. In the list of variables that displays, click **New** and enter **C:\edb\languagepack\v2\Python-3.9**. (Instead of typing, you can click Browse Directory and navigate to the directory in the Browse For Folder dialog.)

Once you've added the Language Pack path, highlight it in the list of variables and click **Move Up** until the path is at the top of the variables list. That way, PostgreSQL will find the correct Python version if you have additional Python installations.

The result should look like the highlighted line in *Figure 1-2*. Click **OK** to close the dialog.



```
C:\edb\languagepack\v2\Python-3.9
C:\Users\adeba\Python39\Scripts\
C:\Users\adeba\Python39\
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
C:\Program Files\Intel\WiFi\bin\
C:\Program Files\Common Files\Intel\WirelessCommon\
```

Figure 1-2: Editing existing `PATH` environment variables in Windows 10

Finally, in the System variables section, click **New**. In the New System Variable dialog, enter **PYTHONHOME** in the Variable name box. In the Variable value box, enter **C:\edb\languagepack\v2\Python-3.9**. When you're finished, click **OK** in all dialogs to close them. Note that these Python path settings will take effect the next time you restart your system.

If you experience any hiccups during the PostgreSQL install, check the resources for the book, where I will note changes that occur as the software is developed and can also answer questions. If you're unable to install PostGIS via Stack Builder, try downloading a separate installer from the PostGIS site at https://postgis.net/windows_downloads/ and consult the guides at https://postgis.net/documentation/.

Now, you can move ahead to the section "Working with pgAdmin."

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. Separately, you'll need to install the pgAdmin GUI and the Python language for use in functions.

Installing Postgres.app and pgAdmin

Follow these steps:

Visit https://postgresapp.com/ and download the latest release of the app. This will be a Disk Image file that ends in *.dmg*.

Double-click the *.dmg* file to open it, and then drag and drop the app icon into your *Applications* folder.

In your *Applications* folder, double-click the app icon to launch Postgres.app. (If you see a dialog that says the app cannot be opened because the developer cannot be verified, click **Cancel**. Then right-click the app icon and choose **Open**.) When Postgres.app opens, click **Initialize** to create and start a PostgreSQL database server.

A small elephant icon will appear in your menu bar to indicate that you now have a database running. To set up the included PostgreSQL command line tools so you're able to use them in future, open your Terminal application and run the following single line of 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
```

You may be prompted for the password you use to log in to your Mac. Enter that. The commands should execute without providing any output.

Next, because Postgres.app doesn't include pgAdmin, follow these steps to install pgAdmin:

- Visit the pgAdmin site's page for macOS downloads at *https://www.pgadmin.org/download/pgadmin-4-macos/*.
- Select the latest version and download the installer (look for a Disk Image file that ends in *.dmg*).
- 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.

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/*.

Installing Python

In Chapter 17, you'll learn how to use the Python programming language with PostgreSQL. To use Python with Postgres.app, you must install a specific version of the language even though macOS comes with Python pre-installed (and you might have set up an additional Python environment). To enable Postgres.app's optional Python language support, follow these steps:

- Visit the official Python site at *https://www.python.org/* and click the **Downloads** menu.
- In the list of releases, find and download the latest version of Python 3.9. Choose the appropriate installer for your Mac's processor—an Intel chip on older Macs or Apple Silicon for newer models. The download is an Apple software package file that ends in *.pkg*.

Double-click the package file to install Python, clicking through license agreements. Close the installer when finished.

Python requirements for Postgres.app may change over time. Check its Python documentation at *https://postgresapp.com/documentation/plpython.html* as well as the resources for this book for updates.

You're now ready to move ahead to the section "Working with pgAdmin."

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 times you can accomplish an installation with a few commands, but finding those commands requires some Internet sleuth work. Thankfully, 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, so there's a chance the PostgreSQL you have downloaded may not be the latest. 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.

The pgAdmin app 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/*. Once finished, you can move ahead to the section "Working with pgAdmin."

Ubuntu Installation Example

To give you a sense of what a PostgreSQL Linux install looks like, here are the steps I took to load PostgreSQL, pgAdmin, PostGIS, and PL/Python on Ubuntu 21.04, codenamed Hirsute Hippo. It's a combination of the directions found at https://wiki.postgresql.org/wiki/Apt plus the "Basic Server Setup" section at https://help.ubuntu.com/community/PostgreSQL/. You can follow along if you're on Ubuntu.

Open your Terminal by pressing CTRL-ALT-T. Then, at the prompt, enter the following lines to import a key for the PostgreSQL APT repository:

```
sudo apt-get install curl ca-certificates gnupg
curl https://www.postgresql.org/media/keys/ACCC4CF8.asc |
sudo apt-key add -
```

Next, run this single line to create the file */etc/apt/sources.list.d/pgdg.list*:

```
sudo sh -c 'echo "deb
https://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-
pgdg main" > /etc/apt/sources.list.d/pgdg.list'
```

Once that's done, update the package lists and install PostgreSQL and pgAdmin with the next two lines. Here, I installed PostgreSQL 13; you can choose a newer version if available.

```
sudo apt-get update
sudo apt-get install postgresql-13
```

You should now have PostgreSQL running. At the Terminal, enter the next line, which allows you to log in to the server and connect as the default `postgres` user to the `postgres` database using the `psql` interactive terminal, which we'll cover in depth in Chapter 18:

```
sudo -u postgres psql postgres
```

When `psql` launches, it displays version information as well as a `postgres=#` prompt. Enter the following at the prompt to set a password:

```
postgres=# \password postgres
```

I also like to create a user account with a name that matches my Ubuntu username. To do this, at the `postgres=#` prompt, enter the following line, substituting your Ubuntu username where you see `anthony`:

```
postgres=# CREATE USER anthony SUPERUSER;
```

Exit `psql` by entering **`\q`** at the prompt. You should be back at your Terminal prompt once again.

To install pgAdmin, first import a key for the repository:

```
curl https://www.pgadmin.org/static/packages_pgadmin_org.pub
| sudo apt-key add
```

Next, run this single line to create the file */etc/apt/sources.list.d/pgadmin4.list* and update package lists:

```
sudo sh -c 'echo "deb
https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/$(lsb_rel
ease -cs) pgadmin4 main" >
/etc/apt/sources.list.d/pgadmin4.list && apt update'
```

Then you can install pgAdmin 4:

```
sudo apt-get install pgadmin4-desktop
```

Finally, to install the PostGIS and PL/Python extensions, run the following lines in your terminal (substituting the version numbers of your PostgreSQL version):

```
sudo apt install postgresql-13-postgis-3
sudo apt install postgresql-plpython3-13
```

Check the official Ubuntu and PostgreSQL documentation for updates. If you experience any errors, typically with Linux an online search will yield helpful tips.

Working with pgAdmin

The final piece of your setup puzzle is to get familiar with pgAdmin, an administration and management tool for PostgreSQL. The pgAdmin software is free, but don't underestimate its performance; it's a full-featured tool as powerful as paid tools such as Microsoft's SQL Server Management Studio. With pgAdmin, you get a graphical interface where you can configure multiple aspects of your PostgreSQL server and databases, and—most appropriately for this book—use a SQL query tool for writing, running, and saving queries.

Launching pgAdmin and Setting a Master Password

Assuming you followed the installation steps for your operating system earlier in the chapter, here's how to launch pgAdmin:

Windows: Go to the Start menu, find the PostgreSQL folder for the version you installed, click it, and then select **pgAdmin4**.

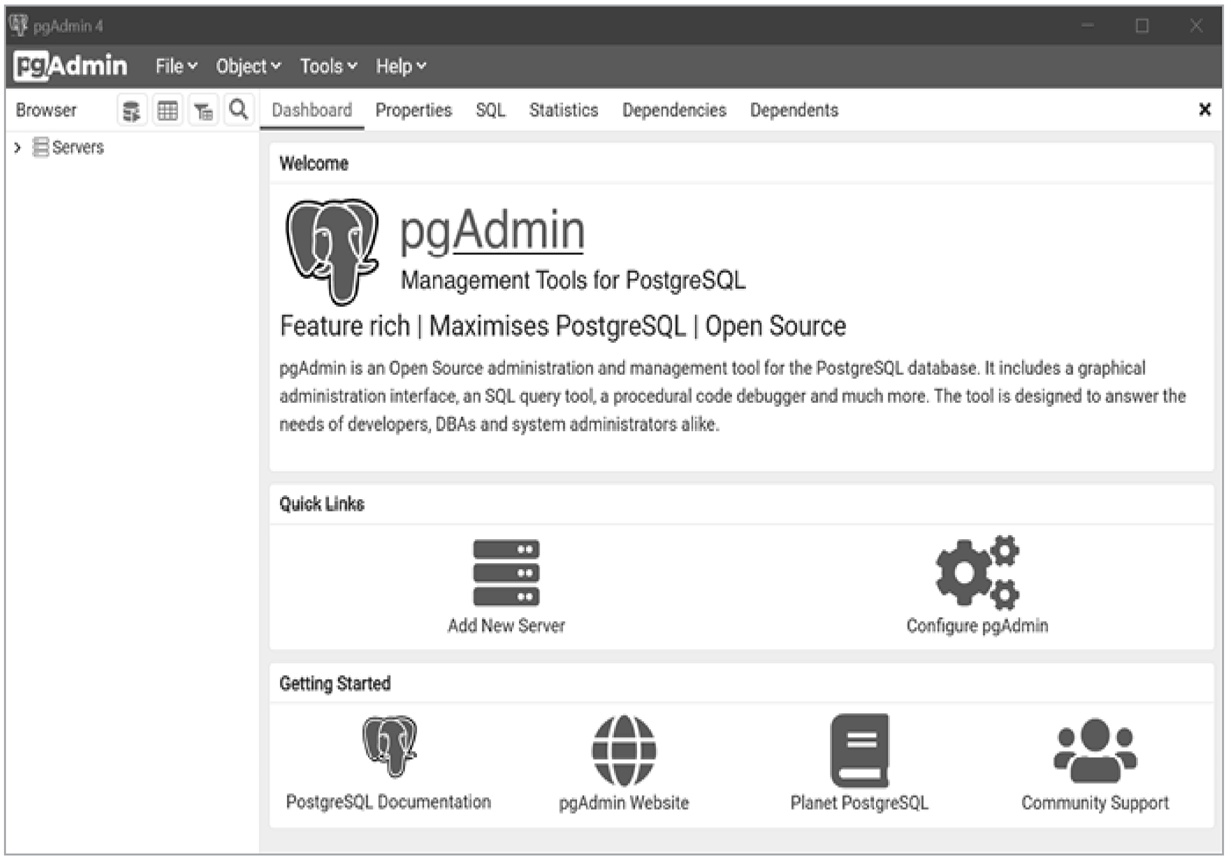
macOS: Click the **pgAdmin** icon in your *Applications* folder, making sure you've also launched Postgres.app.

Linux: Startup may differ depending on your Linux distribution. Typically, at your Terminal prompt, enter **`pgadmin4`** and press ENTER. In Ubuntu, pgAdmin appears as an app in the Activities Overview.

You should see the pgAdmin splash screen, followed by the application opening, as in *Figure 1-3*. If it's your first time launching pgAdmin, you'll also receive a prompt to set a master password. This password is not related to the one you set up during the PostgreSQL install. Set a master password and click **OK**.

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 click ***Open****. The next dialog should give you the option to open the app; going forward, your Mac will remember you’ve granted this permission.*



pgAdmin is an Open Source administration and management tool for the PostgreSQL database. It includes a graphical administration interface, an SQL query tool, a procedural code debugger and much more. The tool is designed to answer the needs of developers, DBAs and system administrators alike.

Figure 1-3: The pgAdmin app running on Windows 10

The pgAdmin layout includes a left vertical pane that 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, let’s connect to your database.

Connecting to the Default postgres Database

PostgreSQL is a *database management system*, which means it's software 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`. A database is a collection of objects that includes tables, functions, and much more, and this is where your actual data will lie. We use the SQL language (as well as pgAdmin) to manage objects and data stored in the database.

In the next chapter, you'll create your own database on your PostgreSQL server to organize your work. For now, let's connect to the default `postgres` database to explore pgAdmin. Use the following steps:

- In the object browser, click the downward-pointing arrow 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.
- Double-click the server name. If prompted, enter the database password you chose during installation (you can choose to save the password so you don't need type it in the future). A brief message appears while pgAdmin is establishing a connection. When you're connected, several new object items should display under the server name.
- Expand Databases and then expand the default database `postgres`.
- Under `postgres`, expand the Schemas object, and then expand public.

Your object browser pane should look similar to *Figure 1-4*.

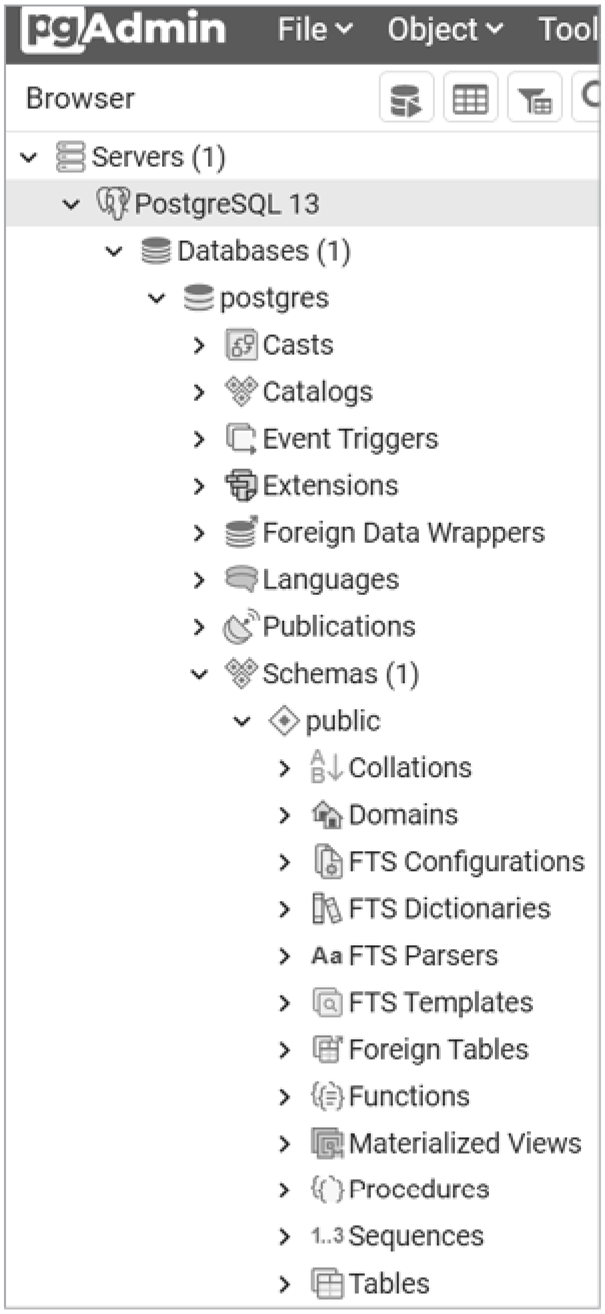


Figure 1-4: The pgAdmin object browser

NOTE

If pgAdmin doesn't show a default under Servers, you'll need to add it. Right-click Servers, and click ***Create▸Server****. In the dialog, type a name for your server in the General tab. On the Connection tab, in the Host name/address box, enter* `localhost`*. Fill in your username and the password you supplied when installing PostgreSQL and then click* ***Save****. You should now see your server listed.*

This collection of objects defines every feature of your database server. That includes tables, where we store data. To view a table's structure or perform actions on it with pgAdmin, you can access the table here. In Chapter 2, you'll use this browser to create a new database and leave the default `postgres` as is.

Exploring the Query Tool

The pgAdmin app includes a *Query Tool*, which is where you write and execute code. To open the Query Tool, in pgAdmin's object browser, first click once on any database to highlight it. For example, click the `postgres` database and then select **Tools▸Query Tool**. You'll see three panes: a Query Editor, a Scratch Pad for holding code snippets while you work, and a Data Output pane that displays query results. You can 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 a database in the object browser and open the Query Tool again via the menu.

Let's run a simple query and see its output, using the statement in *Listing 1-1* that returns the version of PostgreSQL you've installed. This code, along with all the examples in this book, is available for download via the resources at *https://nostarch.com/practical-sql-2nd-edition/* by clicking the link **Download the code from GitHub**.

```
SELECT version();
```

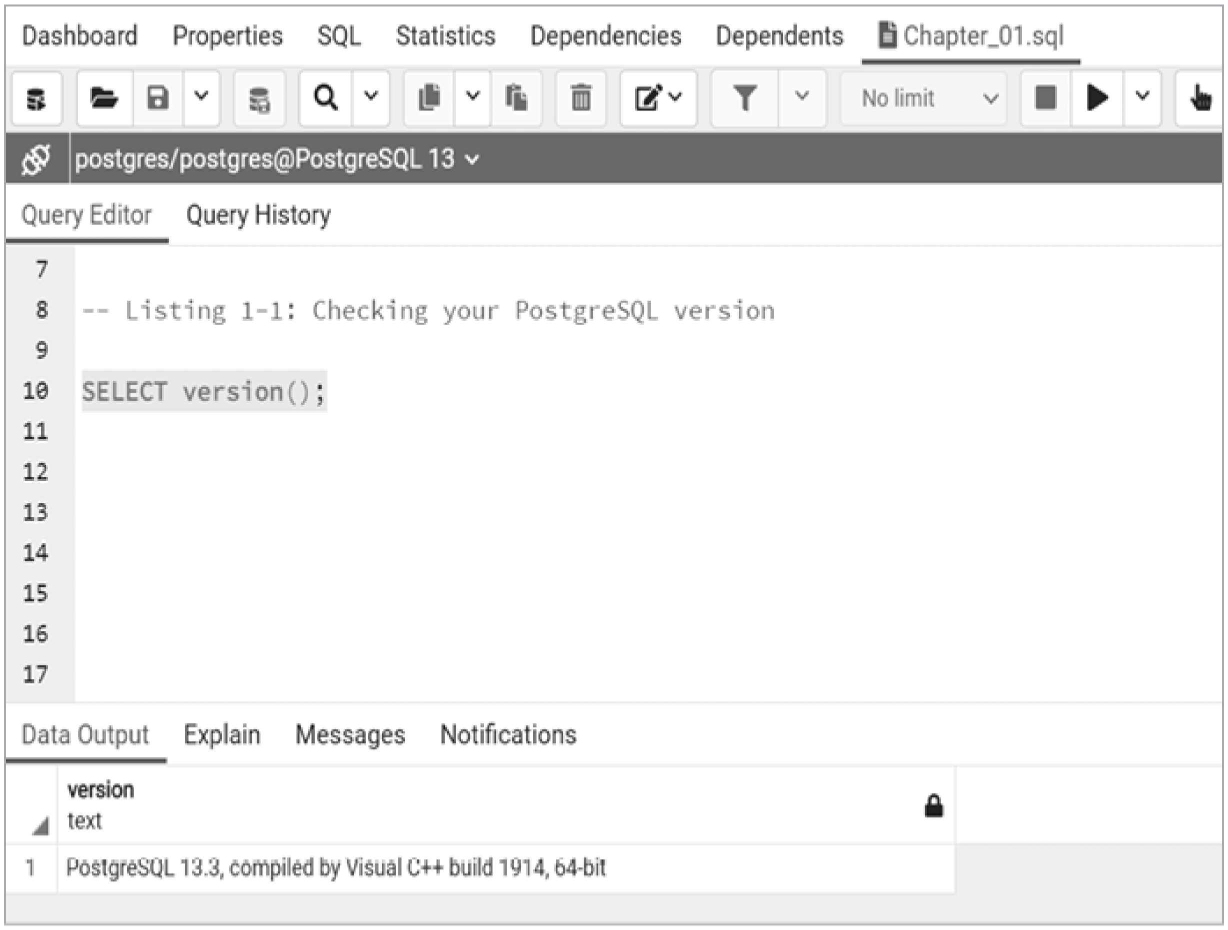
Listing 1-1: Checking your PostgreSQL version

Enter the code into the Query Editor or, if you downloaded the book's code from GitHub, click the **Open File** icon on the pgAdmin toolbar and navigate to the folder where you saved the code to open the file *Chapter_01.sql* in the *Chapter_01* folder. To execute the statement, highlight the line beginning with `SELECT` and click the **Execute/Refresh** icon in the toolbar (it's shaped like a play button). PostgreSQL should return the server's version as a result in the pgAdmin Data Output pane, as in *Figure 1-5* (you may need to expand the column in the Data Output pane by clicking on the right edge and dragging to your right to see the full results).

You'll learn much more about queries later in the book, but for now all you need to know is that this query uses a PostgreSQL-specific *function* called `version()` to retrieve the version information for the server. In my case, the output shows that I'm running PostgreSQL 13.3, and it provides additional specifics on the build of the software.

NOTE

Most of the sample code files you can download from GitHub contain more than one query. To run just one query at a time, first highlight the code for that query and then click ***Execute/Refresh****.*



```
-- Listing 1-1: Checking your PostgreSQL version

SELECT version();
```

```
  version
  text
1 PostgreSQL 13.3, compiled by Visual C++ build 1914, 64-bit
```

Figure 1-5: The pgAdmin Query Tool displaying query results

Customizing pgAdmin

Selecting **File▶Preferences** from the pgAdmin menu opens a dialog where you can customize pgAdmin's appearance and options. Here are three you may want to visit now:

Miscellaneous▶Themes lets you choose between the standard light pgAdmin theme and a dark theme.

Query Tool▶Results grid lets you set a maximum column width in query results. In that dialog, choose **Column data** and enter a value of **300** in **Maximum column width**.

The **Browser** section lets you configure the pgAdmin layout and set keyboard shortcuts.

To get help on pgAdmin options, choose **Help▶Online Help** from the menu. Feel free to explore the preferences further before moving on.

Alternatives to pgAdmin

Although pgAdmin is great for beginners, you’re not required to use it for these exercises. 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 18 provides instructions on using the PostgreSQL interactive terminal `psql` from the command line. (The appendix lists PostgreSQL resources you can explore to find additional administrative tools.)

Wrapping Up

Now that you’ve set up your environment with code, a text editor, PostgreSQL, and pgAdmin, you’re ready to start learning SQL and use it to discover valuable insights into your data!

In Chapter 2, 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!

2
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.

In this chapter, you'll create your first database, add a table, and then insert several rows of data into the table using SQL in the pgAdmin interface. Then, you'll use pgAdmin to view the results. Let's start with a look at tables.

Understanding Tables

Knowing your 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 2023
DAVISHE010    COMPSCI101    3                Fall 2023
ABRILDA002    ENG101        40               Fall 2023
DAVISHE010    ENG101        40               Fall 2023
RILEYPH002    ENG101        40               Fall 2023
```

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 those tables relate 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        2005-01-10
CHRISPA004    Chris         Park         1999-04-10
DAVISHE010    Davis         Hernandez    2006-09-14
RILEYPH002    Riley         Phelps       2005-06-15
```

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 installed in Chapter 1 is a *database management system*, a software package that allows you to define, manage, and query data stored in databases. A database is a collection of objects that includes tables, functions, and much more. 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`.

According to the PostgreSQL documentation, the default `postgres` database is "meant for use by users, utilities and third-party applications" (see *https://www.postgresql.org/docs/current/app-initdb.html*). We'll create a new database to use for the examples in the book rather than use the default, so we can keep objects related to a particular topic or application organized together. This is good practice: it helps avoid a pileup of tables in a single database that have no relation to each other, and it ensures that if your data will be used to power an application, such as a mobile app, then the app database will contain only relevant information.

To create a database, you need just one line of SQL, shown in Listing 2-1, which we'll run in a moment using pgAdmin. You can find this code, along with all the examples in this book, in the files you downloaded from GitHub via the link at *https://www.nostarch.com/practical-sql-2nd-edition/*.

```
CREATE DATABASE analysis;
```

Listing 2-1: Creating a database named `analysis`

This statement creates a database named `analysis` on your server using default PostgreSQL settings. Note that the code consists of two keywords—`CREATE` and `DATABASE`—followed by the name of the new database. You end the statement with a semicolon, which signals the end of the command. You must end all PostgreSQL statements with a semicolon, as part of the ANSI SQL standard. In some circumstances your queries will work even if you omit the semicolon, but not always, so using the semicolon is a good habit to form.

Executing SQL in pgAdmin

In Chapter 1, you 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 the SQL statements you write, known as *executing* the code. Later in the book in Chapter 18, 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 2-1 that creates the database. Then, we'll connect to the new database and create a table. Follow these steps:

. Run PostgreSQL. If you're using Windows, the installer sets PostgreSQL to launch every time you boot up. On macOS, you must double-click

Postgres.app in your Applications folder (if you have an elephant icon in your menu bar, it's already running).

Launch pgAdmin. You'll be prompted to enter the master password for pgAdmin you set the first time you launched the application.

As you did in Chapter 1, in the left vertical pane (the object browser) click the arrow 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. You may receive another password prompt. This prompt is for PostgreSQL, not pgAdmin, so enter the password you set for PostgreSQL during installation. You should see a brief message that pgAdmin is establishing a connection.

In pgAdmin's object browser, expand **Databases** and click `postgres` once to highlight it, as shown in Figure 2-1.

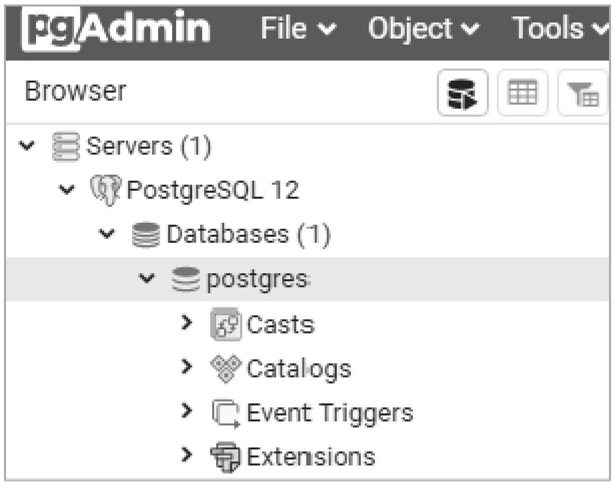
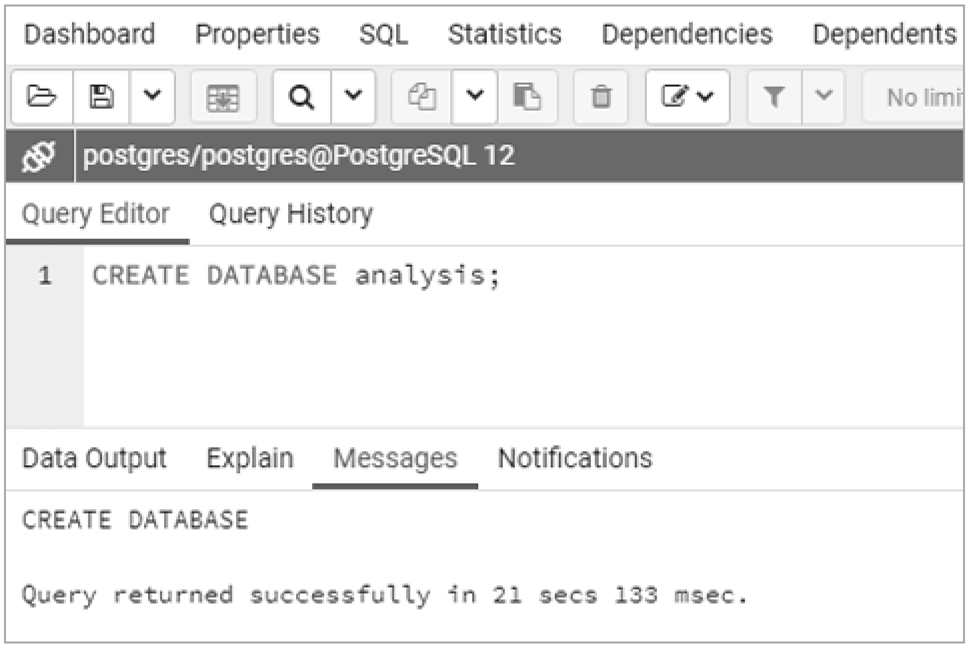


Figure 2-1: The default `postgres` *database*

Open the Query Tool by choosing **Tools▶Query Tool**.

In the Query Editor pane (the top horizontal pane), enter the code from Listing 2-1.

Click the **Execute/Refresh** icon (shaped like a right arrow) 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 2-2.



```
CREATE DATABASE analysis;
```

```
CREATE DATABASE

Query returned successfully in 21 secs 133 msec.
```

Figure 2-2: Creating a database named `analysis`

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 2-3*.

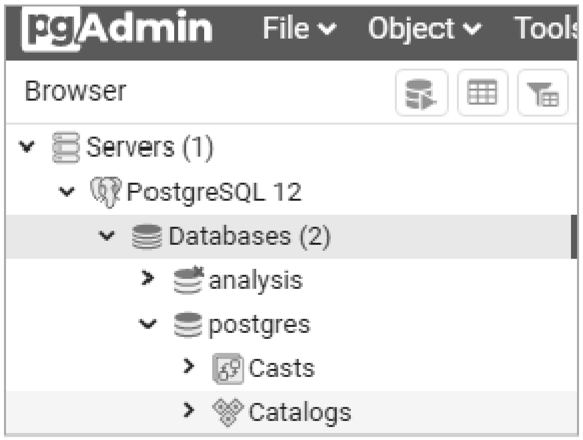


Figure 2-3: The `analysis` database displayed in the object browser

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.

NOTE

*Instead of entering the code from the listings, you can open the files you downloaded from GitHub in pgAdmin and run listings individually by highlighting a listing and clicking **Execute/Refresh**. To open a file, in the Query Tool click the **Open File** icon and navigate to the place where you saved the code.*

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:

Close the Query Tool by clicking the **X** at the far right of the tool pane. You don’t need to save the file when prompted.

In the object browser, click **analysis** once.

Open a new Query Tool window, this time connected to the `analysis` database, by choosing **Tools▶Query Tool**.

You should now see the label `analysis/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, 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 each column 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 accept data in only 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, as well as easily alter the table later if your needs change.

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

Using the CREATE TABLE Statement

For this exercise, we'll use an often-discussed piece of data: teacher salaries. Listing 2-2 shows the SQL statement to create a table called `teachers`. Let's review the code before you enter it into pgAdmin and execute it.

```
1 CREATE TABLE teachers (
      2 id bigserial,
      3 first_name varchar(25),
        last_name varchar(50),
        school varchar(50),
      4 hire_date date,
      5 salary numeric
6 );
```

Listing 2-2: 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 8, 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 `CREATE` and `TABLE` 1 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 makes the code more readable.

Each column name represents one discrete data element defined by a data type. The `id` column 2 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 name and last name and for the school where they teach 3. 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` 4 is set to the data type `date`, and the `salary` column 5 is `numeric`. I'll cover data types more thoroughly in Chapter 4, but this table shows some common examples of data types. The code block wraps up 6 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:

. Open the pgAdmin Query Tool (if it's not open, click `analysis` once in pgAdmin's object browser, and then choose **Tools▶Query Tool**).

. Copy the `CREATE TABLE` script from *Listing 2-2* into the SQL Editor (or highlight the listing if you've elected to open the *Chapter_02.sql* file from GitHub with the Query Tool).

. Execute the script by clicking the **Execute/Refresh** icon (shaped like a right arrow).

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 **analysis** and choose **Refresh**. Choose **Schemas▶public▶Tables** to see your new table, as shown in *Figure 2-4*.

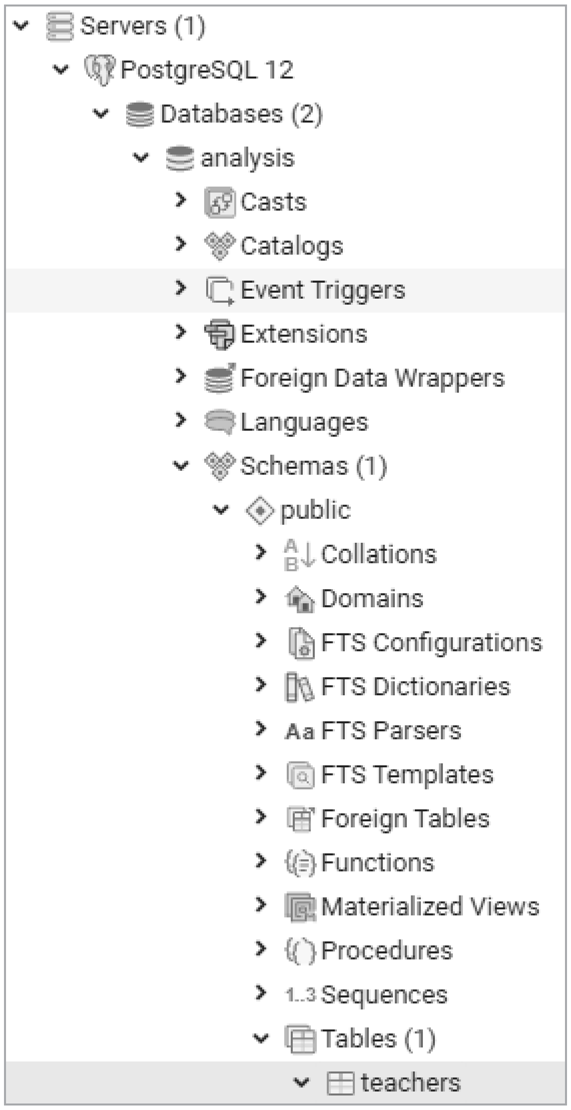


Figure 2-4: The `teachers` table in the object browser

Expand the `teachers` table node by clicking the arrow to the left of its name. This reveals more details about the table, including the column names, as shown in Figure 2-5. Other information appears as well, such as indexes, triggers, and constraints, but I'll cover those in later chapters. Clicking the table name and then selecting the **SQL** menu in the pgAdmin workspace will display SQL statements that would be used to re-create the `teachers` table (note that this display includes additional default notations that were implicitly added when you created the table).

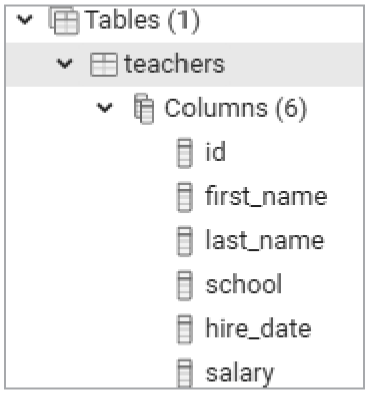


Figure 2-5: Table details for `teachers`

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.

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 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.

Using 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 you did to create the database and table, copy the code in Listing 2-3 into your pgAdmin Query Tool (or, if you opened the *Chapter_02.sql* file from GitHub in the Query Tool, highlight this listing).

```
1 INSERT INTO teachers (first_name, last_name, school,
  hire_date, salary)
2 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);3
```

Listing 2-3: 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 1. In the next row are the `VALUES` keyword and the data to insert into each column in each row 2. 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, except the last row, which ends the entire statement with a semicolon 3.

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 4 when I discuss data types.

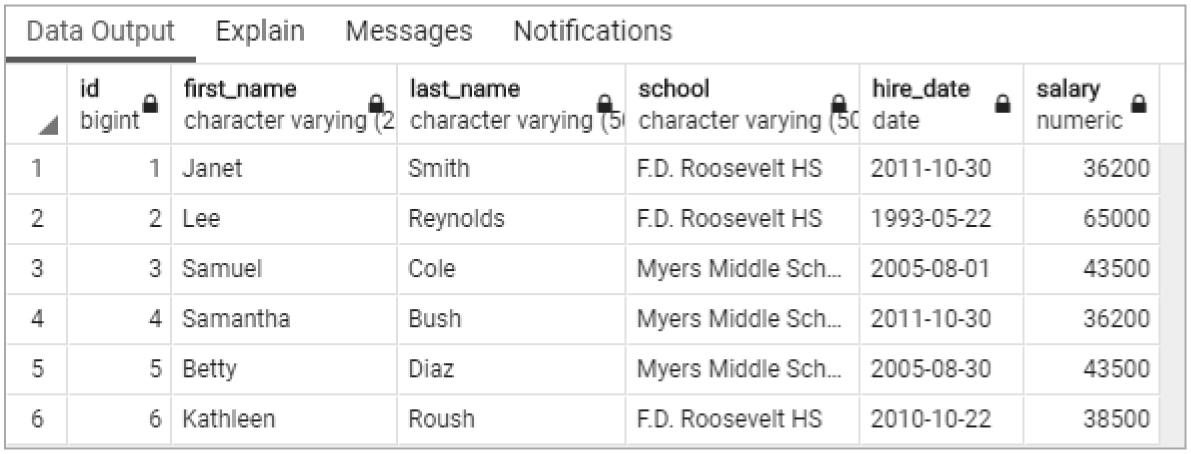
Now, run the code. This time, the message area of the Query Tool should say this:

```
INSERT 0 6
Query returned successfully in 150 msec.
```

The last of the two numbers after the `INSERT` keyword reports the number of rows inserted: `6`. The first number is an unused legacy PostgreSQL value that is returned only to maintain wire protocol; you can safely ignore it.

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 2-6* shows, you'll see the six rows of data in the table with each column filled by the values in the SQL statement.



	id bigint	first_name character varying (2	last_name character varying (5	school character varying (50	hire_date date	salary numeric
1	1	Janet	Smith	F.D. Roosevelt HS	2011-10-30	36200
2	2	Lee	Reynolds	F.D. Roosevelt HS	1993-05-22	65000
3	3	Samuel	Cole	Myers Middle Sch...	2005-08-01	43500
4	4	Samantha	Bush	Myers Middle Sch...	2011-10-30	36200
5	5	Betty	Diaz	Myers Middle Sch...	2005-08-30	43500
6	6	Kathleen	Roush	F.D. Roosevelt HS	2010-10-22	38500

Figure 2-6: 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. Also, each column header displays the data type you defined when creating the table. (Note that in this example, `varchar`, fully expanded in PostgreSQL, is `character varying`.) Seeing the data type in the results will help later when you decide how to write queries that handle data differently depending on its type.

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

Getting Help 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 2-3*, PostgreSQL squawks back an error:

```
ERROR:  syntax error at or near "("
LINE 4:     ('Samuel', 'Cole', 'Myers Middle School', '2005-
08-01', 43...
            ^
```

Fortunately, the error message hints at what's wrong and where: we made a syntax error near an open parenthesis on line 4. 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. I've found that I get the best search results by entering the error message verbatim in the search engine, specifying the name of my database manager, and limiting results to more recent items to avoid using outdated information.

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, here are several generally accepted 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 8).

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 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:

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?

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?

Solutions to all exercises are available in the *Try_It_Yourself.sql* file included with the book's resources.

3
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 résumé.

Interviewing the data 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 and then look into the more powerful things `SELECT` can do.

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*, which 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 2. Once again, open pgAdmin, select the `analysis` database, and open the Query Tool. Then execute the statement shown in *Listing 3-1*. Remember, as an alternative to typing these statements into the Query Tool, you can also run the code by clicking **Open File** and navigating to the place where you saved the code you downloaded from GitHub. Always do this if you see the code is truncated with `--snip--`. For this chapter, you should open *Chapter_03.sql* and highlight each statement before clicking the **Execute/Refresh** icon.

```
SELECT * FROM teachers;
```

Listing 3-1: Querying all rows and columns from the `teachers` table

Once you execute the query, the result set in the Query Tool's output pane contains all the rows and columns you inserted into the `teachers` table in Chapter 2. 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`) is automatically filled 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 later gives us a way to connect this table to other tables in the database.

Before we move on, note that you have two other ways to view all rows in a table. Using pgAdmin, you can right-click the `teachers` table in the object tree and choose **View/Edit Data▶All Rows**. Or you can use a little-known bit of standard SQL:

```
TABLE teachers;
```

Both provide the same result as the code in *Listing 3-1*. Now, let's refine this query to make it more specific.

Querying a Subset of Columns

Often, it's more practical to limit the columns the query retrieves, especially with large databases, so you don't have to wade through excess information. You can do this by naming columns, separated by commas, right after the `SELECT` keyword. Here's an 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. In that case, you would select just the relevant columns, as shown in Listing 3-2. 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 3-2: *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 dataset. Generally, it's wise to start your analysis by checking whether your data is present and in the format you expect, which is a task well suited to `SELECT`. Are dates in a proper format complete with month, date, and year, or are they entered (as I once ruefully observed) as text with the month and year only? Does every row have values in all the columns? Are there mysteriously no last names starting with letters beyond *M*? All these issues indicate potential hazards ranging from missing data to shoddy record keeping 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.

NOTE

pgAdmin allows you to drag and drop column names, table names, and other objects from the object browser into the Query Tool. This can be helpful if you're writing a new query and don't want to keep typing lengthy object names. Expand the object tree to find your tables or columns, as you did in Chapter 1, and click and drag them into the Query Tool.

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 3-3 shows an example using the `teachers` table.

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

Listing 3-3: 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
```

The `ORDER BY` clause also accepts numbers instead of column names, with the number identifying the sort column according to its position in the `SELECT` clause. Thus, you could rewrite Listing 3-3 this way, using `3` to refer to the third column in the `SELECT` clause, `salary`:

```
SELECT first_name, last_name, salary
FROM teachers
ORDER BY 3 DESC;
```

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 3-4.

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

Listing 3-4: 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 1, 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.

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 `DISTINCT` immediately after `SELECT`, as shown in Listing 3-5.

```
SELECT DISTINCT school
FROM teachers
ORDER BY school;
```

Listing 3-5: 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, especially if you sort the output.

When you're working with dates or numbers, `DISTINCT` will help highlight inconsistent or broken formatting. For example, you might inherit a dataset 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
---------
5/30/2023
6//2023
6/1/2023
6/2/2023
```

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 3-6.

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

Listing 3-6: 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
-------------------    ------
F.D. Roosevelt HS      36200
F.D. Roosevelt HS      38500
F.D. Roosevelt HS      65000
Myers Middle School    36200
Myers Middle School    43500
```

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 6 and Chapter 9.

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` clause allows you to find rows that match a specific value, a range of values, or multiple values based on criteria supplied via an *operator*—a keyword that lets us perform math, comparison, and logical operations. You also can use criteria to exclude rows.

Listing 3-7 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 3-7: 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 3-1* summarizes the most commonly used comparison operators. Depending on your database system, many more might be available.

Table 3-1: *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 following examples show comparison operators in action. First, we use the equal 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 from $40,000 to $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;
```

Use caution with `BETWEEN`, because its inclusive nature can lead to inadvertent double-counting of values. For example, if you filter for values with `BETWEEN 10 AND 20` and run a second query using `BETWEEN 20 AND 30`, a row with the value of 20 will appear in both query results. You can avoid this by using the more explicit greater-than and less-than operators to define ranges. For example, this query returns the same result as the previous one but more obviously specifies the range:

```
SELECT first_name, last_name, school, salary
FROM teachers
WHERE salary >= 40000 AND salary <= 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 the matching operators `LIKE` and `ILIKE` deserve additional explanation. Both let you find a variety of values that include characters matching a specified pattern, which is handy if you don't know exactly what you're searching for or if you're rooting out misspelled words. To use `LIKE` and `ILIKE`, you specify a pattern to match using one or both of these symbols:

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 3-8* shows how the two keywords give you different results. The first `WHERE` clause uses `LIKE` 1 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` 2, will return `Samuel` and `Samantha` from the table.

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

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

Listing 3-8: Filtering with `LIKE` and `ILIKE`

Over the years, I've gravitated toward using `ILIKE` and wildcard operators to make sure I'm not inadvertently excluding results from searches, particularly when vetting data. 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" in Chapter 8.

Combining Operators with AND and OR

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

The statements in Listing 3-9 show three examples that combine operators this way.

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

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

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

Listing 3-9: Combining operators using AND and OR

The first query uses AND in the WHERE clause 1 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 2 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 3. 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.

If we use both `AND` with `OR` in a clause but don't use any parentheses, the database will evaluate the `AND` condition first and then the `OR` condition. In the final example, that means we'd see a different result if we omitted parentheses—the database would look for rows where the school name is `F.D. Roosevelt HS` and the salary is less than $38,000 or rows for any school where the salary is more than $40,000. Give it a try in the Query Tool to see.

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 3-10 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 3-10: 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 some connection between a teacher's hire date at the school and their 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

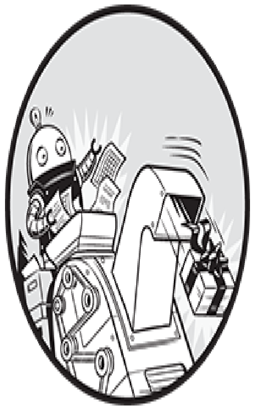
Explore basic queries with these exercises:

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.

Write a query that finds the one teacher whose first name starts with the letter *S* and who earns more than $40,000.

Rank teachers hired since January 1, 2010, ordered by highest paid to lowest.

4
UNDERSTANDING DATA TYPES



It's important to understand data types because storing data in the appropriate format is fundamental to building usable databases and performing accurate analysis. Whenever I dig into a new database, I check the *data type* specified for each column in each table. If I'm lucky, I can get my hands on a *data dictionary*: a document that lists each column; specifies whether it's a number, character, or other type; and explains the column values. Unfortunately, many organizations don't create and maintain good documentation, so it's not unusual to hear, "We don't have a data dictionary." In that case, I inspect the table structures in pgAdmin to learn as much as I can.

Data types are a programming concept applicable to more than just SQL. The concepts you'll explore in this chapter will transfer well to additional languages you may want to learn.

In a SQL database, each column in a table can hold one and only one data type, which you define in the `CREATE TABLE` statement by declaring the data type after the column name. In the following simple example table—which you can review but don't need to create—you will find columns with three different data types: a date, an integer, and text.

```
CREATE TABLE eagle_watch (
    observation_date date,
    eagles_seen integer,
    notes text
);
```

In this table named `eagle_watch` (for a hypothetical inventory of bald eagles), we declare the `observation_date` column to hold date values by adding the `date` type declaration after its name. Similarly, we set `eagles_seen` to hold whole numbers with the `integer` type declaration and declare `notes` to hold characters via the `text` type.

These data types fall into the three categories you'll encounter most:

Characters Any character or symbol

Numbers Includes whole numbers and fractions

Dates and times Temporal information

Let's look at each data type in depth; I'll note whether they're part of standard ANSI SQL or specific to PostgreSQL. An overall, in-depth look at where PostgreSQL deviates from the SQL standard is available at *https://wiki.postgresql.org/wiki/PostgreSQL_vs_SQL_Standard*.

Understanding Characters

Character string types are general-purpose types suitable for any combination of text, numbers, and symbols. Character types include the following:

`char(`*`n`*`)`

A fixed-length column where the character length is specified by `n`. A column set at `char(20)` stores 20 characters per row regardless of how many characters you insert. If you insert fewer than 20 characters in any row, PostgreSQL pads the rest of that column with spaces. This type, which is part of standard SQL, also can be specified with the longer name `character(n)`. Nowadays, `char(n)` is used infrequently and is mainly a remnant of legacy computer systems.

`varchar(n)`

A variable-length column where the *maximum* length is specified by `n`. If you insert fewer characters than the maximum, PostgreSQL will not store extra spaces. For example, the string `blue` will take four spaces, whereas the string `123` will take three. In large databases, this practice saves considerable space. This type, included in standard SQL, also can be specified using the longer name `character varying(n)`.

`text`

A variable-length column of unlimited length. (According to the PostgreSQL documentation, the longest possible character string you can store is about 1 gigabyte.) The `text` type is not part of the SQL standard, but you'll find similar implementations in other database systems, including Microsoft SQL Server and MySQL.

According to PostgreSQL documentation at *https://www.postgresql.org/docs/current/datatype-character.html*, there is no substantial difference in performance among the three types. That may differ if you're using another database manager, so it's wise to check the docs. The flexibility and potential space savings of `varchar` and `text` seem to give them an advantage. But if you search discussions online, some users suggest that defining a column that will always have the same number of characters with `char` is a good way to signal what data it should contain. For instance, you might see `char(2)` used for US state postal abbreviations.

NOTE

You cannot perform math operations on numbers stored in a character column. Store numbers as character types only when they represent codes, such as a US postal ZIP code.

To see these three character types in action, run the script shown in Listing 4-1. This script will build and load a simple table and then export the data to a text file on your computer.

```
  CREATE TABLE char_data_types (
   1 char_column char(10),
     varchar_column varchar(10),
     text_column text
  );

2 INSERT INTO char_data_types
  VALUES
      ('abc', 'abc', 'abc'),
      ('defghi', 'defghi', 'defghi');

3 COPY char_data_types TO 'C:\YourDirectory\typetest.txt'
4 WITH (FORMAT CSV, HEADER, DELIMITER '|');
```

Listing 4-1: Character data types in action

We define three character columns 1 of different types and insert two rows of the same string into each 2. Unlike the `INSERT INTO` statement you learned in Chapter 2, here we're not specifying the names of the columns. If the `VALUES` statements match the number of columns in the table, the database will assume you're inserting values in the order the column definitions were specified in the table.

Next, we use the PostgreSQL `COPY` keyword 3 to export the data to a text file named *typetest.txt* in a directory you specify. You'll need to replace *C:\YourDirectory\* with the full path to the directory on your computer where you want to save the file. The examples in this book use Windows format—which use a backslash between folders and file names—and a path to a directory called *YourDirectory* on the C: drive. Windows users must set permissions for the destination folder according to the note in the section "Downloading Code and Data from GitHub" in Chapter 1.

Linux and macOS file paths have a different format, with forward slashes between folders and filenames. On my Mac, for example, the path to a file on the desktop is */Users/anthony/Desktop/*. The directory must exist already; PostgreSQL won't create it for you.

NOTE

On Linux, you may see a permission denied *error when using COPY. That's because PostgreSQL runs as the* `postgres` *user, which can't read or write to another user's directory. One solution is to read or write from the system* /tmp *folder, accessible to all users. Be cautious, because some configurations cause this directory to be emptied upon reboot. For other options, see "Importing and Exporting Through pgAdmin" in Chapter 5 and "Importing, Exporting, and Using Files" with* `psql` *in Chapter 18.*

In PostgreSQL, `COPY` *`table_name`* `FROM` is the import function, and `COPY` *`table_name`* `TO` is the export function. I'll cover them in depth in Chapter 5; for now, all you need to know is that the `WITH` keyword options 4 will format the data in the file with each column separated by a *pipe* (`|`) character. That way, you can easily see where spaces fill out the unused portions of the `char` column.

To see the output, open *typetest.txt* using the text editor you installed in Chapter 1 (not Word or Excel, or another spreadsheet application). The contents should look like this:

```
char_column|varchar_column|text_column
abc       |abc|abc
defghi    |defghi|defghi
```

Even though you specified 10 characters for both the `char` and `varchar` columns, only the `char` column outputs 10 characters in both rows, padding unused characters with spaces. The `varchar` and `text` columns store only the characters you inserted.

Again, there's no real performance difference among the three types, although this example shows that `char` can potentially consume more storage space than needed. A few unused spaces in each column might seem negligible, but multiply that over millions of rows in dozens of tables and you'll soon wish you had been more economical.

I tend to use `text` on all my character columns. That saves me from having to configure maximum lengths for multiple `varchar` columns and means I won't need to modify a table later if the requirements for a character column change.

Understanding Numbers

Number columns hold various types of (you guessed it) numbers, but that's not all: they also allow you to perform calculations on those numbers. That's an important distinction from numbers you store as strings in a character column, which can't be added, multiplied, divided, or perform any other math operation. Also, numbers stored as characters sort differently than numbers stored as numbers, so if you're doing math or the numeric order is important, use number types.

The SQL number types include the following:

Integers Whole numbers, both positive and negative

Fixed-point and floating-point Two formats of fractions of whole numbers

We'll look at each type separately.

Using Integers

The integer data types are the most common number types you'll find when exploring a SQL database. These are *whole numbers*, both positive and negative, including zero. Think of all the places integers appear in life: your street or apartment number, the serial number on your refrigerator, the number on a raffle ticket.

The SQL standard provides three integer types: `smallint`, `integer`, and `bigint`. The difference between the three types is the maximum size of the numbers they can hold. *Table 4-1* shows the upper and lower limits of each, as well as how much storage each requires in bytes.

Table 4-1: *Integer Data Types*

Data type	Storage size	Range
`smallint`	2 bytes	−32768 to +32767
`integer`	4 bytes	−2147483648 to +2147483647
`bigint`	8 bytes	−9223372036854775808 to +9223372036854775807

The `bigint` type will cover just about any requirement you'll ever have with a number column, though it eats up the most storage. Its use is a must if you're working with numbers larger than about 2.1 billion, but you also can easily make it your go-to default and never worry about not being able to fit a number in the column. On the other hand, if you're confident numbers will remain within the `integer` limit, that type is a good choice because it doesn't consume as much space as `bigint` (a concern when dealing with millions of data rows).

When you know that values will remain constrained, `smallint` makes sense: days of the month or years are good examples. The `smallint` type will use half the storage as `integer`, so it's a smart database design decision if the column values will always fit within its range.

If you try to insert a number into any of these columns that is outside its range, the database will stop the operation and return an `out of range` error.

Auto-Incrementing Integers

Sometimes, it's helpful to create a column that holds integers that *auto-increment* each time you add a row to the table. For example, you might use an auto-incrementing column to create a unique ID number, also known as a *primary key*, for each row in the table. Each row then has its own ID that other tables in the database can reference, a concept I'll cover in Chapter 7.

With PostgreSQL, you have two ways to auto-increment an integer column. One is the *serial* data type, a PostgreSQL-specific implementation of the ANSI SQL standard for auto-numbered *identity columns*. The other is the ANSI SQL standard `IDENTITY` keyword. Let's start with serial.

Auto-Incrementing with serial

In Chapter 2, when you made the `teachers` table, you created an `id` column with the declaration of `bigserial`: this and its siblings `smallserial` and `serial` are not so much true data types as a special *implementation* of the corresponding `smallint`, `integer`, and `bigint` types. When you add a column with a serial type, PostgreSQL will auto-increment the value each time you insert a row, starting with 1, up to the maximum of each integer type.

Table 4-2 shows the serial types and the ranges they cover.

Table 4-2: *Serial Data Types*

Data type	Storage size	Range
`smallserial`	2 bytes	1 to 32767
`serial`	4 bytes	1 to 2147483647
`bigserial`	8 bytes	1 to 9223372036854775807

To use a serial type on a column, declare it in the `CREATE TABLE` statement as you would an integer type. For example, you could create a table called `people` that has an `id` column equivalent in size to the `integer` data type:

```
CREATE TABLE people (
    id serial,
    person_name varchar(100)
);
```

Every time a new row with a `person_name` is added to the table, the `id` column will increment by 1.

Auto-Incrementing with IDENTITY

As of version 10, PostgreSQL includes support for `IDENTITY`, the standard SQL implementation for auto-incrementing integers. The `IDENTITY` syntax is more verbose, but some database users prefer it for its cross-compatibility with other database systems (such as Oracle) and also because it has an option to prevent users from accidentally inserting values in the auto-incrementing column (which serial types will permit).

You can specify `IDENTITY` in two ways:

- `GENERATED ALWAYS AS IDENTITY` tells the database to always fill the column with an auto-incremented value. A user cannot insert a value into the `id` column without manually overriding that setting. See the `OVERRIDING SYSTEM VALUE` section of the PostgreSQL `INSERT` documentation at *https://www.postgresql.org/docs/current/sql-insert.html* for details.
- `GENERATED BY DEFAULT AS IDENTITY` tells the database to fill the column with an auto-incremented value by default if the user does not supply one. This option allows for the possibility of duplicate values, which can make use of it problematic for creating key columns. I'll delve into that more in Chapter 7.

For now, we'll stick with the first option, using `ALWAYS`. To create a table called `people` that has an `id` column populated via `IDENTITY`, you would use this syntax:

```
CREATE TABLE people (
    id integer GENERATED ALWAYS AS IDENTITY,
    person_name varchar(100)
);
```

For the `id` data type, we use `integer` followed by the keywords `GENERATED ALWAYS AS IDENTITY`. Now, every time we insert a `person_name` value into the table, the database will fill the `id` column with an auto-incremented value.

Given its compatibility with the ANSI SQL standard, I'll use `IDENTITY` for the remainder of the book.

NOTE

Even though the value in an auto-incrementing column increases each time a row is added, some scenarios will create gaps in the sequence of numbers in the column. If a row is deleted, for example, the value in that row is never replaced. Or, if a row insert is aborted, the sequence for the column will still be incremented.

Using Decimal Numbers

Decimals represent a whole number plus a fraction of a whole number; the fraction is represented by digits following a *decimal point*. In a SQL database, they're handled by *fixed-point* and *floating-point* data types. For example, the distance from my house to the nearest grocery store is 6.7 miles; I could insert 6.7 into either a fixed-point or floating-point column with no complaint from PostgreSQL. The only difference is how the computer stores the data. In a moment, you'll see that has important implications.

Understanding Fixed-Point Numbers

The fixed-point type, also called the *arbitrary precision* type, is `numeric(`*`precision`*`,`*`scale`*`)`. You give the argument `precision` as the maximum number of digits to the left and right of the decimal point, and the argument `scale` as the number of digits allowable on the right of the decimal point. Alternately, you can specify this type using `decimal(`*`precision`*`,`*`scale`*`)`. Both are part of the ANSI SQL standard. If you omit specifying a scale value, the scale will be set to zero; in effect, that creates an integer. If you omit specifying the precision and the scale, the database will store values of any precision and scale up to the maximum allowed. (That's up to 131,072 digits before the decimal point and 16,383 digits after the decimal point, according to the PostgreSQL documentation at *https://www.postgresql.org/docs/current/datatype-numeric.html*.)

For example, let's say you're collecting rainfall totals from several local airports—not an unlikely data analysis task. The US National Weather Service provides this data with rainfall typically measured to two decimal places. (And, if you're like me, you have a distant memory of your primary school math teacher explaining that two digits after a decimal is the hundredths place.)

To record rainfall in the database using five digits total (the precision) and two digits maximum to the right of the decimal (the scale), you'd specify it as `numeric(5,2)`. The database will always return two digits to the right of the decimal point, even if you don't enter a number that contains two digits such as 1.47, 1.00, and 121.50.

Understanding Floating-Point Types

The two floating-point types are `real` and `double precision`, both part of the SQL standard. The difference between the two is how much data they store. The `real` type allows precision to six decimal digits, and `double precision` to 15 decimal digits of precision, both of which include the number of digits on both sides of the point. These floating-point types are also called *variable-precision* types. The database stores the number in parts representing the digits and an exponent—the location where the decimal point belongs. So, unlike `numeric`, where we specify fixed precision and scale, the decimal point in a given column can "float" depending on the number.

Using Fixed- and Floating-Point Types

Each type has differing limits on the number of total digits, or precision, it can hold, as shown in *Table 4-3*.

Table 4-3: *Fixed-Point and Floating-Point Data Types*

Data type	Storage size	Storage type	Range
`numeric`, `decimal`	Variable	Fixed-point	Up to 131,072 digits before the decimal point; up to 16,383 digits after the decimal point
`real`	4 bytes	Floating-point	6 decimal digits precision
`double precision`	8 bytes	Floating-point	15 decimal digits precision

To see how each of the three data types handles the same numbers, create a small table and insert a variety of test cases, as shown in *Listing 4-2*.

```
CREATE TABLE number_data_types (
 1 numeric_column numeric(20,5),
    real_column real,
    double_column double precision
);

2 INSERT INTO number_data_types
VALUES
    (.7, .7, .7),
    (2.13579, 2.13579, 2.13579),
    (2.1357987654, 2.1357987654, 2.1357987654);

SELECT * FROM number_data_types;
```

Listing 4-2: Number data types in action

We create a table with one column for each of the fractional data types 1 and load three rows into the table 2. Each row repeats the same number across all three columns. When the last line of the script runs and we select everything from the table, we get the following:

```
numeric_column    real_column    double_column
--------------    -----------    -------------
       0.70000            0.7              0.7
       2.13579        2.13579          2.13579
       2.13580      2.1357987     2.1357987654
```

Notice what happened. The `numeric` column, set with a scale of five, stores five digits after the decimal point whether or not you inserted that many. If fewer than five, it pads the rest with zeros. If more than five, it rounds them—as with the third-row number with 10 digits after the decimal.

The `real` and `double precision` columns add no padding. On the third row, you see PostgreSQL's default behavior in those two columns, which is to output floating-point numbers using their shortest precise decimal representation rather than show the entire value. Note that older versions of PostgreSQL may display slightly different results.

Running into Trouble with Floating-Point Math

If you're thinking, "Well, numbers stored as a floating-point look just like numbers stored as fixed," tread cautiously. The way computers store floating-point numbers can lead to unintended mathematical errors. Look at what happens when we do some calculations on these numbers. Run the script in Listing 4-3.

```
SELECT
 1 numeric_column * 10000000 AS fixed,
       real_column * 10000000 AS floating
   FROM number_data_types
2 WHERE numeric_column = .7;
```

Listing 4-3: Rounding issues with float columns

Here, we multiply the `numeric_column` and the `real_column` by 10 million 1 and use a `WHERE` clause to filter out just the first row 2. We should get the same result for both calculations, right? Here's what the query returns:

```
fixed              floating
-------------      ----------------
7000000.00000      6999999.88079071
```

Hello! No wonder floating-point types are referred to as "inexact." It's a good thing I'm not using this math to launch a mission to Mars or calculate the federal budget deficit.

The reason floating-point math produces such errors is that the computer attempts to squeeze lots of information into a finite number of bits. The topic is the subject of a lot of writings and is beyond the scope of this book, but if you're interested, you'll find the link to a good synopsis at *https://www.nostarch.com/practical-sql-2nd-edition/*.

The storage required by the `numeric` data type is variable, and depending on the precision and scale specified, `numeric` can consume considerably more space than the floating-point types. If you're working with millions of rows, it's worth considering whether you can live with relatively inexact floating-point math.

Choosing Your Number Data Type

For now, here are three guidelines to consider when you're dealing with number data types:

Use integers when possible. Unless your data uses decimals, stick with integer types.

If you're working with decimal data and need calculations to be exact (dealing with money, for example), choose `numeric` or its equivalent, `decimal`. Float types will save space, but the inexactness of floating-point math won't pass muster in many applications. Use them only when exactness is not as important.

Choose a big enough number type. Unless you're designing a database to hold millions of rows, err on the side of bigger. When using `numeric` or `decimal`, set the precision large enough to accommodate the number of digits on both sides of the decimal point. With whole numbers, use `bigint` unless you're absolutely sure column values will be constrained to fit into the smaller `integer` or `smallint` type.

Understanding Dates and Times

Whenever you enter a date into a search form, you're reaping the benefit of databases having an awareness of the current time (received from the server) plus the ability to handle formats for dates, times, and the nuances of the calendar, such as leap years and time zones. This is essential for storytelling with data, because the issue of *when* something occurred is usually as valuable a question as who, what, or how many were involved.

PostgreSQL's date and time support includes the four major data types shown in *Table 4-4*.

Table 4-4: *Date and Time Data Types*

Data type	Storage size	Description	Range
`timestamp`	8 bytes	Date and time	4713 BC to 294276 AD
`date`	4 bytes	Date (no time)	4713 BC to 5874897 AD
`time`	8 bytes	Time (no date)	00:00:00 to 24:00:00
`interval`	16 bytes	Time interval	+/− 178,000,000 years

Here's a rundown of data types for times and dates in PostgreSQL:

`timestamp` Records date and time, which are useful for a range of situations you might track: departures and arrivals of passenger flights, a schedule of Major League Baseball games, or incidents along a timeline. You will almost always want to add the keywords `with time zone` to ensure that the time recorded for an event includes the time zone where it occurred. Otherwise, times recorded in various places around the globe become impossible to compare. The format `timestamp with time zone` is part of the SQL standard; with PostgreSQL you can specify the same data type using `timestamptz`.

`date` Records just the date. Part of the SQL standard.

`time` Records just the time and is part of the SQL standard. Although you can add the `with time zone` keywords, without a date the time zone will be meaningless.

`interval` Holds a value representing a unit of time expressed in the format *`quantity unit`*. It doesn't record the start or end of a time period, only its length. Examples include `12 days` or `8 hours`. (The PostgreSQL documentation at *https://www.postgresql.org/docs/current/datatype-datetime.html* lists unit values ranging from `microsecond` to `millennium`.) You'll typically use this type for calculations or filtering on other date and time columns. It's also part of the SQL standard, although PostgreSQL-specific syntax offers more options.

Let's focus on the `timestamp with time zone` and `interval` types. To see these in action, run the script in *Listing 4-4*.

```
1 CREATE TABLE date_time_types (
       timestamp_column timestamp with time zone,
       interval_column interval
  );

2 INSERT INTO date_time_types
  VALUES
       ('2022-12-31 01:00 EST','2 days'),
       ('2022-12-31 01:00 -8','1 month'),
       ('2022-12-31 01:00 Australia/Melbourne','1 century'),
    3 (now(),'1 week');

  SELECT * FROM date_time_types;
```

Listing 4-4: The `timestamp` and `interval` types in action

Here, we create a table with a column for both types 1 and insert four rows 2. For the first three rows, our insert for the `timestamp_column` uses the same date and time (December 31, 2022 at 1 AM) using the

International Organization for Standardization (ISO) format for dates and times: `YYYY-MM-DD HH:MM:SS`. SQL supports additional date formats (such as `MM/DD/YYYY`), but ISO is recommended for portability worldwide.

Following the time, we specify a time zone but use a different format in each of the first three rows: in the first row, we use the abbreviation `EST`, which is Eastern standard time in the United States.

In the second row, we set the time zone with the value `-8`. That represents the number of hours difference, or *offset*, from Coordinated Universal Time (UTC), the time standard for the world. The value of UTC is +/− 00:00, so `-8` specifies a time zone eight hours behind UTC. In the United States, when daylight saving time is in effect, `-8` is the value for the Alaska time zone. From November through early March, when the United States reverts to standard time, it refers to the Pacific time zone. (For a map of UTC time zones, see *https://en.wikipedia.org/wiki/Coordinated_Universal_Time#/media/File:Standard_World_Time_Zones.tif*.)

For the third row, we specify the time zone using the name of an area and location: `Australia/Melbourne`. That format uses values found in a standard time zone database often employed in computer programming. You can learn more about the time zone database at *https://en.wikipedia.org/wiki/Tz_database*.

In the fourth row, instead of specifying dates, times, and time zones, the script uses PostgreSQL's `now()` function 3, which captures the current transaction time from your hardware.

After the script runs, the output should look similar to (but not exactly like) this:

```
timestamp_column                 interval_column
-----------------------------    ---------------
2022-12-31 01:00:00-05           2 days
2022-12-31 04:00:00-05           1 mon
2022-12-30 09:00:00-05           100 years
2020-05-31 21:31:15.716063-05    7 days
```

Even though we supplied the same date and time in the first three rows on the `timestamp_column`, each row's output differs. The reason is that pgAdmin reports the date and time relative to my time zone, which in the results shown is indicated by the UTC offset of `-05` at the end of each timestamp. A UTC offset of `-05` means five hours behind UTC, equivalent to the US Eastern time zone during fall and winter months when standard time is observed. If you live in a different time zone, you'll likely see a different offset; the times and dates also may differ from what's shown here. We can change how PostgreSQL reports these timestamp values, and I'll cover how to do that plus other tips for wrangling dates and times in Chapter 12.

Finally, the `interval_column` shows the values you entered. PostgreSQL changed `1 century` to `100 years` and `1 week` to `7 days` because of its preferred default settings for interval display. Read the "Interval Input" section of the PostgreSQL documentation at *https://www.postgresql.org/docs/current/datatype-datetime.html* to learn more about options related to intervals.

Using the interval Data Type in Calculations

The `interval` data type is useful for easy-to-understand calculations on date and time data. For example, let's say you have a column that holds the date a client signed a contract. Using interval data, you can add 90 days to each contract date to determine when to follow up with the client.

To see how the `interval` data type works, we'll use the `date_time_types` table we just created, as shown in Listing 4-5.

```
SELECT
    timestamp_column,
    interval_column,
 1 timestamp_column - interval_column AS new_date
FROM date_time_types;
```

Listing 4-5: Using the `interval` data type

This is a typical `SELECT` statement, except we'll compute a column called `new_date` 1 that contains the result of `timestamp_column` minus `interval_column`. (Computed columns are called *expressions*; we'll use this technique often.) In each row, we subtract the unit of time indicated by the `interval` data type from the date. This produces the following result:

```
timestamp_column                 interval_column    new_date
-----------------------------    ---------------    ---------
--------------------
2022-12-31 01:00:00-05           2 days             2022-12-
29 01:00:00-05
2022-12-31 04:00:00-05           1 mon              2022-11-
30 04:00:00-05
2022-12-30 09:00:00-05           100 years          1922-12-
30 09:00:00-05
2020-05-31 21:31:15.716063-05    7 days             2020-05-
24 21:31:15.716063-05
```

Note that the `new_date` column by default is formatted as type `timestamp with time zone`, allowing for the display of time values as well as dates if the interval value uses them. (You can see the data type listed in the pgAdmin results grid, listed beneath the column names.) Again, your output may be different based on your time zone.

Understanding JSON and JSONB

JSON, short for *JavaScript Object Notation*, is a structured data format used for both storing data and exchanging data between computer systems. All major programming languages support reading and writing data in JSON format, which organizes information in a collection of *key/value* pairs as well as lists of values. Here's a simple example:

```
{
  "business_name": "Old Ebbitt Grill",
  "business_type": "Restaurant",
  "employees": 300,
  "address": {
    "street": "675 15th St NW",
    "city": "Washington",
    "state": "DC",
    "zip_code": "20005"
  }
}
```

This snippet of JSON shows the format's basic structure. A *key*, for example `business_name`, is associated with a *value*—in this case, `Old Ebbitt Grill`. A key also can have as its value a collection of additional key/value pairs, as shown with `address`. The JSON standard enforces rules about formatting, such as separating keys and values with a colon and enclosing key names in double quotes. You can use online tools such as *https://jsonlint.com/* to check whether a JSON object has valid formatting.

PostgreSQL currently offers two data types for JSON, which both enforce valid JSON and support functions for working with data in that format:

`json` Stores an exact copy of the JSON text

`jsonb` Stores the JSON text in a binary format

There are significant differences between the two. For example, `jsonb` supports indexing, which can improve processing speed.

JSON entered the SQL standard in 2016, but PostgreSQL added support several years earlier, starting with version 9.2. PostgreSQL currently implements several functions found in the SQL standard but offers its own additional JSON functions and operators. We'll cover these as well as both types more extensively in Chapter 16.

Using Miscellaneous Types

Character, number, and date/time types will likely comprise the bulk of the work you do with SQL. But PostgreSQL supports many additional types, including but not limited to the following:

A *Boolean* type that stores a value of `true` or `false`

Geometric types that include points, lines, circles, and other two-dimensional objects

Text search types for PostgreSQL's full-text search engine

Network address types, such as IP or MAC addresses

A *universally unique identifier* (*UUID*) type, sometimes used as a unique key value in tables

Range types, which let you specify a range of values, such as integers or timestamps

Types for storing *binary* data

An *XML* data type that stores information in that structured format

I'll cover these types as required throughout the book.

Transforming Values from One Type to Another with CAST

Occasionally, you may need to transform a value from its stored data type to another type. For example, you may want to retrieve a number as a character so you can combine it with text. Or you might need to convert dates stored as characters into an actual date type so you can sort them in date order or perform interval calculations. You can perform these conversions using the `CAST()` function.

The `CAST()` function succeeds only when the target data type can accommodate the original value. Casting an integer as text is possible, because the character types can include numbers. Casting text with letters of the alphabet as a number is not.

Listing 4-6 has three examples using the three data type tables we just created. The first two examples work, but the third will try to perform an invalid type conversion so you can see what a type casting error looks like.

```
1 SELECT timestamp_column, CAST(timestamp_column AS varchar(10))
   FROM date_time_types;

2 SELECT numeric_column,
          CAST(numeric_column AS integer),
          CAST(numeric_column AS text)
   FROM number_data_types;

3 SELECT CAST(char_column AS integer) FROM char_data_types;
```

Listing 4-6: *Three* `CAST()` *examples*

The first SELECT statement 1 returns the timestamp_column value as a varchar, which you'll recall is a variable-length character column. In this case, I've set the character length to 10, which means when converted to a character string, only the first 10 characters are kept. That's handy in this case, because that just gives us the date segment of the column and excludes the time. Of course, there are better ways to remove the time from a timestamp, and I'll cover those in "Extracting the Components of a timestamp Value" in Chapter 12.

The second SELECT statement 2 returns the numeric_column value three times: in its original form and then as an integer and as text. Upon conversion to an integer, PostgreSQL rounds the value to a whole number. But with the text conversion, no rounding occurs.

The final SELECT3 doesn't work: it returns an error of invalid input syntax for type integer because letters can't become integers!

Using CAST Shortcut Notation

It's always best to write SQL that can be read by another person who might pick it up later, and the way CAST() is written makes what you intended when you used it fairly obvious. However, PostgreSQL also offers a less-obvious shortcut notation that takes less space: the *double colon*.

Insert the double colon in between the name of the column and the data type you want to convert it to. For example, these two statements cast timestamp_column as a varchar:

```
SELECT timestamp_column, CAST(timestamp_column AS
varchar(10))
FROM date_time_types;

SELECT timestamp_column::varchar(10)
FROM date_time_types;
```

Use whichever suits you, but be aware that the double colon is a PostgreSQL-only implementation not found in other SQL variants, and so won't port.

Wrapping Up

You’re now equipped to better understand the nuances of the data formats you encounter while digging into databases. If you come across monetary values stored as floating-point numbers, you’ll be sure to convert them to decimals before performing any math. And you’ll know how to use the right kind of text column to keep your database from growing too big.

Next, I’ll continue with SQL foundations and show you how to import external data into your database.

TRY IT YOURSELF

Continue exploring data types with these exercises:

Your company delivers fruit and vegetables to local grocery stores, and you need to track the mileage driven by each driver each day to a tenth of a mile. Assuming no driver would ever travel more than 999 miles in a day, what would be an appropriate data type for the mileage column in your table? Why?

In the table listing each driver in your company, what are appropriate data types for the drivers’ first and last names? Why is it a good idea to separate first and last names into two columns rather than having one larger name column?

Assume you have a text column that includes strings formatted as dates. One of the strings is written as `'4//2021'`. What will happen when you try to convert that string to the `timestamp` data type?

5
IMPORTING AND EXPORTING DATA

So far, you've learned how to add a handful of rows to a table using SQL `INSERT` statements. A row-by-row insert is useful for making quick test tables or adding a few rows to an existing table. But it's more likely you'll need to load hundreds, thousands, or even millions of rows, and no one wants to write separate `INSERT` statements in those situations. Fortunately, you don't have to.

If your data exists in a *delimited* text file, with one table row per line of text and each column value separated by a comma or other character, PostgreSQL can import the data in bulk via its `COPY` command. This command is a PostgreSQL-specific implementation with options for including or excluding columns and handling various delimited text types.

In the opposite direction, `COPY` will also *export* data from PostgreSQL tables or from the result of a query to a delimited text file. This technique is handy when you want to share data with colleagues or move it into another format, such as an Excel file.

I briefly touched on `COPY` for export in the "Understanding Characters" section of Chapter 4, but in this chapter, I'll discuss import and export in more depth. For importing, I'll start by introducing you to one of my favorite datasets: annual US Census population estimates by county.

Three steps form the outline of most of the imports you'll do:

1. Obtain the source data in the form of a delimited text file.
2. Create a table to store the data.
3. Write a `COPY` statement to perform the import.

After the import is done, we'll check the data and look at additional options for importing and exporting.

A delimited text file is the most common file format that's portable across proprietary and open source systems, so we'll focus on that file type. If you want to transfer data from another database program's proprietary format directly to PostgreSQL—for example, from Microsoft Access or MySQL—you'll need to use a third-party tool. Check the PostgreSQL wiki at *https://wiki.postgresql.org/wiki/* and search for "Converting from other databases to PostgreSQL" for a list of tools and options.

If you're using SQL with another database manager, check the other database's documentation for how it handles bulk imports. The MySQL database, for example, has a `LOAD DATA INFILE` statement, and Microsoft's SQL Server has its own `BULK INSERT` command.

Working with Delimited Text Files

Many software applications store data in a unique format, and translating one data format to another is about as easy as trying to read the Cyrillic alphabet when one understands only English. Fortunately, most software can import from and export to a delimited text file, which is a common data format that serves as a middle ground.

A delimited text file contains rows of data, each of which represents one row in a table. In each row, each data column is separated, or delimited, by a particular character. I've seen all kinds of characters used as delimiters, from ampersands to pipes, but the comma is most commonly used; hence the name of a file type you'll see often is *comma-separated values (CSV)*. The terms *CSV* and *comma-delimited* are interchangeable.

Here's a typical data row you might see in a comma-delimited file:

```
John,Doe,123 Main St.,Hyde Park,NY,845-555-1212
```

Notice that a comma separates each piece of data—first name, last name, street, town, state, and phone—without any spaces. The commas tell the software to treat each item as a separate column, upon either import or export. Simple enough.

Handling Header Rows

A feature you'll often find inside a delimited text file is a *header row*. As the name implies, it's a single row at the top, or *head*, of the file that lists the name of each data column. Often, a header is added when data is exported from a database or a spreadsheet. Here's an example with the delimited row I've been using. Each item in a header row corresponds to its respective column:

```
FIRSTNAME,LASTNAME,STREET,CITY,STATE,PHONE
John,Doe,123 Main St.,Hyde Park,NY,845-555-1212
```

Header rows serve a few purposes. For one, the values in the header row identify the data in each column, which is particularly useful when you're deciphering a file's contents. Second, some database managers (although not PostgreSQL) use the header row to map columns in the delimited file to the correct columns in the import table. PostgreSQL doesn't use the header row, so we don't want to import that row to a table. We use the `HEADER` option in the `COPY` command to exclude it. I'll cover this with all `COPY` options in the next section.

Quoting Columns That Contain Delimiters

Using commas as a column delimiter leads to a potential dilemma: what if the value in a column includes a comma? For example, sometimes people combine an apartment number with a street address, as in 123 Main St., Apartment 200. Unless the system for delimiting accounts for that extra comma, during import the line will appear to have an extra column and cause the import to fail.

To handle such cases, delimited files use an arbitrary character called a *text qualifier* to enclose a column that includes the delimiter character. This acts as a signal to ignore that delimiter and treat everything between the text qualifiers as a single column. Most of the time in comma-delimited files the text qualifier used is the double quote. Here's the example data again, but with the street name column surrounded by double quotes:

```
FIRSTNAME,LASTNAME,STREET,CITY,STATE,PHONE
John,Doe,"123 Main St., Apartment 200",Hyde Park,NY,845-555-
1212
```

On import, the database will recognize that double quotes signify one column regardless of whether it finds a delimiter within the quotes. When importing CSV files, PostgreSQL by default ignores delimiters inside double-quoted columns, but you can specify a different text qualifier if your import requires it. (And, given the sometimes-odd choices made by IT professionals, you may indeed need to employ a different character.)

Finally, in CSV mode, if PostgreSQL finds two consecutive text qualifiers inside a double-quoted column, it will remove one. For example, let's say PostgreSQL finds this:

```
"123 Main St."" Apartment 200"
```

If so, it will treat that text as a single column upon import, leaving just one of the qualifiers:

```
123 Main St." Apartment 200
```

A situation like that could indicate an error in the formatting of your CSV file, which is why, as you'll see later, it's always a good idea to review your data after importing.

Using COPY to Import Data

To import data from an external file into our database, we first create a table in our database that matches the columns and data types in our source file. Once that's done, the `COPY` statement for the import is just the three lines of code in Listing 5-1.

```
1 COPY table_name
2 FROM 'C:\YourDirectory\your_file.csv'
3 WITH (FORMAT CSV, HEADER);
```

Listing 5-1: Using `COPY` for data import

We start the block of code with the `COPY` keyword 1 followed by the name of the target table, which must already exist in your database. Think of this syntax as meaning, "Copy data to my table called *`table_name`*."

The `FROM` keyword 2 identifies the full path to the source file, and we enclose the path in single quotes. The way you designate the path depends on your operating system. For Windows, begin with the drive letter, colon, backslash, and directory names. For example, to import a file located on my Windows desktop, the `FROM` line would read as follows:

```
FROM 'C:\Users\Anthony\Desktop\my_file.csv'
```

On macOS or Linux, start at the system root directory with a forward slash and proceed from there. Here's what the `FROM` line might look like when importing a file located on my macOS desktop:

```
FROM '/Users/anthony/Desktop/my_file.csv'
```

For the examples in the book, I use the Windows-style path *`C:\YourDirectory\`* as a placeholder. Replace that with the path where you stored the CSV file you downloaded from GitHub.

The `WITH` keyword 3 lets you specify options, surrounded by parentheses, that you use to tailor your input or output file. Here we specify that the external file should be comma-delimited and that we should exclude the file's header row in the import. It's worth examining all the options in the official PostgreSQL documentation at *https://www.postgresql.org/docs/current/sql-copy.html*, but here is a list of the options you'll commonly use:

Input and output file format

Use the `FORMAT` *`format_name`* option to specify the type of file you're reading or writing. Format names are `CSV`, `TEXT`, or `BINARY`. Unless you're deep into building technical systems, you'll rarely encounter a need to work with `BINARY`, where data is stored as a sequence of bytes. More often, you'll work with standard CSV files. In the `TEXT` format, a *tab* character is the delimiter by default (although you can specify another character), and backslash characters such as `\r` are recognized as their ASCII equivalents—in this case, a carriage return. The `TEXT` format is used mainly by PostgreSQL's built-in backup programs.

Presence of a header row

On import, use `HEADER` to specify that the source file has a header row that you want to exclude. The database will start importing with the second line of the file so that the column names in the header don't become part of the data in the table. (Be sure to check your source CSV to make sure this is what you want; not every CSV comes with a header row!) On export, using `HEADER` tells the database to include the column names as a header row in the output file, which helps a user understand the file's contents.

Delimiter

The `DELIMITER '`*`character`*`'` option lets you specify which character your import or export file uses as a delimiter. The delimiter must be a single character and cannot be a carriage return. If you use `FORMAT CSV`, the assumed delimiter is a comma. I include `DELIMITER` here to show that you have the option to specify a different delimiter if that's how your data arrived. For example, if you received pipe-delimited data, you would treat the option this way: `DELIMITER '|'`.

Quote character

Earlier, you learned that in a CSV file, commas inside a single column value will mess up your import unless the column value is surrounded by a character that serves as a text qualifier, telling the database to handle the value within as one column. By default, PostgreSQL uses the double quote, but if the CSV you're importing uses a different character for the text qualifier, you can specify it with the `QUOTE '`*`quote_character`*`'` option.

Now that you better understand delimited files, you're ready to import one.

Importing Census Data Describing Counties

The dataset you'll work with in this import exercise is considerably larger than the `teachers` table you made in Chapter 2. It contains census population estimates for every county in the United States and is 3,142 rows deep and 16 columns wide. (Census counties include some geographies with other names: parishes in Louisiana, boroughs and census areas in Alaska, and cities, particularly in Virginia.)

To understand the data, it helps to know a little about the US Census Bureau, a federal agency that tracks the nation's demographics. Its best-known program is a full count of the population it undertakes every 10 years, most recently in 2020. That data, which enumerates the age, gender, race, and ethnicity of each person in the country, is used to determine how many members from each state make up the 435-member US House of Representatives. In recent decades, faster-growing states such as Texas and Florida have gained seats, while slower-growing states such as New York and Ohio have lost representatives in the House.

The data we'll work with are the census' annual population estimates. These use the most recent 10-year census count as a base, and they factor in births, deaths, and domestic and international migration to produce population estimates each year for the nation, states, counties, and other geographies. In lieu of an annual physical count, it's the best way to get an updated measure on how many people live where in the United States. For this exercise, I compiled select columns from the 2019 US Census county-level population estimates (plus a few descriptive columns from census geographic data) into a file named *us_counties_pop_est_2019.csv*. You should have this file on your computer if you followed the directions in the section "Downloading Code and Data from GitHub" in Chapter 1. If not, go back and do that now.

NOTE

The 2019-vintage population estimates we're using do not reflect the split in 2019 of the former Valdez-Cordova census area into two new Alaska county equivalents. That change increased the number of US counties to 3,143.

Open the file with a text editor. You should see a header row that begins with these columns:

```
state_fips,county_fips,region,state_name,county_name, --snip--
```

Let's explore the columns by examining the code for creating the import table.

Creating the us_counties_pop_est_2019 Table

The code in *Listing 5-2* shows the `CREATE TABLE` script. In pgAdmin click the `analysis` database that you created in Chapter 2. (It's best to store the data in this book in `analysis` because we'll reuse some of it in later chapters.) From the pgAdmin menu bar, select **Tools▶Query Tool**. You can type the code into the tool or copy and paste it from the files you downloaded from GitHub. Once you have the script in the window, run it.

```
CREATE TABLE us_counties_pop_est_2019 (
 1 state_fips text,
    county_fips text,
 2 region smallint,
 3 state_name text,
    county_name text,
 4 area_land bigint,
    area_water bigint,
 5 internal_point_lat numeric(10,7),
    internal_point_lon numeric(10,7),
 6 pop_est_2018 integer,
    pop_est_2019 integer,
    births_2019 integer,
    deaths_2019 integer,
    international_migr_2019 integer,
    domestic_migr_2019 integer,
    residual_2019 integer,
 7 CONSTRAINT counties_2019_key PRIMARY KEY (state_fips,
county_fips)
);
```

Listing 5-2: `CREATE TABLE` *statement for census county population estimates*

Return to the main pgAdmin window, and in the object browser, right-click and refresh the `analysis` database. Choose **Schemas▶public▶Tables** to see the new table. Although it's empty, you can see the structure by running a basic `SELECT` query in pgAdmin's Query Tool:

```
SELECT * FROM us_counties_pop_est_2019;
```

When you run the `SELECT` query, you'll see the columns in the table you created appear in the pgAdmin Data Output pane. No data rows exist yet. We need to import them.

Understanding Census Columns and Data Types

Before we import the CSV file into the table, let's walk through several of the columns and the data types I chose in *Listing 5-2*. As my guide, I used two official census data dictionaries: one for the estimates found at *https://www2.census.gov/programs-surveys/popest/technical-documentation/file-layouts/2010-2019/co-est2019-alldata.pdf* and one for the decennial count that includes the geographic columns at *http://www.census.gov/prod/cen2010/doc/pl94-171.pdf*. I've given some columns more readable names in the table definition. Relying on a data dictionary when possible is good practice, because it helps you avoid misconfiguring columns or potentially losing data. Always ask if one is available, or do an online search if the data is public.

In this set of census data, and thus the table you just made, each row displays the population estimates and components of annual change (births, deaths, and migration) for one county. The first two columns are the county's `state_fips` 1 and `county_fips`, which are the standard federal codes for those entities. We use `text` for both because those codes can contain leading zeros that would be lost if we stored the values as integers. For example, Alaska's `state_fips` is `02`. If we used an integer type, that leading `0` would be stripped on import, leaving `2`, which is the wrong code for the state. Also, we won't be doing any math with this value, so don't need integers. It's always important to distinguish codes from numbers; these state and county values are actually labels as opposed to numbers used for math.

Numbers from 1 to 4 in `region` 2 represent the general location of a county in the United States: the Northeast, Midwest, South, or West. No number is higher than 4, so we define the columns with type `smallint`. The `state_name` 3 and `county_name` columns contain the complete name of both the state and county, stored as `text`.

The number of square meters for land and water in the county are recorded in `area_land` 4 and `area_water`, respectively. The two, combined, comprise a county's total area. In certain places—such as Alaska, where there's lots of land to go with all that snow—some values easily surpass the `integer` type's maximum of 2,147,483,647. For that reason, we're using `bigint`, which will handle the 377,038,836,685 square meters of land in the Yukon-Koyukuk census area with room to spare.

The latitude and longitude of a point near the center of the county, called an *internal point*, are specified in `internal_point_lat` and `internal_point_lon` 5, respectively. The Census Bureau—along with many mapping systems—expresses latitude and longitude coordinates using a *decimal degrees* system. *Latitude* represents positions north and south on the globe, with the equator at 0 degrees, the North Pole at 90 degrees, and the South Pole at −90 degrees.

Longitude represents locations east and west, with the *Prime Meridian* that passes through Greenwich in London at 0 degrees longitude. From there, longitude increases both east and west (positive numbers to the east and negative to the west) until they meet at 180 degrees on the opposite side of the globe. The location there, known as the *antimeridian*, is used as the basis for the *International Date Line*.

When reporting interior points, the Census Bureau uses up to seven decimal places. With a value up to 180 to the left of the decimal, we need to account for a maximum of 10 digits total. So, we're using `numeric` with a precision of `10` and a scale of `7`.

NOTE

PostgreSQL, through the PostGIS extension, can store geometric data, which includes points that represent latitude and longitude in a single column. We'll explore geometric data when we cover geographical queries in Chapter 15.

Next, we reach a series of columns 6 that contain the county's population estimates and components of change. *Table 5-1* lists their definitions.

Table 5-1: *Census Population Estimate Columns*

Column name	Description
`pop_est_2018`	Estimated population on July 1, 2018
`pop_est_2019`	Estimated population on July 1, 2019
`births_2019`	Number of births from July 1, 2018, to June 30, 2019
`deaths_2019`	Number of deaths from July 1, 2018, to June 30, 2019
`international_migr_2019`	Net international migration from July 1, 2018, to June 30, 2019
`domestic_migr_2019`	Net domestic migration from July 1, 2018, to June 30, 2019
`residual_2019`	Number used to adjust estimates for consistency

Finally, the `CREATE TABLE` statement ends with a `CONSTRAINT` clause 7 specifying that the columns `state_fips` and `county_fips` will serve as the table's primary key. This means that the combination of those columns is unique for every row in the table, a concept we'll cover extensively in Chapter 8. For now, let's run the import.

Performing the Census Import with COPY

Now you’re ready to bring the census data into the table. Run the code in Listing 5-3, remembering to change the path to the file to match the location of the data on your computer.

```
COPY us_counties_pop_est_2019
FROM 'C:\YourDirectory\us_counties_pop_est_2019.csv'
WITH (FORMAT CSV, HEADER);
```

Listing 5-3: Importing census data using `COPY`

When the code executes, you should see the following message in pgAdmin:

```
COPY 3142
Query returned successfully in 75 msec.
```

That’s good news: the import CSV has the same number of rows. If you have an issue with the source CSV or your import statement, the database will throw an error. For example, if one of the rows in the CSV had more columns than in the target table, you’d see an error message in the Data Output pane of pgAdmin that provides a hint as to how to fix it:

```
ERROR: extra data after last expected column
Context: COPY us_counties_pop_est_2019, line 2:
"01,001,3,Alabama, ..."
```

Even if no errors are reported, it’s always a good idea to visually scan the data you just imported to ensure everything looks as expected.

Inspecting the Import

Start with a `SELECT` query of all columns and rows:

```
SELECT * FROM us_counties_pop_est_2019;
```

There should be 3,142 rows displayed in pgAdmin, and as you scroll left and right through the result set, each column should have the expected values. Let’s review some columns that we took particular care to define with the appropriate data types. For example, run the following query to show the counties with the largest `area_land` values. We'll use a `LIMIT` clause, which will cause the query to return only the number of rows we want; here, we'll ask for three:

```
SELECT county_name, state_name, area_land
FROM us_counties_pop_est_2019
ORDER BY area_land DESC
LIMIT 3;
```

This query ranks county-level geographies from largest land area to smallest in square meters. We defined `area_land` as `bigint` because the largest values in the field are bigger than the upper range provided by regular `integer`. As you might expect, big Alaskan geographies are at the top:

```
county_name                  state_name    area_land
-------------------------    ----------    ------------
Yukon-Koyukuk Census Area    Alaska        377038836685
North Slope Borough          Alaska        230054247231
Bethel Census Area           Alaska        105232821617
```

Next, let's check the latitude and longitude columns of `internal_point_lat` and `internal_point_lon`, which we defined with `numeric(10,7)`. This code sorts the counties by longitude from the greatest to smallest value. This time, we'll use `LIMIT` to retrieve five rows:

```
SELECT county_name, state_name, internal_point_lat,
internal_point_lon
FROM us_counties_pop_est_2019
ORDER BY internal_point_lon DESC
LIMIT 5;
```

Longitude measures locations from east to west, with locations west of the Prime Meridian in England represented as negative numbers starting with −1, −2, −3, and so on, the farther west you go. We sorted in descending order, so we'd expect the easternmost counties of the United States to show at the top of the query result. Instead—surprise!—there's a lone Alaska geography at the top:

```
      county_name            state_name  internal_point_lat
internal_point_lon
-------------------------- ----------  ------------------  --
----------------
Aleutians West Census Area Alaska              51.9489640
179.6211882
Washington County          Maine               44.9670088
-67.6093542
Hancock County             Maine               44.5649063
-68.3707034
Aroostook County           Maine               46.7091929
-68.6124095
Penobscot County           Maine               45.4092843
-68.6666160
```

Here's why: the Alaskan Aleutian Islands extend so far west (farther west than Hawaii) that they cross the antimeridian at 180 degrees longitude. Once past the antimeridian, longitude turns positive, counting back down to 0. Fortunately, it's not a mistake in the data; however, it's a fact you can tuck away for your next trivia team competition.

Congratulations! You have a legitimate set of government demographic data in your database. I'll use it to demonstrate exporting data with `COPY` later in this chapter, and then you'll use it to learn math functions in Chapter 6. Before we move on to exporting data, let's examine a few additional importing techniques.

Importing a Subset of Columns with COPY

If a CSV file doesn't have data for all the columns in your target database table, you can still import the data you have by specifying which columns are present in the data. Consider this scenario: you're researching the salaries of all town supervisors in your state so you can analyze government spending trends by geography. To get started, you create a table called `supervisor_salaries` with the code in *Listing 5-4*.

```
CREATE TABLE supervisor_salaries (
    id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    town text,
    county text,
    supervisor text,
    start_date date,
    salary numeric(10,2),
    benefits numeric(10,2)
);
```

Listing 5-4: Creating a table to track supervisor salaries

You want columns for the town and county, the supervisor's name, the date they started, and salary and benefits (assuming you just care about current levels). You're also adding an auto-incrementing `id` column as a primary key. However, the first county clerk you contact says, "Sorry, we only have town, supervisor, and salary. You'll need to get the rest from elsewhere." You tell them to send a CSV anyway. You'll import what you can.

I've included such a sample CSV you can download via the book's resources at *https://www.nostarch.com/practical-sql-2nd-edition/*, called *supervisor_salaries.csv*. If you view the file with a text editor, you should see these two lines at the top:

```
town,supervisor,salary
Anytown,Jones,67000
```

You could try to import it using this basic `COPY` syntax:

```
COPY supervisor_salaries
FROM 'C:\YourDirectory\supervisor_salaries.csv'
WITH (FORMAT CSV, HEADER);
```

But if you do, PostgreSQL will return an error:

```
ERROR: invalid input syntax for type integer: "Anytown"
Context: COPY supervisor_salaries, line 2, column id:
"Anytown"
SQL state: 22P04
```

The problem is that your table's first column is the auto-incrementing `id`, but your CSV file begins with the text column `town`. Even if your CSV file had an integer present in its first column, the `GENERATED ALWAYS AS IDENTITY` keywords would prevent you from adding a value to `id`. The workaround for this situation is to tell the database which columns in the table are present in the CSV, as shown in *Listing 5-5*.

```
COPY supervisor_salaries 1 (town, supervisor, salary)
FROM 'C:\YourDirectory\supervisor_salaries.csv'
WITH (FORMAT CSV, HEADER);
```

Listing 5-5: Importing salaries data from CSV to three table columns

By noting in parentheses 1 the three present columns after the table name, we tell PostgreSQL to only look for data to fill those columns when it reads the CSV. Now, if you select the first couple of rows from the table, you'll see those columns filled with the appropriate values:

```
id     town        county     supervisor     start_date
salary        benefits
--   ---------    ------     ----------     ----------     ------
----    --------
1    Anytown                 Jones
67000.00
2    Bumblyburg              Larry
74999.00
```

Importing a Subset of Rows with COPY

Starting with PostgreSQL version 12, you can add a `WHERE` clause to a `COPY` statement to filter which rows from the source CSV you import into a table. You can see how this works using the supervisor salaries data.

Start by clearing all the data you already imported into `supervisor_salaries` using a `DELETE` query.

```
DELETE FROM supervisor_salaries;
```

This will remove data from the table, but it will not reset the `id` column's `IDENTITY` column sequence. We'll cover how to do that when we discuss table design in Chapter 8. When that query finishes, run the `COPY` statement in *Listing 5-6*, which adds a `WHERE` clause that filters the import to include only rows in which the `town` column in the CSV input matches New Brillig.

```
COPY supervisor_salaries (town, supervisor, salary)
FROM 'C:\YourDirectory\supervisor_salaries.csv'
WITH (FORMAT CSV, HEADER)
WHERE town = 'New Brillig';
```

Listing 5-6: Importing a subset of rows with `WHERE`

Next, run `SELECT * FROM supervisor_salaries;` to view the contents of the table. You should see just one row:

```
id    town      county supervisor start_date  salary
benefits
-- ----------- ------ ---------- ---------- --------- -------
-
10 New Brillig        Carroll               102690.00
```

This is a handy shortcut. Now, let's see how to use a temporary table to do even more data wrangling during an import.

Adding a Value to a Column During Import

What if you know that "Mills" is the name that should be added to the `county` column during the import, even though that value is missing from the CSV file? One way to modify your import to include the name is by loading your CSV into a *temporary table* before adding it to `supervisors_salary`. Temporary tables exist only until you end your database session. When you reopen the database (or lose your connection), those tables disappear. They're handy for performing intermediary operations on data as part of your processing pipeline; we'll use one to add the county name to the `supervisor_salaries` table as we import the CSV.

Again, clear the data you've imported into `supervisor_salaries` using a `DELETE` query. When it completes, run the code in Listing 5-7, which will make a temporary table and import your CSV. Then, we will query data from that table and include the county name for an insert into the `supervisor_salaries` table.

```
1 CREATE TEMPORARY TABLE supervisor_salaries_temp
      (LIKE supervisor_salaries INCLUDING ALL);
```

```
2 COPY supervisor_salaries_temp (town, supervisor, salary)
   FROM 'C:\YourDirectory\supervisor_salaries.csv'
   WITH (FORMAT CSV, HEADER);

3 INSERT INTO supervisor_salaries (town, county, supervisor,
   salary)
   SELECT town, 'Mills', supervisor, salary
   FROM supervisor_salaries_temp;

4 DROP TABLE supervisor_salaries_temp;
```

Listing 5-7: Using a temporary table to add a default value to a column during import

This script performs four tasks. First, we create a temporary table called `supervisor_salaries_temp` 1 based on the original `supervisor_salaries` table by passing as an argument the `LIKE` keyword followed by the source table name. The keywords `INCLUDING ALL` tell PostgreSQL to not only copy the table rows and columns but also components such as indexes and the `IDENTITY` settings. Then we import the *supervisor_salaries.csv* file 2 into the temporary table using the now-familiar `COPY` syntax.

Next, we use an `INSERT` statement 3 to fill the salaries table. Instead of specifying values, we employ a `SELECT` statement to query the temporary table. That query specifies `Mills` as the value for the second column, not as a column name, but as a string inside single quotes.

Finally, we use `DROP TABLE` to erase the temporary table 4 since we're done using it for this import. The temporary table will automatically disappear when you disconnect from the PostgreSQL session, but this removes it now in case we want to do another import and use a fresh temporary table for another CSV.

After you run the query, run a `SELECT` statement on the first couple of rows to see the effect:

```
id     town       county      supervisor   start_date
salary    benefits
--   --------    ---------   ----------   ----------   ------
---  --------
```

```
11   Anytown     Mills       Jones
67000.00
12   Bumblyburg  Mills       Larry
74999.00
```

You've filled the `county` field with a value even though your source CSV didn't have one. The path to this import might seem laborious, but it's instructive to see how data processing can require multiple steps to get the desired results. The good news is that this temporary table demo is an apt indicator of the flexibility SQL offers to control data handling.

Using COPY to Export Data

When exporting data with `COPY`, rather than using `FROM` to identify the source data, you use `TO` for the path and name of the output file. You control how much data to export—an entire table, just a few columns, or the results of a query.

Let's look at three quick examples.

Exporting All Data

The simplest export sends everything in a table to a file. Earlier, you created the table `us_counties_pop_est_2019` with 16 columns and 3,142 rows of census data. The SQL statement in *Listing 5-8* exports all the data to a text file named *us_counties_export.txt*. To demonstrate the flexibility you have in choosing output options, the `WITH` keyword tells PostgreSQL to include a header row and use the pipe symbol instead of a comma for a delimiter. I've used the *.txt* file extension here for two reasons. First, it demonstrates that you can name your file with an extension other than *.csv*; second, we're using a pipe for a delimiter, not a comma, so I want to avoid calling the file *.csv* unless it truly has commas as a separator.

Remember to change the output directory to your preferred save location.

```
COPY us_counties_pop_est_2019
TO 'C:\YourDirectory\us_counties_export.txt'
WITH (FORMAT CSV, HEADER, DELIMITER '|');
```

Listing 5-8: Exporting an entire table with `COPY`

View the export file with a text editor to see the data in this format (I've truncated the results):

```
state_fips|county_fips|region|state_name|county_name| --snip-
-
01|001|3|Alabama|Autauga County --snip--
```

The file includes a header row with column names, and all columns are separated by the pipe delimiter.

Exporting Particular Columns

You don't always need (or want) to export all your data: you might have sensitive information, such as Social Security numbers or birthdates, that need to remain private. Or, in the case of the census county data, maybe you're working with a mapping program and only need the county name and its geographic coordinates to plot the locations. We can export only these three columns by listing them in parentheses after the table name, as shown in Listing 5-9. Of course, you must enter these column names precisely as they're listed in the data for PostgreSQL to recognize them.

```
COPY us_counties_pop_est_2019
    (county_name, internal_point_lat, internal_point_lon)
TO 'C:\YourDirectory\us_counties_latlon_export.txt'
WITH (FORMAT CSV, HEADER, DELIMITER '|');
```

Listing 5-9: Exporting selected columns from a table with `COPY`

Exporting Query Results

Additionally, you can add a query to `COPY` to fine-tune your output. In Listing 5-10 we export the name and state of only those counties whose names contain the letters `mill`, catching it in either uppercase or lowercase by using the case-insensitive `ILIKE` and the `%` wildcard character we covered in "Using LIKE and ILIKE with WHERE" in Chapter 3. Also note that for this example, I've removed the `DELIMITER` keyword from the `WITH` clause. As a result, the output will default to comma-separated values.

```
COPY (
    SELECT county_name, state_name
    FROM us_counties_pop_est_2019
    WHERE county_name ILIKE '%mill%'
     )
TO 'C:\YourDirectory\us_counties_mill_export.csv'
WITH (FORMAT CSV, HEADER);
```

Listing 5-10: Exporting query results with `COPY`

After running the code, your output file should have nine rows with county names including Miller, Roger Mills, and Vermillion:

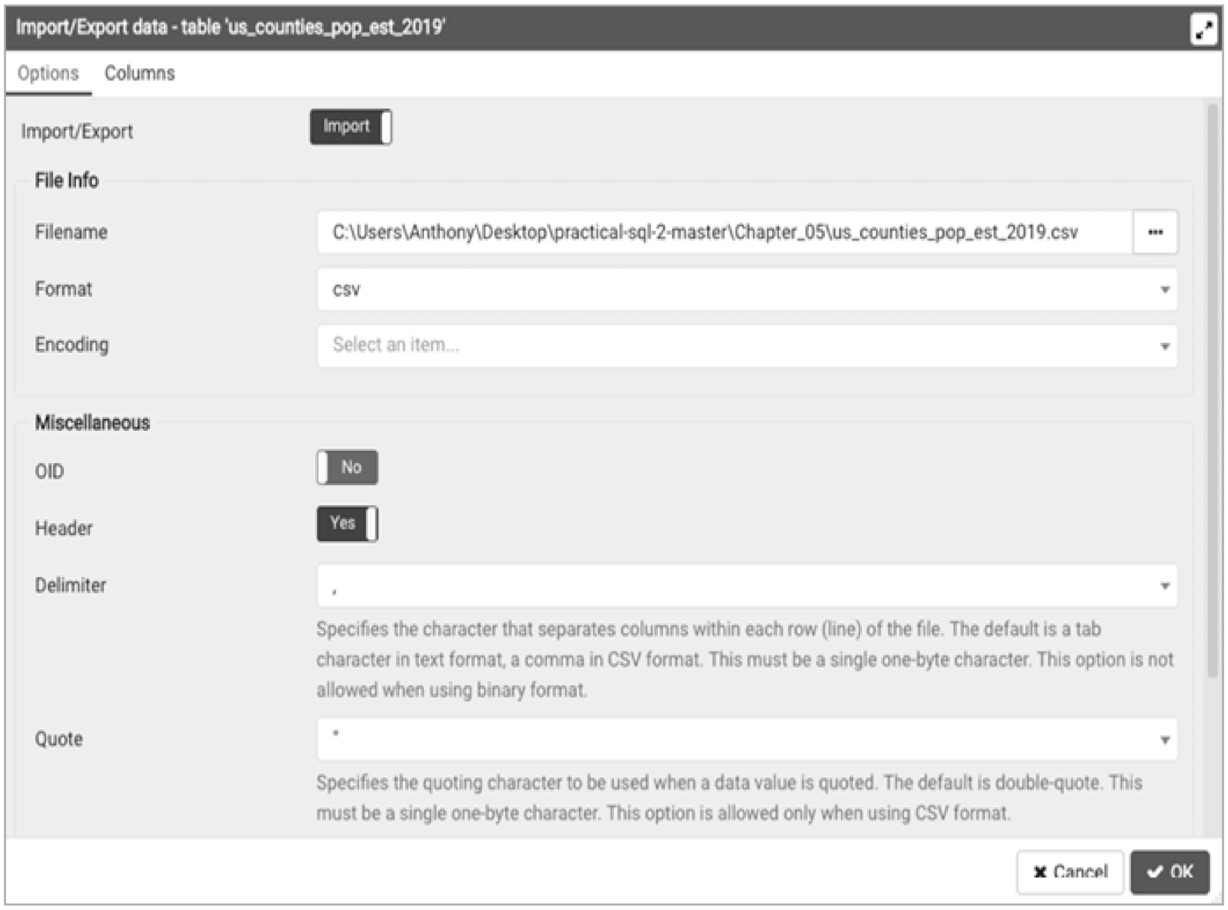
```
county_name,state_name
Miller County,Arkansas
Miller County,Georgia
Vermillion County,Indiana
--snip--
```

Importing and Exporting Through pgAdmin

At times, the SQL `COPY` command won't be able to handle certain imports and exports. This typically happens when you're connected to a PostgreSQL instance running on a computer other than yours. A machine in a cloud computing environment such as Amazon Web Services is a good example. In that scenario, PostgreSQL's `COPY` command will look for files and file paths that exist on that remote machine; it can't find files on your local computer. To use `COPY`, you'd need to transfer your data to the remote server, but you might not always have the rights to do that.

One workaround is to use pgAdmin's built-in import/export wizard. In pgAdmin's object browser (the left vertical pane), locate the list of tables in your `analysis` database by choosing **Databases▸analysis▸Schemas▸public▸Tables**.

Next, right-click the table you want to import to or export from, and select **Import/Export**. A dialog appears that lets you choose to either import or export from that table, as shown in Figure 5-1.



Import/Export data - table 'us_counties_pop_est_2019'

Import/Export: Import

File Info

Filename: C:\Users\Anthony\Desktop\practical-sql-2-master\Chapter_05\us_counties_pop_est_2019.csv

Format: csv

Encoding: Select an item...

Miscellaneous

OID: No

Header: Yes

Delimiter: ,

Specifies the character that separates columns within each row (line) of the file. The default is a tab character in text format, a comma in CSV format. This must be a single one-byte character. This option is not allowed when using binary format.

Quote: "

Specifies the quoting character to be used when a data value is quoted. The default is double-quote. This must be a single one-byte character. This option is allowed only when using CSV format.

Figure 5-1: The pgAdmin Import/Export dialog

To import, move the Import/Export slider to **Import**. Then click the three dots to the right of the **Filename** box to locate your CSV file. From the Format drop-down list, choose **csv**. Then adjust the header, delimiter, quoting, and other options as needed. Click **OK** to import the data.

To export, use the same dialog and follow similar steps.

In Chapter 18, when we discuss using PostgreSQL from your computer's command line, we'll explore another way to accomplish this using a utility called `psql` and its `\copy` command. pgAdmin's import/export wizard actually uses `\copy` in the background but gives it a friendlier face.

Wrapping Up

Now that you've learned how to bring external data into your database, you can start digging into a myriad of datasets, whether you want to explore one of the thousands of publicly available datasets, or data related to your own career or studies. Plenty of data is available in CSV format or a format easily convertible to CSV. Look for data dictionaries to help you understand the data and choose the right data type for each field.

The census data you imported as part of this chapter's exercises will play a starring role in the next chapter, in which we explore math functions with SQL.

TRY IT YOURSELF

Continue your exploration of data import and export with these exercises. Remember to consult the PostgreSQL documentation at *https://www.postgresql.org/docs/current/sql-copy.html* for hints:

Write a `WITH` statement to include with `COPY` to handle the import of an imaginary text file whose first couple of rows look like this:

```
id:movie:actor
50:#Mission: Impossible#:Tom Cruise
```

Using the table `us_counties_pop_est_2019` you created and filled in this chapter, export to a CSV file the 20 counties in the United States that had the most births. Make sure you export only each county's name, state, and number of births. (Hint: births are totaled for each county in the column `births_2019`.)

Imagine you're importing a file that contains a column with these values:

```
17519.668
20084.461
18976.335
```

Will a column in your target table with data type `numeric(3,8)` work for these values? Why or why not?

6
BASIC MATH AND STATS WITH SQL

If your data includes any of the number data types we explored in Chapter 4—integers, decimals, or floating points—sooner or later your analysis will include some calculations. You might want to know the average of all the dollar values in a column or add values in two columns to produce a total for each row. SQL can handle those calculations and more, from basic math through advanced statistics.

In this chapter, I'll start with the basics and progress to math functions and beginning statistics. I'll also discuss calculations related to percentages and percent change. For several of the exercises, we'll use the 2019 US Census population estimates data you imported in Chapter 5.

Understanding Math Operators and Functions

Let's start with the basic math you learned in grade school (all's forgiven if you've forgotten some of it). Table 6-1 shows nine math operators you'll use most often in your calculations. The first four (addition, subtraction, multiplication, and division) are part of the ANSI SQL standard and are implemented in all database systems. The others are PostgreSQL-specific operators, although most other database managers likely have functions or operators to perform those operations too. For example, the modulo operator (`%`) works in Microsoft SQL Server and MySQL as well as with PostgreSQL. If you're using another database system, check its documentation.

Table 6-1: *Basic Math Operators*

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division (returns the quotient only, no remainder)
`%`	Modulo (returns just the remainder)
`^`	Exponentiation
`|/`	Square root
`||/`	Cube root
`!`	Factorial

We'll step through each of these operators by executing simple SQL queries on plain numbers rather than operating on a table or another database object. You can either enter the statements separately into the pgAdmin query tool and execute them one at a time, or if you copied the code for this chapter from the resources at *https://www.nostarch.com/practical-sql-2nd-edition/*, you can highlight each line and execute it.

Understanding Math and Data Types

As you work through the examples, note the data type of each result, which is listed beneath each column name in the pgAdmin results grid. The type returned for a calculation will vary depending on the operation and the data type of the input numbers. When using an operator between two numbers—addition, subtraction, multiplication, or division—the data type returned follows this pattern:

Two integers return an `integer`.

A `numeric` on either side or both sides of the operator returns a `numeric`.

Anything with a floating-point number returns a floating-point number of type `double precision`.

However, the exponentiation, root, and factorial functions are different. Each takes just one number, either before or after the operator, and returns numeric and floating-point types, even when the input is an integer.

Sometimes the result's data type will suit your needs; other times, you may need to use `CAST` to change the data type, as mentioned in "Transforming Values from One Type to Another with CAST" in Chapter 4, such as if you need to feed the result into a function that takes a certain type. I'll note those times as we work through the book.

NOTE

PostgreSQL defines the arguments that operators accept, the internal functions they call, and the data types they return in a table called `pg_operator`. For example, the `+` operator is defined once for accepting integers, again for accepting numerics, and so on.

Adding, Subtracting, and Multiplying

Let's start with simple integer addition, subtraction, and multiplication. Listing 6-1 shows three examples, each with the `SELECT` keyword followed by the math formula. Since Chapter 3, we've used `SELECT` for its main purpose: to retrieve data from a table. But with PostgreSQL, Microsoft's SQL Server, MySQL, and some other database management systems, you can omit the table name and perform simple math and string operations, as we do here. For readability's sake, I recommend you use a single space before and after the math operator; although using spaces isn't strictly necessary for your code to work, it is good practice.

```
1 SELECT 2 + 2;
2 SELECT 9 - 1;
3 SELECT 3 * 4;
```

Listing 6-1: Basic addition, subtraction, and multiplication with SQL

None of these statements is rocket science, so you shouldn't be surprised that running `SELECT 2 + 2;` 1 in the Query Tool shows a result of `4`. Similarly, the examples for subtraction 2 and multiplication 3 yield what you'd expect: `8` and `12`. The output displays in a column, as with any query result. But because we're not querying a table and specifying a column, the results appear beneath a `?column?` name, signifying an unknown column:

```
?column?
--------
       4
```

That's okay. We're not affecting any data in a table, just displaying a result. If you want to display a column name, you can provide an alias, as in `SELECT 3 * 4 AS result;`.

Performing Division and Modulo

Division with SQL gets a little trickier because of the difference between math with integers and math with decimals. Add in *modulo*, an operator that returns just the *remainder* in a division operation, and the results can be confusing. So, to make it clear, Listing 6-2 shows four examples.

```
1 SELECT 11 / 6;
2 SELECT 11 % 6;
3 SELECT 11.0 / 6;
4 SELECT CAST(11 AS numeric(3,1)) / 6;
```

Listing 6-2: Integer and decimal division with SQL

The `/` operator 1 divides the integer `11` by another integer, `6`. If you do that math in your head, you know the answer is `1` with a remainder of `5`. However, running this query yields `1`, which is how SQL handles division of one integer by another—by reporting only the integer *quotient* without any remainder. If you want to retrieve the *remainder* as an integer, you must perform the same calculation using the modulo operator `%`, as in 2. That statement returns just the remainder, in this case `5`. No single operation today will provide you with both the quotient and the remainder as integers, though an enterprising developer could add that functionality in the future.

Modulo is useful for more than just fetching a remainder: you can also use it as a test condition. For example, to check whether a number is even, you can test it using the `% 2` operation. If the result is `0` with no remainder, the number is even.

There are two ways to divide two numbers and have the result return as a `numeric` type. First, if one or both of the numbers is a `numeric`, the result will by default be expressed as a `numeric`. That's what happens when I divide `11.0` by `6` 3. Execute that query, and the result is `1.83333`. The number of decimal digits displayed may vary according to your PostgreSQL and system settings.

Second, if you're working with data stored only as integers and need to force decimal division, you can use `CAST` to convert one of the integers to a `numeric` type 4. Executing this also returns `1.83333`.

Using Exponents, Roots, and Factorials

Beyond the basics, PostgreSQL-flavored SQL also provides operators and functions to square, cube, or otherwise raise a base number to an exponent, as well as find roots or the factorial of a number. Listing 6-3 shows these operations in action.

```
1 SELECT 3 ^ 4;
2 SELECT |/ 10;
   SELECT sqrt(10);
3 SELECT ||/ 10;
4 SELECT factorial(4);
   SELECT 4 !;
```

Listing 6-3: Exponents, roots, and factorials with SQL

The exponentiation operator (`^`) allows you to raise a given base number to an exponent, as in 1, where `3 ^ 4` (colloquially, we'd call that three to the fourth power) returns `81`.

You can find the square root of a number in two ways: using the `|/` operator 2 or the `sqrt(n)` function. For a cube root, use the `||/` operator 3. Both are *prefix operators*, named because they come before a single value.

To find the *factorial* of a number, you can use the `factorial(n)` function or the `!` operator. The `!`, available only in PostgreSQL versions 13 and earlier, is a *suffix operator*, coming after a single value. You'll use factorials in many places in math, but perhaps the most common is to determine how many ways a number of items can be ordered. Say you have four photographs. How many ways could you order them on a wall? To find the answer, you'd calculate the factorial by starting with the number of items and multiplying it by all the smaller positive integers. So, at 4, the function `factorial(4)` is equivalent to 4 × 3 × 2 × 1. That's 24 ways to order four photos. No wonder decorating takes so long sometimes!

Again, these operators are specific to PostgreSQL; they're not part of the SQL standard. If you're using another database application, check its documentation for how it implements these operations.

Minding the Order of Operations

You may recall from early math lessons what the order of operations, or *operator precedence*, is on a mathematical expression. Which calculations does SQL execute first? Not surprisingly, SQL follows the established math standard. For the PostgreSQL operators discussed so far, the order is as follows:

- Exponents and roots
- Multiplication, division, modulo
- Addition and subtraction

Given these rules, you'll need to encase an operation in parentheses if you want to calculate it in a different order. For example, the following two expressions yield different results:

```
SELECT 7 + 8 * 9;
SELECT (7 + 8) * 9;
```

The first expression returns `79` because the multiplication operation receives precedence and is processed before the addition. The second returns `135` because the parentheses force the addition operation to occur first.

Here’s a second example using exponents:

```
SELECT 3 ^ 3 - 1;
SELECT 3 ^ (3 - 1);
```

Exponent operations take precedence over subtraction, so without parentheses the entire expression is evaluated left to right and the operation to find 3 to the power of 3 happens first. Then 1 is subtracted, returning `26`. In the second example, the parentheses force the subtraction to happen first, so the operation results in `9`, which is 3 to the power of 2.

Keep operator precedence in mind to avoid having to correct your analysis later!

Doing Math Across Census Table Columns

Let’s try to use the most frequently used SQL math operators on real data by digging into the 2019 US Census population estimates table, `us_counties_pop_est_2019`, that you imported in Chapter 5. Instead of using numbers in queries, we’ll use the names of the columns that contain the numbers. When we execute the query, the calculation will occur on each row of the table.

To refresh your memory about the data, run the script in *Listing 6-4*. It should return 3,142 rows showing the name and state of each county in the United States plus the 2019 components of population change: births, deaths, and international and domestic migration.

```
SELECT county_name AS1 county,
       state_name AS state,
       pop_est_2019 AS pop,
       births_2019 AS births,
       deaths_2019 AS deaths,
       international_migr_2019 AS int_migr,
       domestic_migr_2019 AS dom_migr,
       residual_2019 AS residual
FROM us_counties_pop_est_2019;
```

Listing 6-4: Selecting census population estimate columns with aliases

This query doesn't return all columns in the table, just the ones with data related to the population estimates. In addition, I employ the `AS` keyword 1 to give each column a shorter *alias* in the result set. Because all the data in this query is from 2019, I'm eliminating the year from the names of the results columns to reduce scrolling in the pgAdmin output. It's an arbitrary decision that you can adjust.

Adding and Subtracting Columns

Now, let's try a simple calculation using two of the columns. Listing 6-5 subtracts the number of deaths from the number of births in each county, a measure the census refers to as natural increase. Let's see what this shows.

```
SELECT county_name AS county,
       state_name AS state,
       births_2019 AS births,
       deaths_2019 AS deaths,
    1 births_2019 - deaths_2019 AS natural_increase
FROM us_counties_pop_est_2019
ORDER BY state_name, county_name;
```

Listing 6-5: Subtracting two columns in us_counties_pop_est_2019

Providing `births_2019 - deaths_2019` 1 as one of the columns in the `SELECT` statement handles the calculation. Again, I use the `AS` keyword to provide a readable alias for the column. If you don't provide an alias, PostgreSQL uses the label `?column?`, which is far less than helpful.

Run the query to see the results. The first few rows should resemble this output:

```
county                 state      births     deaths
natural_increase
--------------         -------    ------     ------     ---------------
-
Autauga County         Alabama      624        541                   83
```

```
Baldwin County   Alabama    2304       2326                  -22
Barbour County   Alabama     256        312                  -56
Bibb County      Alabama     240        252                  -12
```

A quick check with a calculator or pencil and paper confirms that the `natural_increase` column equals the difference between the two columns you subtracted. Excellent! Notice as you scroll through the output that some counties have more births than deaths, while others have the opposite. Typically, counties with a younger mix of residents see births outpace deaths; those with an older set of people—think rural areas and retirement hotspots—tend to see a greater number of deaths than births.

Now, let's build on this to test our data and validate that we imported columns correctly. The population estimate for 2019 should equal the sum of the 2018 estimate and the columns about births, deaths, migration, and residual factor. The code in Listing 6-6 should show that it does.

```
SELECT county_name AS county,
       state_name AS state,
    1 pop_est_2019 AS pop,
    2 pop_est_2018 + births_2019 - deaths_2019 +
           international_migr_2019 + domestic_migr_2019 +
           residual_2019 AS components_total,
    3 pop_est_2019 - (pop_est_2018 + births_2019 -
deaths_2019 +
           international_migr_2019 + domestic_migr_2019 +
           residual_2019) AS difference
FROM us_counties_pop_est_2019
4 ORDER BY difference DESC;
```

Listing 6-6: Checking census data totals

This query includes the 2019 population estimate 1, followed by a calculation adding the components to the 2018 population estimate as `component_total` 2. The 2018 estimate plus the components should equal the 2019 estimate. Rather than manually check, we also add a column that subtracts the components total from the 2019 estimate 3. That column, named `difference`, should contain a zero in each row if all the data is in the right place. To avoid having to scan all 3,142 rows, we add an `ORDER BY` clause 4 on the named column. Any rows showing a difference should appear at the top or bottom of the query result.

Run the query; the first few rows should provide this result:

```
    county        state     pop    components_total  difference
--------------  -------  ------  ----------------  ----------
Autauga County  Alabama   55869             55869           0
Baldwin County  Alabama  223234            223234           0
Barbour County  Alabama   24686             24686           0
```

With the `difference` column showing zeros, we can be confident that our import was clean. Whenever I encounter or import a new dataset, I like to perform little tests like this. They help me better understand the data and head off any potential issues before I dig into analysis.

Finding Percentages of the Whole

One way to spot differences in the items in a dataset is to calculate the percentage of the whole that a particular data point represents. Then, you can glean meaningful insights—and sometimes surprises—by comparing that percentage across all the items in your dataset.

To figure out the percentage of the whole, divide the number in question by the total. For example, if you had a basket of 12 apples and used 9 in a pie, that would be 9 / 12 or 0.75—commonly expressed as 75 percent.

We'll try this on the census population estimates using the two columns that represent the size of each county's geographical features. The columns `area_land` and `area_water` show a county's land and water measurement in square meters. Using the code in Listing 6-7, we can calculate for each county the percentage of its area that is made up of water.

```
SELECT county_name AS county,
       state_name AS state,
    1 area_water::numeric / (area_land + area_water) * 100 AS
pct_water
FROM us_counties_pop_est_2019
ORDER BY pct_water DESC;
```

Listing 6-7: Calculating the percent of a county's area that is water

The key piece of this query divides `area_water` by the sum of `area_land` and `area_water`, which together represent the total area of the county 1.

If we use the data as their original integer types, we won't get the fractional result we need: every row will display a result of 0, the quotient. Instead, we force decimal division by casting one of the integers to the numeric type. Here, for brevity, we use the PostgreSQL-specific double-colon notation after the first reference to `area_water`, but you can also use the ANSI SQL standard `CAST` function covered in Chapter 4. Finally, we multiply the result by 100 to present the result as a fraction of 100—the way most people understand percentages.

By sorting from highest to lowest percentage, the top of the output is as follows:

```
      county            state            pct_water
------------------  -------------  -----------------------
Keweenaw County     Michigan       90.94723747453215452900
Leelanau County     Michigan       86.28858968116583102500
Nantucket County    Massachusetts  84.79692499185512352300
St. Bernard Parish  Louisiana      82.48371149202893908400
Alger County        Michigan       81.87221940647501072300
```

If you check the Wikipedia entry for Keweenaw County, you'll discover the reason why its total area is more than 90 percent water: its land area includes an island in Lake Superior, and the lake's waters are included in the total reported by the census. Add that to your trivia collection!

Tracking Percent Change

Another key indicator in data analysis is percent change: how much bigger, or smaller, is one number than another? Percent change calculations are often employed when analyzing change over time, and they're particularly useful for comparing change among similar items.

Some examples include the following:

The year-over-year change in the number of vehicles sold by each automobile maker

The monthly change in subscriptions to each email list owned by a marketing firm

The annual increase or decrease in enrollment at schools across a nation

The formula to calculate percent change can be expressed like this:

(*new number* – *old number*) / *old number*

So, if you own a lemonade stand and sold 73 glasses of lemonade today and 59 glasses yesterday, you'd figure the day-to-day percent change like this:

(73 – 59) / 59 = .237 = 23.7%

Let's try this with a small collection of test data related to spending in departments of a hypothetical local government. *Listing 6-8* calculates which departments had the greatest percentage increase and decrease.

```
1 CREATE TABLE percent_change (
      department text,
      spend_2019 numeric(10,2),
      spend_2022 numeric(10,2)
  );

2 INSERT INTO percent_change
  VALUES
      ('Assessor', 178556, 179500),
      ('Building', 250000, 289000),
      ('Clerk', 451980, 650000),
      ('Library', 87777, 90001),
      ('Parks', 250000, 223000),
      ('Water', 199000, 195000);

  SELECT department,
         spend_2019,
         spend_2022,
       3 round( (spend_2022 - spend_2019) /
                     spend_2019 * 100, 1) AS pct_change
  FROM percent_change;
```

Listing 6-8: Calculating percent change

We create a small table called `percent_change` 1 and insert six rows 2 with data on department spending for the years 2019 and 2022. The percent change formula 3 subtracts `spend_2019` from `spend_2022` and then divides by `spend_2019`. We multiply by 100 to express the result as a portion of 100.

To simplify the output, this time I've added the `round()` function to remove all but one decimal place. The function takes two arguments: the column or expression to be rounded and the number of decimal places to display. Because both numbers are type `numeric`, the result will also be a `numeric`.

The script creates this result:

```
department   spend_2019   spend_2022   pct_change
----------   ----------   ----------   ----------
Assessor      178556.00    179500.00          0.5
Building      250000.00    289000.00         15.6
Clerk         451980.00    650000.00         43.8
Library        87777.00     90001.00          2.5
Parks         250000.00    223000.00        -10.8
Water         199000.00    195000.00         -2.0
```

Now, it's just a matter of finding out why the Clerk department's spending has outpaced others in the town.

Using Aggregate Functions for Averages and Sums

So far, we've performed math operations across columns in each row of a table. SQL also lets you calculate a result from values within the same column using *aggregate functions*. You can see a full list of PostgreSQL aggregates, which calculate a single result from multiple inputs, at *https://www.postgresql.org/docs/current/functions-aggregate.html*. Two of the most-used aggregate functions in data analysis are `avg()` and `sum()`.

Returning to the `us_counties_pop_est_2019` census table, it's reasonable to want to calculate the total population of all counties plus the average population of all counties. Using `avg()` and `sum()` on column `pop_est_2019` (the population estimate for 2019) makes it easy, as shown in

Listing 6-9. Again, we use the `round()` function to remove numbers after the decimal point in the average calculation.

```
SELECT sum(pop_est_2019) AS county_sum,
       round(avg(pop_est_2019), 0) AS county_average
FROM us_counties_pop_est_2019;
```

Listing 6-9: Using the `sum()` and `avg()` aggregate functions

This calculation produces the following result:

```
county_sum  county_average
----------  --------------
 328239523          104468
```

The estimated population for all counties in the United States in 2019 added up to approximately 328.2 million, and the average of the county population estimates was 104,468.

Finding the Median

The *median* value in a set of numbers is as important an indicator, if not more so, than the average. Here's the difference between median and average:

Average The sum of all the values divided by the number of values

Median The "middle" value in an ordered set of values

Median is important in data analysis because it reduces the effect of outliers. Consider this example: let's say six kids, ages 10, 11, 10, 9, 13, and 12, go on a field trip. It's easy to add the ages and divide by six to get the group's average age:

$$(10 + 11 + 10 + 9 + 13 + 12) / 6 = 10.8$$

Because the ages fall within a narrow range, the 10.8 average is a good representation of the group. But averages are less helpful when the values are bunched, or skewed, toward one end of the distribution, or if the group includes outliers.

For example, say an older chaperone joins the field trip. With ages of 10, 11, 10, 9, 13, 12, and 46, the average age increases considerably:

(10 + 11 + 10 + 9 + 13 + 12 + 46) / 7 = 15.9

Now the average doesn't represent the group well because the outlier skews it, making it an unreliable indicator.

It's better in this case to find the median, the midpoint in an ordered list of values—the point at which half the values are more and half are less. Using the field trip, we order the attendees' ages from lowest to highest:

9, 10, 10, 11, 12, 13, 46

The middle (median) value is 11. Given this group, the median of 11 is a better picture of the typical age than the average of 15.9.

If the set of values is an even number, you take the average of the two middle numbers to find the median. Let's add another student (age 12) to the field trip:

9, 10, 10, 11, 12, 12, 13, 46

Now, the two middle values are 11 and 12. To find the median, we average them: 11.5.

Medians are reported frequently in financial news. Reports on housing prices often use medians because a few sales of McMansions in a ZIP code that is otherwise modest can make averages useless. The same goes for sports player salaries: one or two superstars can skew a team's average.

A good test is to calculate the average and the median for a group of values. If they're close, the group is probably normally distributed (the familiar bell curve), and the average is useful. If they're far apart, the values are not normally distributed, and the median is the better representation.

Finding the Median with Percentile Functions

PostgreSQL (as with most relational databases) does not have a built-in `median()` function like you'd find in Excel or other spreadsheet programs.

It's also not included in the ANSI SQL standard. Instead we can use a SQL *percentile* function to find the median and use *quantiles* or *cut points* to divide a group of numbers into equal sizes. Percentile functions are part of standard ANSI SQL.

In statistics, percentiles indicate the point in an ordered set of data below which a certain percentage of the data is found. For example, a doctor might tell you that your height places you in the 60th percentile for an adult in your age group. That means 60 percent of people are shorter than you.

The median is equivalent to the 50th percentile—again, half the values are below and half above. There are two versions of the percentile function —`percentile_cont(n)` and `percentile_disc(n)`. Both functions are part of the ANSI SQL standard and are present in PostgreSQL, Microsoft SQL Server, and other databases.

The `percentile_cont(n)` function calculates percentiles as *continuous* values. That is, the result does not have to be one of the numbers in the dataset but can be a decimal value in between two of the numbers. This follows the methodology for calculating medians on an even number of values, where the median is the average of the two middle numbers. The `percentile_disc(n)` function returns only *discrete* values, meaning the result will be rounded to one of the numbers in the set.

In Listing 6-10 we make a test table with six numbers and find the percentiles.

```
CREATE TABLE percentile_test (
    numbers integer
);

INSERT INTO percentile_test (numbers) VALUES
    (1), (2), (3), (4), (5), (6);

SELECT
  1 percentile_cont(.5)
    WITHIN GROUP (ORDER BY numbers),
  2 percentile_disc(.5)
    WITHIN GROUP (ORDER BY numbers)
FROM percentile_test;
```

Listing 6-10: Testing SQL percentile functions

In both the continuous 1 and discrete 2 percentile functions, we enter `.5` to represent the 50th percentile, equivalent to the median. Running the code returns the following:

```
percentile_cont    percentile_disc
---------------    ---------------
            3.5                  3
```

The `percentile_cont()` function returned what we'd expect the median to be: `3.5`. But because `percentile_disc()` calculates discrete values, it reports `3`, the last value in the first 50 percent of the numbers. Because the accepted method of calculating medians is to average the two middle values in an even-numbered set, use `percentile_cont(.5)` to find a median.

Finding Median and Percentiles with Census Data

Our census data can show how a median tells a different story than an average. Listing 6-11 adds `percentile_cont()` alongside the `sum()` and `avg()` aggregates we've used so far to find the sum, average, and median population of all counties.

```
SELECT sum(pop_est_2019) AS county_sum,
       round(avg(pop_est_2019), 0) AS county_average,
       percentile_cont(.5)
       WITHIN GROUP (ORDER BY pop_est_2019) AS county_median
FROM us_counties_pop_est_2019;
```

Listing 6-11: Using `sum()`, `avg()`, and `percentile_cont()` aggregate functions

Your result should equal the following:

```
county_sum  county_avg  county_median
----------  ----------  -------------
 328239523      104468          25726
```

The median and average are far apart, which shows that averages can mislead. As of 2019 estimates, half the counties in America had fewer than 25,726 people, whereas half had more. If you gave a presentation on US demographics and told the audience that the "average county in America has 104,468 people," they'd walk away with a skewed picture of reality. More than 40 counties were estimated to have a million or more people in 2019, and Los Angeles County had more than 10 million. That pushed the average higher.

Finding Other Quantiles with Percentile Functions

You can also slice data into smaller equal groups for analysis. Most common are *quartiles* (four equal groups), *quintiles* (five groups), and *deciles* (10 groups). To find any individual value, you can just plug it into a percentile function. To find the value marking the first quartile or the lowest 25 percent of data, you'd use a value of `.25`:

```
percentile_cont(.25)
```

However, entering values one at a time is laborious if you want to generate multiple cut points. Instead, you can pass values into `percentile_cont()` using an *array*, a list of items.

Listing 6-12 shows how to calculate all four quartiles at once.

```
SELECT percentile_cont(1ARRAY[.25,.5,.75])
       WITHIN GROUP (ORDER BY pop_est_2019) AS quartiles
FROM us_counties_pop_est_2019;
```

Listing 6-12: *Passing an array of values to* `percentile_cont()`

In this example, we create our cut points by enclosing values in an *array constructor* 1 called `ARRAY[]`. An array constructor is an expression that builds an array from the elements included between the square brackets. Inside the brackets, we provide comma-separated values representing the three points at which to cut to create four quartiles. Run the query, and you should see this output:

```
quartiles
------------------------
{10902.5,25726,68072.75}
```

Because we passed in an array, PostgreSQL returns an array, denoted in the results by curly brackets. Each quartile is separated by commas. The first quartile is 10,902.5, which means 25 percent of counties have a population that is equal to or lower than this value. The second quartile is the same as the median: 25,726. The third quartile is 68,072.75, meaning the largest 25 percent of counties have at least this large of a population. (When reporting these, we'd of course round up or down, as we don't deal in fractions when talking about people.)

Arrays are defined in the ANSI SQL standard, and our use here is just one of several ways you work with arrays in PostgreSQL. You can, for example, define a table column as an array of a particular data type. That's useful if you want store multiple values in a single database column, such as a collection of tags for a blog post, instead of storing them in a separate table. See the PostgreSQL documentation at *https://www.postgresql.org/docs/current/arrays.html* for examples of declaring, searching, and modifying arrays.

Arrays also come with a host of functions (noted for PostgreSQL at *https://www.postgresql.org/docs/current/functions-array.html*) that allow you to perform tasks such as adding or removing values or counting the elements. A handy function for working with the result returned in *Listing 6-12* is `unnest()`, which makes the array easier to read by turning it into rows. *Listing 6-13* shows the code.

```
SELECT unnest(
            percentile_cont(ARRAY[.25,.5,.75])
            WITHIN GROUP (ORDER BY pop_est_2019)
            ) AS quartiles
FROM us_counties_pop_est_2019;
```

Listing 6-13: Using `unnest()` to turn an array into rows

Now the output should be in rows:

```
quartiles
---------
  10902.5
    25726
 68072.75
```

If we were computing deciles, pulling them from the resulting array and displaying them in rows would be especially helpful.

Finding the Mode

We can find the *mode*, the value that appears most often, using the PostgreSQL `mode()` function. The function is not part of standard SQL and has a syntax similar to the percentile functions. *Listing 6-14* shows a `mode()` calculation on `births_2019`, the column showing the number of babies born.

```
SELECT mode() WITHIN GROUP (ORDER BY births_2019)
FROM us_counties_pop_est_2019;
```

Listing 6-14: Finding the most frequent value with `mode()`

The result is `86`, a number of births shared by 16 counties.

Wrapping Up

Working with numbers is a key step in acquiring meaning from your data, and with the math skills covered in this chapter, you're ready to handle the foundations of numerical analysis with SQL. Later in the book, you'll learn about deeper statistical concepts including regression and correlation, but at this point you've mastered the basics of sums, averages, and percentiles. You've also learned how a median can be a fairer assessment of a group of values than an average. That alone can help you avoid inaccurate conclusions.

In the next chapter, I'll introduce you to the power of joining data in two or more tables to increase your options for data analysis. We'll use the 2019 US Census data you've already loaded into the `analysis` database and explore additional datasets.

TRY IT YOURSELF

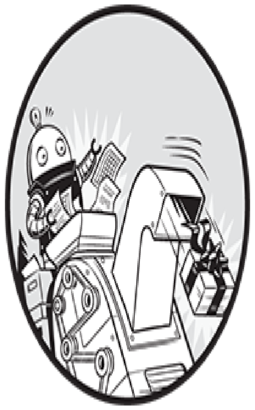
Here are three exercises to test your SQL math skills:

Write a SQL statement for calculating the area of a circle whose radius is 5 inches. (If you don’t remember the formula, it’s an easy web search.) Do you need parentheses in your calculation? Why or why not?

Using the 2019 US Census county estimates data, calculate a ratio of births to deaths for each county in New York state. Which region of the state generally saw a higher ratio of births to deaths in 2019?

Was the 2019 median county population estimate higher in California or New York?

7 JOINING TABLES IN A RELATIONAL DATABASE



In Chapter 2, I introduced the concept of a *relational database*, an application that supports data stored across multiple, related tables. In a relational model, each table typically holds data on a single entity—such as students, cars, purchases, houses—and each row in the table describes one of those entities. A process known as a *table join* allows us to link rows in one table to rows in other tables.

The concept of relational databases came from the British computer scientist Edgar F. Codd. While working for IBM in 1970, he published a paper called "A Relational Model of Data for Large Shared Data Banks." His ideas revolutionized database design and led to the development of SQL. With the relational model, you can build tables that eliminate duplicate data, are easier to maintain, and provide for increased flexibility in writing queries to get just the data you want.

Linking Tables Using JOIN

To connect tables in a query, we use a `JOIN ... ON` construct (or one of the other `JOIN` variants I'll cover in this chapter). A `JOIN`, which is part of the ANSI SQL standard, links one table to another in the database using a *Boolean* value expression in the `ON` clause. A commonly used syntax tests for equality and commonly takes this form:

```
SELECT *
FROM table_a JOIN table_b
ON table_a.key_column = table_b.foreign_key_column
```

This is similar to the basic `SELECT` you've already learned, but instead of naming one table in the `FROM` clause, we name a table, give the `JOIN` keyword, and then name a second table. The `ON` clause follows, where we place an expression using the equals comparison operator. When the query runs, it returns rows from both tables where the expression in the `ON` clause evaluates to `true`, meaning values in the specified columns are equal.

You can use any expression that evaluates to the *Boolean* results `true` or `false`. For example, you could match where values from one column are greater than or equal to values in the other:

```
ON table_a.key_column >= table_b.foreign_key_column
```

That's rare, but it's an option if your analysis requires it.

Relating Tables with Key Columns

Consider this example of relating tables with key columns: imagine you're a data analyst with the task of checking on a public agency's payroll spending by department. You file a Freedom of Information Act request for that agency's salary data, expecting to receive a simple spreadsheet listing each employee and their salary, arranged like this:

```
dept    location    first_name    last_name    salary
----    --------    ----------    ---------    ------
IT      Boston      Julia         Reyes        115300
IT      Boston      Janet         King          98000
Tax     Atlanta     Arthur        Pappas        72700
Tax     Atlanta     Michael       Taylor        89500
```

But that's not what arrives. Instead, the agency sends you a data dump from its payroll system: a dozen CSV files, each representing one table in its database. You read the document explaining the data layout (be sure to always ask for it!) and start to make sense of the columns in each table. Two tables stand out: one named `employees` and another named `departments`.

Using the code in Listing 7-1, let's create versions of these tables, insert rows, and examine how to join the data in both tables. Using the `analysis` database you've created for these exercises, run all the code, and then look at the data either by using a basic `SELECT` statement or by clicking the table name in pgAdmin and selecting **View/Edit Data▶All Rows**.

```
CREATE TABLE departments (
    dept_id integer,
    dept text,
    city text,
1 CONSTRAINT dept_key PRIMARY KEY (dept_id),
2 CONSTRAINT dept_city_unique UNIQUE (dept, city)
);

CREATE TABLE employees (
    emp_id integer,
    first_name text,
    last_name text,
    salary numeric(10,2),
3 dept_id integer REFERENCES departments (dept_id),
4 CONSTRAINT emp_key PRIMARY KEY (emp_id)
);

INSERT INTO departments
VALUES
    (1, 'Tax', 'Atlanta'),
    (2, 'IT', 'Boston');

INSERT INTO employees
VALUES
    (1, 'Julia', 'Reyes', 115300, 1),
    (2, 'Janet', 'King', 98000, 1),
    (3, 'Arthur', 'Pappas', 72700, 2),
    (4, 'Michael', 'Taylor', 89500, 2);
```

Listing 7-1: Creating the `departments` and `employees` tables

The two tables follow Codd's relational model in that each describes attributes about a single entity: the agency's departments and employees. In the `departments` table, you should see the following contents:

```
dept_id    dept    city
-------    ----    -------
      1    Tax     Atlanta
      2    IT      Boston
```

The `dept_id` column is the table's primary key. A *primary key* is a column or collection of columns whose values uniquely identify each row in a table. A valid primary key column enforces certain constraints:

The column or collection of columns must have a unique value for each row.

The column or collection of columns can't have missing values.

You define the primary key for `departments` 1 and `employees` 4 using a `CONSTRAINT` keyword, which I'll cover in depth with additional constraint types in Chapter 8. The values in `dept_id` uniquely identify each row in `departments`, and although this example contains only a department name and city, this table would likely include additional information, such as an address or contact information.

The `employees` table should have the following contents:

```
emp_id first_name last_name  salary   dept_id
------ ---------- --------- --------- -------
     1 Julia      Reyes     115300.00       1
     2 Janet      King       98000.00       1
     3 Arthur     Pappas     72700.00       2
     4 Michael    Taylor     89500.00       2
```

The values in `emp_id` uniquely identify each row in the `employees` table. To identify which department each employee works in, the table includes a `dept_id` column. The values in this column refer to values in the `departments` table's primary key. We call this a *foreign key*, which you add as a constraint 3 when creating the table. A foreign key constraint requires that its values already exist in the columns it references. Often, that's another table's primary key, but it can reference any columns that have unique values for each row. So, values in `dept_id` in the `employees` table must exist in `dept_id` in the `departments` table; otherwise, you can't add them. This helps enforce the integrity of the data. Unlike a primary key, a foreign key column can be empty, and it can contain duplicate values.

In this example, the `dept_id` associated with the employee `Julia Reyes` is `1`; this refers to the value of `1` in the `departments` table's primary key, `dept_id`. That tells us that `Julia Reyes` is part of the `Tax` department located in `Atlanta`.

NOTE

Primary key values need to be unique only within a table. That's why it's okay for both the `employees` *table and the* `departments` *table to have primary key values using the same numbers.*

The `departments` table also includes a `UNIQUE` constraint, which I'll discuss in more depth in "The UNIQUE Constraint" in the next chapter. Briefly, it guarantees that values in a column, or a combination of values in more than one column, are unique. Here, it requires that each row have a unique pair of values for `dept` and `city` 2, which helps avoid duplicate data —the table won't have two departments in Atlanta named `Tax`, for example. Often, you can use such unique combinations to create a *natural key* for a primary key, which we'll also discuss in the next chapter.

You might ask: what's the advantage of breaking data into components like this? Well, consider what this sample of data would look like if you had received it the way you initially thought you would, all in one table:

```
dept    location    first_name    last_name    salary
----    --------    ----------    ---------    ------
IT      Boston      Julia         Reyes        115300
IT      Boston      Janet         King          98000
Tax     Atlanta     Arthur        Pappas        72700
Tax     Atlanta     Michael       Taylor        89500
```

First, when you combine data from various entities in one table, inevitably you have to repeat information. This happens here: the department name and location are spelled out for each employee. This may be acceptable when the table consists of four rows like this, or even 4,000. But when a table holds millions of rows, repeating lengthy strings is redundant and wastes precious space.

Second, cramming all that data into one table makes managing the data difficult. What if the Marketing department changes its name to Brand Marketing? Each row in the table would require an update, which can introduce errors if someone mistakenly updates some but not all the rows. In this model, an update to a department name is much simpler—just change one row in a table.

Finally, the fact that information is organized, or *normalized*, across several tables doesn't prevent us from viewing it as a whole. We can always query the data using `JOIN` to bring columns from tables together.

Now that you know the basics of how tables can relate, let's look at how to join them in a query.

Querying Multiple Tables Using JOIN

When you join tables in a query, the database connects rows in both tables where the columns you specified for the join have values that result in the `ON` clause expression returning `true`. The query results then include columns from both tables if you requested them as part of the query. You also can use columns from the joined tables to filter results using a `WHERE` clause.

Queries that join tables are similar in syntax to basic `SELECT` statements. The difference is that the query also specifies the following:

The tables and columns to join, using a SQL `JOIN ... ON` construct

The type of join to perform using variations of the `JOIN` keyword

Let's look at the `JOIN ... ON` construct syntax first and then explore various types of joins. To join the example `employees` and `departments` tables and see all the related data from both, start by writing a query like the one in *Listing 7-2*.

```
1 SELECT *
2 FROM employees JOIN departments
3 ON employees.dept_id = departments.dept_id
  ORDER BY employees.dept_id;
```

Listing 7-2: Joining the `employees` and `departments` tables

In the example, you include an asterisk wildcard with the `SELECT` statement to include all columns from all tables used in the query 1. Next, in the `FROM` clause, you place the `JOIN` keyword 2 between the two tables you want to link. Finally, you specify the expression to evaluate using the `ON` clause 3. For each table, you provide the table name, a period, and the column that contains the key values. An equal sign goes between the two table and column names.

When you run the query, the results include all values from both tables where values in the `dept_id` columns match. In fact, even the `dept_id` column appears twice because you selected all columns of both tables:

```
emp_id   first_name   last_name   salary      dept_id
dept_id   dept   city
------   ----------   ---------   ---------   -------   -----
--   ----   -------
     1   Julia        Reyes       115300.00         1
1    Tax    Atlanta
     2   Janet        King         98000.00         1
1    Tax    Atlanta
     3   Arthur       Pappas       72700.00         2
2    IT     Boston
     4   Michael      Taylor       89500.00         2
2    IT     Boston
```

So, even though the data lives in two tables, each with a focused set of columns, you can query those tables to pull the relevant data back together. In "Selecting Specific Columns in a Join" later in this chapter, I'll show you how to retrieve only the columns you want from both tables.

Understanding JOIN Types

There's more than one way to join tables in SQL, and the type of join you'll use depends on how you want to retrieve data. The following list describes the different types of joins. While reviewing each, it's helpful to think of two tables side by side, one on the left of the `JOIN` keyword and the other on the right. A data-driven example of each join follows the list:

`JOIN` Returns rows from both tables where matching values are found in the joined columns of both tables. Alternate syntax is `INNER JOIN`.

`LEFT JOIN` Returns every row from the left table. When SQL finds a row with a matching value in the right table, values from that row are included in the results. Otherwise, no values from the right table are displayed.

`RIGHT JOIN` Returns every row from the right table. When SQL finds a row with a matching value in the left table, values from that row are included in the results. Otherwise, no values from the left table are displayed.

`FULL OUTER JOIN` Returns every row from both tables and joins the rows where values in the joined columns match. If there's no match for a value in either the left or right table, the query result contains no values for that table.

`CROSS JOIN` Returns every possible combination of rows from both tables.

Let's use data to see these joins in action. Say you have two simple tables that hold names of schools for a district that is planning future enrollments: `district_2020` and `district_2035`. There are four rows in `district_2020`:

```
id         school_2020
--    ------------------------
 1    Oak Street School
 2    Roosevelt High School
 5    Dover Middle School
 6    Webutuck High School
```

There are five rows in `district_2035`:

```
id        school_2035
--    ---------------------
 1    Oak Street School
 2    Roosevelt High School
 3    Morrison Elementary
```

```
4    Chase Magnet Academy
6    Webutuck High School
```

Notice that the district expects changes over time. Only schools with an `id` of `1`, `2`, and `6` exist in both tables, while others appear in just one of them. This scenario is common, and a common first task for a data analyst—especially if you have tables with many more rows than these—is to use SQL to identify which schools are present in both tables. Using different joins can help you find those schools, plus other details.

Again, using your `analysis` database, run the code in *Listing 7-3* to build and populate these two tables.

```
CREATE TABLE district_2020 (
  1 id integer CONSTRAINT id_key_2020 PRIMARY KEY,
    school_2020 text
);

CREATE TABLE district_2035 (
  2 id integer CONSTRAINT id_key_2035 PRIMARY KEY,
    school_2035 text
);

3 INSERT INTO district_2020 VALUES
      (1, 'Oak Street School'),
      (2, 'Roosevelt High School'),
      (5, 'Dover Middle School'),
      (6, 'Webutuck High School');

  INSERT INTO district_2035 VALUES
      (1, 'Oak Street School'),
      (2, 'Roosevelt High School'),
      (3, 'Morrison Elementary'),
      (4, 'Chase Magnet Academy'),
      (6, 'Webutuck High School');
```

Listing 7-3: Creating two tables to explore `JOIN` types

We create and fill two tables: the declarations for these should by now look familiar, but there's one new element: we add a primary key to each table. After the declaration for the `district_2020` `id` column 1 and the `district_2035` `id` column 2, the keywords `CONSTRAINT` *`key_name`* `PRIMARY`

KEY indicate that those columns will serve as the primary key for their table. That means for each row in both tables, the id column must be filled and contain a value that is unique for each row in that table. Finally, we use the familiar INSERT statements 3 to add the data to the tables.

JOIN

We use JOIN, or INNER JOIN, when we want to return only rows from both tables where values match in the columns we used for the join. To see an example of this, run the code in *Listing 7-4*, which joins the two tables you just made.

```
SELECT *
FROM district_2020 JOIN district_2035
ON district_2020.id = district_2035.id
ORDER BY district_2020.id;
```

Listing 7-4: Using JOIN

Similar to the method we used in *Listing 7-2*, we name the two tables to join on both sides of the JOIN keyword. Then, in the ON clause, we specify the expression we're using for the join, in this case equality in the id columns of both tables. Three school IDs exist in both tables, so the query returns only the three rows where those IDs match. Schools that exist in only one of the two tables don't appear in the result. Notice also that the columns from the table on the left side of the JOIN keyword display on the left of the result table:

```
id      school_2020      id      school_2035
-- --------------------- -- ---------------------
 1 Oak Street School      1 Oak Street School
 2 Roosevelt High School  2 Roosevelt High School
 6 Webutuck High School   6 Webutuck High School
```

When should you use JOIN? Typically, when you're working with well-structured, well-maintained datasets and need to find rows that exist in all the tables you're joining. Because JOIN doesn't provide rows that exist in only one of the tables, if you want to see all the data in one or more of the tables, use one of the other join types.

JOIN with USING

If you're using identical names for columns in a join's `ON` clause, you can reduce redundant output and simplify the query syntax by substituting a `USING` clause in place of the `ON` clause, as in Listing 7-5.

```
 SELECT *
 FROM district_2020 JOIN district_2035
1 USING (id)
 ORDER BY district_2020.id;
```

Listing 7-5: `JOIN` with `USING`

After naming the tables to join, we add `USING` 1 followed by, in parentheses, the name of the column for the join in both tables—in this case, `id`. If we're joining on more than one column, we separate them by commas in the parentheses. Run the query, and you should see these results:

```
id      school_2020            school_2035
-- --------------------- ---------------------
 1 Oak Street School     Oak Street School
 2 Roosevelt High School Roosevelt High School
 6 Webutuck High School  Webutuck High School
```

Note that `id`, which in the case of this `JOIN` is present in both tables and has identical values, is displayed just once. It's a simple, handy shorthand.

LEFT JOIN and RIGHT JOIN

In contrast to `JOIN`, the `LEFT JOIN` and `RIGHT JOIN` keywords each return all rows from one table and, when a row with a matching value in the other table exists, values from that row are included in the results. Otherwise, no values from the other table are displayed.

Let's look at `LEFT JOIN` in action first. Execute the code in Listing 7-6.

```
SELECT *
FROM district_2020 LEFT JOIN district_2035
ON district_2020.id = district_2035.id
ORDER BY district_2020.id;
```

Listing 7-6: Using `LEFT JOIN`

The result of the query shows all four rows from `district_2020`, which is on the left side of the join, as well as the three rows in `district_2035` where values match in the `id` columns. Because `district_2035` doesn't contain a value of `5` in its `id` column, there's no match, so `LEFT JOIN` returns an empty row on the right rather than omitting the entire row from the left table as with `JOIN`. Finally, the rows from `district_2035` that don't match any values in `district_2020` are omitted from the results:

```
id      school_2020      id      school_2035
-- --------------------- -- ---------------------
 1 Oak Street School      1 Oak Street School
 2 Roosevelt High School  2 Roosevelt High School
 5 Dover Middle School
 6 Webutuck High School   6 Webutuck High School
```

We see similar but opposite behavior by running `RIGHT JOIN`, as in Listing 7-7.

```
SELECT *
FROM district_2020 RIGHT JOIN district_2035
ON district_2020.id = district_2035.id
ORDER BY district_2035.id;
```

Listing 7-7: Using `RIGHT JOIN`

This time, the query returns all rows from `district_2035`, which is on the right side of the join, plus rows from `district_2020` where the `id` columns have matching values. The query result omits the row of `district_2020` where there's no match with `district_2035` on `id`:

```
id      school_2020      id      school_2035
-- --------------------- -- ---------------------
 1 Oak Street School      1 Oak Street School
 2 Roosevelt High School  2 Roosevelt High School
                          3 Morrison Elementary
                          4 Chase Magnet Academy
 6 Webutuck High School   6 Webutuck High School
```

You'd use either of these join types in a few circumstances:

You want your query results to contain all the rows from one of the tables.

You want to look for missing values in one of the tables. An example is when you're comparing data about an entity representing two different time periods.

When you know some rows in a joined table won't have matching values.

As with `JOIN`, you can substitute the `USING` clause for the `ON` clause if the tables meet the criteria.

FULL OUTER JOIN

When you want to see all rows from both tables in a join, regardless of whether any match, use the `FULL OUTER JOIN` option. To see it in action, run *Listing 7-8*.

```
SELECT *
FROM district_2020 FULL OUTER JOIN district_2035
ON district_2020.id = district_2035.id
ORDER BY district_2020.id;
```

Listing 7-8: Using `FULL OUTER JOIN`

The result gives every row from the left table, including matching rows and blanks for missing rows from the right table, followed by any leftover missing rows from the right table:

```
id      school_2020      id      school_2035
-- --------------------- -- ---------------------
 1 Oak Street School      1 Oak Street School
 2 Roosevelt High School  2 Roosevelt High School
 5 Dover Middle School
 6 Webutuck High School   6 Webutuck High School
                          3 Morrison Elementary
                          4 Chase Magnet Academy
```

A full outer join is admittedly less useful and used less often than inner and left or right joins. Still, you can use it for a couple of tasks: to link two data sources that partially overlap or to visualize the degree to which tables share matching values.

CROSS JOIN

In a `CROSS JOIN` query, the result (also known as a *Cartesian product*) lines up each row in the left table with each row in the right table to present all possible combinations of rows. Listing 7-9 shows the `CROSS JOIN` syntax; because the join doesn't need to find matches between key columns, there's no need to provide an `ON` clause.

```
SELECT *
FROM district_2020 CROSS JOIN district_2035
ORDER BY district_2020.id, district_2035.id;
```

Listing 7-9: *Using* `CROSS JOIN`

The result has 20 rows—the product of four rows in the left table times five rows in the right:

```
id      school_2020      id      school_2035
-- --------------------- -- ---------------------
 1 Oak Street School      1 Oak Street School
 1 Oak Street School      2 Roosevelt High School
 1 Oak Street School      3 Morrison Elementary
 1 Oak Street School      4 Chase Magnet Academy
 1 Oak Street School      6 Webutuck High School
 2 Roosevelt High School  1 Oak Street School
 2 Roosevelt High School  2 Roosevelt High School
 2 Roosevelt High School  3 Morrison Elementary
 2 Roosevelt High School  4 Chase Magnet Academy
 2 Roosevelt High School  6 Webutuck High School
 5 Dover Middle School    1 Oak Street School
 5 Dover Middle School    2 Roosevelt High School
 5 Dover Middle School    3 Morrison Elementary
 5 Dover Middle School    4 Chase Magnet Academy
 5 Dover Middle School    6 Webutuck High School
 6 Webutuck High School   1 Oak Street School
 6 Webutuck High School   2 Roosevelt High School
 6 Webutuck High School   3 Morrison Elementary
 6 Webutuck High School   4 Chase Magnet Academy
 6 Webutuck High School   6 Webutuck High School
```

Unless you want to take an extra-long coffee break, I suggest avoiding a `CROSS JOIN` query on large tables. Two tables with 250,000 records each would produce a result set of 62.5 *billion* rows and tax even the hardiest server. A more practical use would be generating data to create a checklist, such as all colors you'd want to offer for each of a handful of shirt styles in a store.

Using NULL to Find Rows with Missing Values

Any time you join tables, it's wise to investigate whether the key values in one table appear in the other, and which values are missing, if any. Discrepancies happen for all sorts of reasons. Some data may have changed over time. For example, a table of new products will likely contain codes that aren't present in an older product table. Or there could be problems such as a clerical errors or incomplete output from the database. All this is important context for making correct inferences about the data.

When you have only a handful of rows, eyeballing the data is an easy way to look for rows with missing data, as we did in the previous join examples. For large tables, you need a better strategy: filtering to show all rows without a match. To do this, we employ the keyword `NULL`.

In SQL, `NULL` is a special value that represents a condition in which there's no data present or where the data is unknown because it wasn't included. For example, if a person filling out an address form skips the "Middle Initial" field, rather than storing an empty string in the database, we'd use `NULL` to represent the unknown value. It's important to keep in mind that `NULL` is different from `0` or an empty string that you'd place in a text column using two quotes (`''`). Both those values could have some unintended meaning that's open to misinterpretation, so you use `NULL` to show that the value is unknown. And unlike `0` or an empty string, you can use `NULL` across data types.

When a SQL join returns empty rows in one of the tables, those columns don't come back empty but instead come back with the value `NULL`. In *Listing 7-10*, we'll find those rows by adding a `WHERE` clause to filter for `NULL` by using the phrase `IS NULL` on the `id` column of the `district_2035` table. If we wanted to look for columns *with* data, we'd use `IS NOT NULL`.

```
SELECT *
FROM district_2020 LEFT JOIN district_2035
ON district_2020.id = district_2035.id
WHERE district_2035.id IS NULL;
```

Listing 7-10: Filtering to show missing values with `IS NULL`

Now the result of the join shows only the one row from the table on the left of the join that didn't have a match in the table on the right. This is commonly referred to as an *anti-join*.

```
id     school_2020       id       school_2035
-- ------------------- ------ ---------------------
 5 Dover Middle School
```

It's easy to reverse the output to see rows on the table on the right of the join that have no matches with the table on the left. You'd change the query to use a `RIGHT JOIN` and modify the `WHERE` clause to filter on `district_2020.id IS NULL`.

NOTE

pgAdmin displays `NULL` *values in results tables with the designation* `[null]`*. If you're using the* `psql` *command-line tool that we'll discuss in Chapter 18, by default* `NULL` *values are displayed as blanks. You can change that behavior to mimic pgAdmin by running the command* `\pset null '[null]'` *at the* `psql` *prompt.*

Understanding the Three Types of Table Relationships

Part of the science (or art, some may say) of joining tables involves understanding how the database designer intends for the tables to relate, also known as the database's *relational model*. There are three types of table relationships: one to one, one to many, and many to many.

One-to-One Relationship

In our `JOIN` example in *Listing 7-4*, there are no duplicate `id` values in either table: only one row in the `district_2020` table exists with an `id` of `1`, and only one row in the `district_2035` table has an `id` of `1`. That means any given `id` in either table will find no more than one match in the other table. In database parlance, this is called a *one-to-one* relationship. Consider another example: joining two tables with state-by-state census data. One table might contain household income data and the other data is about educational attainment. Both tables would have 51 rows (one for each state plus Washington, D.C.), and if we joined them on a key such as state name, state abbreviation, or a standard geography code, we'd have only one match for each key value in each table.

One-to-Many Relationship

In a *one-to-many* relationship, a key value in one table will have multiple matching values in another table's joined column. Consider a database that tracks automobiles. One table would hold data on manufacturers, with one row each for Ford, Honda, Tesla, and so on. A second table with model names, such as Mustang, Civic, Model 3, and Accord, would have several rows matching each row in the manufacturers' table.

Many-to-Many Relationship

A *many-to-many* relationship exists when multiple items in one table can relate to multiple items in another table, and vice versa. For example, in a baseball league, each player can be assigned to multiple positions, and each position can be played by multiple players. Because of this complexity, many-to-many relationships usually feature a third, intermediate table in between the two. In the case of the baseball league, a database might have a `players` table, a `positions` table, and a third called `players_positions` that has two columns that support the many-to-many relationship: the `id` from the `players` table and the `id` from the `positions` table.

Understanding these relationships is essential because it helps us discern whether the results of queries accurately reflect the structure of the database.

Selecting Specific Columns in a Join

So far, we've used the asterisk wildcard to select all columns from both tables. That's okay for quick data checks, but more often you'll want to specify a subset of columns. You can focus on just the data you want and avoid inadvertently changing the query results if someone adds a new column to a table.

As you learned in single-table queries, to select particular columns you use the `SELECT` keyword followed by the desired column names. When joining tables, it's a best practice to include the table name along with the column. The reason is that more than one table can contain columns with the same name, which is certainly true of our joined tables so far.

Consider the following query, which tries to fetch an `id` column without naming the table:

```
SELECT id
FROM district_2020 LEFT JOIN district_2035
ON district_2020.id = district_2035.id;
```

Because `id` exists in both `district_2020` and `district_2035`, the server throws an error that appears in pgAdmin's results pane: `column reference "id" is ambiguous`. It's not clear which table `id` belongs to.

To fix the error, we need to add the table name in front of each column we're querying, as we do in the `ON` clause. *Listing 7-11* shows the syntax, specifying that we want the `id` column from `district_2020`. We're also fetching the school names from both tables.

```
SELECT district_2020.id,
       district_2020.school_2020,
       district_2035.school_2035
FROM district_2020 LEFT JOIN district_2035
ON district_2020.id = district_2035.id
ORDER BY district_2020.id;
```

Listing 7-11: Querying specific columns in a join

We simply prefix each column name with the table it comes from, and the rest of the query syntax is the same. The result returns the requested columns from each table:

```
id      school_2020           school_2035
-- --------------------- ----------------------
 1 Oak Street School     Oak Street School
 2 Roosevelt High School Roosevelt High School
 5 Dover Middle School
 6 Webutuck High School  Webutuck High School
```

We can also add the `AS` keyword we used previously with census data to make it clear in the results that the `id` column is from `district_2020`. The syntax would look like this:

```
SELECT district_2020.id AS d20_id, ...
```

This would display the name of the `district_2020 id` column as `d20_id` in the results.

Simplifying JOIN Syntax with Table Aliases

Specifying the table for a column is easy enough, but repeating a lengthy table name for multiple columns clutters your code. One of the best ways to serve your colleagues is to write code that's readable, which should generally not involve making them wade through a table name repeated over 25 columns! One way to write more concise code is to use a shorthand approach called *table aliases*.

To create a table alias, we place a character or two after the table name when we declare it in the `FROM` clause. (You can use more than a couple of characters for an alias, but if the goal is to simplify code, don't go overboard.) Those characters then serve as an alias we can use instead of the full table name anywhere we reference the table in the code. Listing 7-12 demonstrates how this works.

```
SELECT d20.id,
       d20.school_2020,
       d35.school_2035
1 FROM district_2020 AS d20 LEFT JOIN district_2035 AS d35
```

```
ON d20.id = d35.id
ORDER BY d20.id;
```

Listing 7-12: Simplifying code with table aliases

In the `FROM` clause, we declare the alias `d20` to represent `district_2020` and the alias `d35` to represent `district_2035` 1 using the `AS` keyword. Both aliases are shorter than the table names but still meaningful. Once that's in place, we can use the aliases instead of the full table names everywhere else in the code. Immediately, our SQL looks more compact, and that's ideal. Note that the `AS` keyword is optional here; you can omit it when declaring an alias for both table names and column names.

Joining Multiple Tables

Of course, SQL joins aren't limited to two tables. We can continue adding tables to the query as long as we have columns with matching values to join on. Let's say we obtain two more school-related tables and want to join them to `district_2020` in a three-table join. The `district_2020_enrollment` table has the number of students per school:

```
id    enrollment
--    ----------
 1           360
 2          1001
 5           450
 6           927
```

The `district_2020_grades` table contains the grade levels housed in each building:

```
id    grades
--    ------
 1    K-3
 2    9-12
 5    6-8
 6    9-12
```

To write the query, we'll use *Listing 7-13* to create the tables, load the data, and run a query to join them to `district_2020`.

```
CREATE TABLE district_2020_enrollment (
    id integer,
    enrollment integer
);

CREATE TABLE district_2020_grades (
    id integer,
    grades varchar(10)
);

INSERT INTO district_2020_enrollment
VALUES
    (1, 360),
    (2, 1001),
    (5, 450),
    (6, 927);

INSERT INTO district_2020_grades
VALUES
    (1, 'K-3'),
    (2, '9-12'),
    (5, '6-8'),
    (6, '9-12');

SELECT d20.id,
       d20.school_2020,
       en.enrollment,
       gr.grades
1 FROM district_2020 AS d20 JOIN district_2020_enrollment AS en
     ON d20.id = en.id
2 JOIN district_2020_grades AS gr
     ON d20.id = gr.id
ORDER BY d20.id;
```

Listing 7-13: Joining multiple tables

After we run the `CREATE TABLE` and `INSERT` portions of the script, we have new `district_2020_enrollment` and `district_2020_grades` tables, each with records that relate to `district_2020` from earlier in the chapter. We then connect all three tables.

In the SELECT query, we join district_2020 to district_2020_enrollment 1 using the tables' id columns. We also declare table aliases to keep the code compact. Next, the query joins district_2020 to district_2020_grades, again on the id columns 2.

Our result now includes columns from all three tables:

```
id      school_2020      enrollment grades
-- -------------------- ---------- ------
 1 Oak Street School           360 K-3
 2 Roosevelt High School      1001 9-12
 5 Dover Middle School         450 6-8
 6 Webutuck High School        927 9-12
```

If you need to, you can add even more tables to the query using additional joins. You can also join on different columns, depending on the tables' relationships. Although there is no hard limit in SQL to the number of tables you can join in a single query, some database systems might impose one. Check the documentation.

Combining Query Results with Set Operators

Certain instances require us to re-order our data so that columns from various tables aren't returned side by side, as a join produces, but brought together into one result. Examples include required input formats for JavaScript-based data visualizations or analysis with libraries used in the R and Python programming languages. One way to manipulate our data this way is to use the ANSI standard SQL *set operators* UNION, INTERSECT, and EXCEPT. Set operators combine the results of multiple SELECT queries. Here's a quick look at what each does:

UNION Given two queries, it appends the rows in the results of the second query to the rows returned by the first query and removes duplicates, producing a combined set of distinct rows. Modifying the syntax to UNION ALL will return all rows, including duplicates.

INTERSECT Returns only rows that exist in the results of both queries and removes duplicates.

EXCEPT Returns rows that exist in the results of the first query but not in the results of the second query. Duplicates are removed.

For each of these, both queries must produce the same number of columns, and the resulting columns from both queries must have compatible data types. Let's continue using our school district tables for brief examples of how they work.

UNION and UNION ALL

In *Listing 7-14*, we use `UNION` to combine queries that retrieve all rows from both `district_2020` and `district_2035`.

```
  SELECT * FROM district_2020
1 UNION
  SELECT * FROM district_2035
2 ORDER BY id;
```

Listing 7-14: Combining query results with `UNION`

The query consists of two complete `SELECT` statements with the `UNION` keyword 1 placed between them. The `ORDER BY` 2 on the `id` column happens after the set operation occurs and thus can't be listed as part of each `SELECT`. From our work with this data already, you know that these queries will return several rows that are identical in both tables. But by merging the queries with `UNION`, our results eliminate duplicates:

```
id      school_2020
-- ---------------------
 1 Oak Street School
 2 Roosevelt High School
 3 Morrison Elementary
 4 Chase Magnet Academy
 5 Dover Middle School
 6 Webutuck High School
```

Notice that the names of the schools are in the column `school_2020`, which is part of the first query's results. The school names in the second query's column `school_2035` from the `district_2035` table were simply appended to the results from the first query. For that reason, the columns in the second query must match those in the first and have compatible data types.

If we want the results to include duplicate rows, we substitute `UNION ALL` for `UNION` in the query, as in Listing 7-15.

```
SELECT * FROM district_2020
UNION ALL
SELECT * FROM district_2035
ORDER BY id;
```

Listing 7-15: Combining query results with `UNION ALL`

That produces all rows, with duplicates included:

```
id      school_2020
-- ---------------------
 1 Oak Street School
 1 Oak Street School
 2 Roosevelt High School
 2 Roosevelt High School
 3 Morrison Elementary
 4 Chase Magnet Academy
 5 Dover Middle School
 6 Webutuck High School
 6 Webutuck High School
```

Finally, it's often helpful to customize merged results. You may want to know, for example, which table values in each row came from, or you may want to include or exclude certain columns. Listing 7-16 shows one example using `UNION ALL`.

```
1 SELECT '2020' AS year,
       2 school_2020 AS school
  FROM district_2020

  UNION ALL

  SELECT '2035' AS year,
         school_2035
  FROM district_2035
  ORDER BY school, year;
```

Listing 7-16: Customizing a `UNION` query

In the first query's `SELECT` statement 1, we designate the string `2020` as the value to fill a column named `year`. We also do this in the second query using `2035` as the string. This is similar to the technique you employed in the section "Adding a Value to a Column During Import" in Chapter 5. Then, we rename the `school_2020` column 2 as `school` because it will show schools from both years.

Execute the query to see the results:

```
year        school
---- --------------------
2035 Chase Magnet Academy
2020 Dover Middle School
2035 Morrison Elementary
2020 Oak Street School
2035 Oak Street School
2020 Roosevelt High School
2035 Roosevelt High School
2020 Webutuck High School
2035 Webutuck High School
```

Now our query produces a year designation for each school, and we can see, for example, that the row with Dover Middle School comes from the result of querying the `district_2020` table.

INTERSECT and EXCEPT

Now that you know how to use `UNION`, you can apply the same concepts to `INTERSECT` and `EXCEPT`. Listing 7-17 shows both, which you can run separately to see how the results differ.

```
  SELECT * FROM district_2020
1 INTERSECT
  SELECT * FROM district_2035
  ORDER BY id;

  SELECT * FROM district_2020
2 EXCEPT
  SELECT * FROM district_2035
  ORDER BY id;
```

Listing 7-17: Combining query results with `INTERSECT` and `EXCEPT`

The query using `INTERSECT` 1 returns just the rows that exist in the results of both queries and eliminates duplicates:

```
id  school_2020
-- --------------
 1 Oak Street School
 2 Roosevelt High School
 6 Webutuck High School
```

The query using `EXCEPT` 2 returns rows that exist in the first query but not in the second, also eliminating duplicates if present:

```
id     school_2020
-- -------------------
 5 Dover Middle School
```

Along with `UNION`, queries using `INTERSECT` and `EXCEPT` give you plenty of ability to arrange and examine your data.

Finally, let's return briefly to joins to see how you can perform calculations on numbers in different tables.

Performing Math on Joined Table Columns

The math functions we explored in Chapter 6 are just as usable when working with joined tables. We need to include the table name when referencing a column in an operation, as we did when selecting table columns. If you work with any data that has a new release at regular intervals, you'll find this concept useful for joining a newly released table to an older one and exploring how values have changed.

That's certainly what I and many journalists do each time a new set of census data is released. We'll load the new data and try to find patterns in the growth or decline of the population, income, education, and other indicators. Let's look at how to do this by revisiting the `us_counties_pop_est_2019` table we created in Chapter 5 and loading similar county data that shows 2010 county population estimates into a new table. To make the table, import the data, and join it to the 2019 estimates, run the code in Listing 7-18.

```
1 CREATE TABLE us_counties_pop_est_2010 (
      state_fips text,
      county_fips text,
      region smallint,
      state_name text,
      county_name text,
      estimates_base_2010 integer,
      CONSTRAINT counties_2010_key PRIMARY KEY (state_fips,
  county_fips)
  );

2 COPY us_counties_pop_est_2010
  FROM 'C:\YourDirectory\us_counties_pop_est_2010.csv'
  WITH (FORMAT CSV, HEADER);

3 SELECT c2019.county_name,
         c2019.state_name,
         c2019.pop_est_2019 AS pop_2019,
         c2010.estimates_base_2010 AS pop_2010,
         c2019.pop_est_2019 - c2010.estimates_base_2010 AS
  raw_change,
       4 round( (c2019.pop_est_2019::numeric -
  c2010.estimates_base_2010)
             / c2010.estimates_base_2010 * 100, 1 ) AS
  pct_change
  FROM us_counties_pop_est_2019 AS c2019
      JOIN us_counties_pop_est_2010 AS c2010
5 ON c2019.state_fips = c2010.state_fips
      AND c2019.county_fips = c2010.county_fips
6 ORDER BY pct_change DESC;
```

Listing 7-18: Performing math on joined census tables

In this code, we're building on earlier foundations. We have the familiar `CREATE TABLE` statement 1, which for this exercise includes state, county, and region codes, and we have columns with the names of the states and counties. It also includes an `estimates_base_2010` column that has the Census Bureau's estimated 2010 population for each county (the Census

Bureau modifies its complete, every-10-year count to create a base number for comparisons with estimates later in the decade). The `COPY` statement 2 imports a CSV file with the census data; you can find *us_counties_pop_est_2010.csv* along with all of the book's resources at *https://nostarch.com/practical-sql-2nd-edition/*. After you've downloaded the file, you'll need to change the file path to the location where you saved it.

When you've finished the import, you should have a table named `us_counties_pop_est_2010` with 3,142 rows. Now that we have tables with population estimates for 2010 and 2019, it makes sense to calculate the percent change in population for each county between those years. Which counties have led the nation in growth? Which ones have seen a decline in population?

We'll use the percent change formula we used in Chapter 6 to get the answer. The `SELECT` statement 3 includes the county and state names from the 2019 table, which is aliased with `c2019`. Next are the population estimate columns from the 2019 and 2010 tables, both renamed using `AS` to simplify their names in the results. To get the raw change in population, we subtract the 2010 estimates base from the 2019 estimates, and to find the percent change, we employ the formula 4 and round the result to one decimal point.

We join by matching values in two columns in both tables: `state_fips` and `county_fips` 5. The reason to join on two columns instead of one is that in both tables, the combination of a state code and a county code represents a unique county. We combine the two conditions using the `AND` logical operator. Using that syntax, rows are joined when both conditions are satisfied. Finally, we sort the results in descending order by percent change 6 so we can see the fastest growers at the top.

That's a lot of work, but it's worth it. Here's what the first five rows of the results indicate:

```
county_name        state_name    pop_2019  pop_2010
raw_change  pct_change
---------------    ----------    --------  --------  ---------
-  ----------
McKenzie County    North Dakota     15024      6359
```

```
8665       136.3
Loving County     Texas               169        82
87       106.1
Williams County   North Dakota      37589     22399
15190        67.8
Hays County       Texas            230191    157103
73088        46.5
Wasatch County    Utah              34091     23525
10566        44.9
```

Two counties, McKenzie in North Dakota and Loving in Texas, more than doubled their populations from 2010 to 2019, with other North Dakota and Texas counties showing substantial gains. Each of these places has its own story. For McKenzie County and others in North Dakota, a boom in oil and gas exploration in the Bakken geological formation is behind the surge. That's just one valuable insight we've extracted from this analysis and a starting point for understanding national population trends.

Wrapping Up

Given that table relationships are foundational to database architecture, learning to join tables in queries allows you to handle many of the more complex datasets you'll encounter. Experimenting with the different types of joins on tables can tell you a great deal about how data has been gathered and reveal when there's a quality issue. Make trying various joins a routine part of your exploration of a new dataset.

Moving forward, we'll continue building on these bigger concepts as we drill deeper into finding information in datasets and working with the nuances of handling data types and making sure we have quality data. But first, we'll look at one more foundational element: employing best practices to build reliable, speedy databases with SQL.

TRY IT YOURSELF

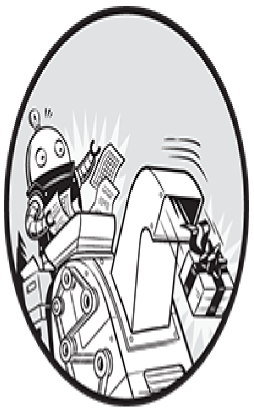
Continue your exploration of joins and set operators with these exercises:

According to the census population estimates, which county had the greatest percentage loss of population between 2010 and 2019? Try an internet search to find out what happened. (Hint: The decrease is related to a particular type of facility.)

Apply the concepts you learned about `UNION` to create query results that merge queries of the census county population estimates for 2010 and 2019. Your results should include a column called `year` that specifies the year of the estimate for each row in the results.

Using the `percentile_cont()` function from Chapter 6, determine the median of the percent change in estimated county population between 2010 and 2019.

8
TABLE DESIGN THAT WORKS FOR YOU



Obsession with order and detail can be a good thing. When you're running out the door, it's reassuring to see your keys hanging on the hook where you *always* leave them. The same holds true for database design. When you need to excavate a nugget of information from dozens of tables and millions of rows, you'll appreciate a dose of that same detail obsession. With data organized into a finely tuned, smartly named set of tables, the analysis experience becomes much more manageable.

In this chapter, I'll build on Chapter 7 by introducing best practices for organizing and speeding up SQL databases, whether they're yours or ones you inherit for analysis. We'll dig deeper into table design by exploring naming rules and conventions, ways to maintain the integrity of your data, and how to add indexes to tables to speed up queries.

Following Naming Conventions

Programming languages tend to have their own style patterns, and even various factions of SQL coders prefer certain conventions when naming tables, columns, and other objects (called *identifiers*). Some like *camel case*, as in `berrySmoothie`, where words are strung together and the first letter of each word is capitalized except for the first word. *Pascal case*, as in `BerrySmoothie`, follows a similar pattern but capitalizes the first letter too. With *snake case*, as in `berry_smoothie`, all the words are lowercase and separated by underscores.

You'll find passionate supporters of each naming convention, with some preferences tied to individual database applications or programming languages. For example, Microsoft uses Pascal case in the documentation for its SQL Server database. In this book, for PostgreSQL-related reasons I'll explain in a moment, we're using snake case, as in the table `us_counties_pop_est_2019`. Whichever convention you prefer or find yourself required to use, it's important to apply it consistently. Be sure to check whether your organization has a style guide or offer to collaborate on one, and then follow it religiously.

Mixing styles or following none generally leads to a mess. For example, imagine connecting to a database and finding the following collection of tables:

```
Customers

customers

custBackup

customer_analysis

customer_test2

customer_testMarch2012

customeranalysis
```

You would have questions. For one, which table actually holds the current data on customers? A disorganized naming scheme—and a general lack of tidiness—makes it hard for others to dive into your data and makes it challenging for you to pick up where you left off.

Let's explore considerations related to naming identifiers and suggestions for best practices.

Quoting Identifiers Enables Mixed Case

Regardless of any capitalization you supply, PostgreSQL treats identifiers as lowercase unless you place double quotes around the identifier. Consider these two `CREATE TABLE` statements for PostgreSQL:

```
CREATE TABLE customers (
    customer_id text,
    --snip--
);

CREATE TABLE Customers (
    customer_id text,
    --snip--
);
```

When you execute these statements in order, the first command creates a table called `customers`. The second statement, rather than creating a separate table called `Customers`, will throw an error: `relation "customers" already exists`. Because you didn't quote the identifier, PostgreSQL treats `customers` and `Customers` as the same identifier, disregarding the case. To preserve the uppercase letter and create a separate table named `Customers`, you must surround the identifier with quotes, like this:

```
CREATE TABLE "Customers" (
    customer_id serial,
    --snip--
);
```

However, because this requires that to query `Customers` rather than `customers`, you have to quote its name in the `SELECT` statement:

```
SELECT * FROM "Customers";
```

That can be a chore to remember and makes a user vulnerable to a mix-up. Make sure your tables have names that are clear and distinct from other tables in the database.

Pitfalls with Quoting Identifiers

Quoting identifiers also allows you to use characters not otherwise allowed, including spaces. That may appeal to some folks, but there are negatives. You may want to throw quotes around `"trees planted"` as a column name in a reforestation database, but then all users will have to provide quotes on every reference to that column. Omit the quotes in a query, and the database will respond with an error, identifying `trees` and `planted` as separate columns and responding that `trees` does not exist. A more readable and reliable option is to use snake case, as in `trees_planted`.

Quotes also let you use SQL *reserved keywords*, which are words that have special meaning in SQL. You've already encountered several, such as `TABLE`, `WHERE`, or `SELECT`. Most database developers frown on using reserved keywords as identifiers. At a minimum it's confusing, and at worst neglecting or forgetting to quote that keyword later may result in an error because the database will interpret the word as a command instead of an identifier.

NOTE

For PostgreSQL, you can find a list of keywords documented at https://www.postgresql.org/docs/current/sql-keywords-appendix.html. In addition, many code editors and database tools, including pgAdmin, automatically highlight keywords in a particular color.

Guidelines for Naming Identifiers

Given the extra burden of quoting and its potential problems, it's best to keep your identifier names simple, unquoted, and consistent. Here are my recommendations:

Use snake case. Snake case is readable and reliable, as shown in the earlier `trees_planted` example. It's used throughout the official PostgreSQL documentation and helps make multiword names easy to understand: `video_on_demand` makes more sense at a glance than `videoondemand`.

Make names easy to understand and avoid cryptic abbreviations. If you're building a database related to travel, `arrival_time` is a clearer column name than `arv_tm`.

For table names, use plurals. Tables hold rows, and each row represents one instance of an entity. So, use plural names for tables, such as `teachers`, `vehicles`, or `departments`. I do make exceptions at times. For example, to preserve the names of imported CSV files, I use them as a table name, especially when they are one-off imports.

Mind the length. The maximum number of characters allowed for an identifier name varies by database application: the SQL standard is 128 characters, but PostgreSQL limits you to 63, and older Oracle systems have a maximum of 30. If you're writing code that may get reused in another database system, lean toward shorter identifier names.

When making copies of tables, use names that will help you manage them later. One method is to append a `_`*`YYYY_MM_DD`* date to the table name when you create the copy, such as `vehicle_parts_2021_04_08`. An additional benefit is that the table names will sort in date order.

Controlling Column Values with Constraints

You can maintain further control over the data a column will accept by using certain constraints. A column's data type broadly defines the kind of data it will accept: integers versus characters, for example. Additional constraints let us further specify acceptable values based on rules and logical tests. With constraints, we can avoid the "garbage in, garbage out" phenomenon, which happens when poor-quality data results in inaccurate or incomplete analysis. Well-designed constraints help maintain the quality of the data and ensure the integrity of the relationships among tables.

In Chapter 7, you learned about *primary* and *foreign keys*, which are two of the most commonly used constraints. SQL also has the following constraint types:

`CHECK` Allows only those rows where a supplied Boolean expression evaluates to `true`

`UNIQUE` Ensures that values in a column or group of columns are unique in each row in the table

`NOT NULL` Prevents `NULL` values in a column

We can add constraints in two ways: as a *column constraint* or as a *table constraint*. A column constraint applies only to that column. We declare it with the column name and data type in the `CREATE TABLE` statement, and it gets checked whenever a change is made to the column. With a table constraint, we can supply criteria that apply to one or more columns. We declare it in the `CREATE TABLE` statement immediately after defining all the table columns, and it gets checked whenever a change is made to a row in the table.

Let's explore these constraints, their syntax, and their usefulness in table design.

Primary Keys: Natural vs. Surrogate

As explored in Chapter 7, a *primary key* is a column or collection of columns whose values uniquely identify each row in a table. A primary key is a constraint, and it imposes two rules on the column or columns that make up the key:

Values must be unique for each row.

No column can have missing values.

In a table of products stored in a warehouse, the primary key could be a column of unique product codes. In the simple primary key examples in "Relating Tables with Key Columns" in Chapter 7, our tables had a primary key made from a single ID column with an integer inserted by us, the user. Often, the data will suggest the best path and help us decide whether to use a *natural key* or a *surrogate key* as the primary key.

Using Existing Columns for Natural Keys

A natural key uses one or more of the table's existing columns that meet the criteria for a primary key: unique for every row and never empty. Values in the columns can change as long as the new value doesn't cause a violation of the constraint.

A natural key might be a driver's license identification number issued by a local Department of Motor Vehicles. Within a governmental jurisdiction, such as a state in the United States, we'd reasonably expect that all drivers would receive a unique ID on their licenses, which we could store as `driver_id`. However, if we were compiling a national driver's license database, we might not be able to make that assumption; several states could independently issue the same ID code. In that case, the `driver_id` column may not have unique values and cannot be used as the natural key. As a solution, we could create a *composite primary key* by combining `driver_id` with a column holding the state name, which would give us a unique combination for each row. For example, both rows in this table have a unique combination of the `driver_id` and `st` columns:

```
driver_id   st  first_name  last_name
----------  --  ----------  ---------
10302019    NY  Patrick     Corbin
10302019    FL  Howard      Kendrick
```

We'll visit both approaches in this chapter, and as you work with data, keep an eye out for values suitable for natural keys. A part number, a serial number, or a book's ISBN are all good examples.

Introducing Columns for Surrogate Keys

A *surrogate* key is a single column that you fill with artificial values; we might use it when a table doesn't have data that supports creating a natural primary key. The surrogate key might be a sequential number autogenerated by the database. We've already done this with the serial data type and the `IDENTITY` syntax (covered in "Auto-Incrementing Integers" in Chapter 4). A table using an autogenerated integer for a surrogate key might look like this:

```
id  first_name  last_name
--  ----------  ---------
 1  Patrick     Corbin
 2  Howard      Kendrick
 3  David       Martinez
```

Some developers like to use a *universally unique identifier (UUID)*, which is a code comprised of 32 hexadecimal digits in groups separated by hyphens. Often, UUIDs are used to identify computer hardware or software and look like the following:

```
2911d8a8-6dea-4a46-af23-d64175a08237
```

PostgreSQL offers a UUID data type as well as two modules that generate UUIDs: `uuid-ossp` and `pgcrypto`. The PostgreSQL documentation at *https://www.postgresql.org/docs/current/datatype-uuid.html* is a good starting point for diving deeper.

NOTE

Exercise caution when considering UUIDs for a surrogate key. Because of their size, they are inefficient compared with options such as `bigint`.

Evaluating the Pros and Cons of Key Types

There are well-reasoned arguments for using either type of primary key, but both have drawbacks. Points to consider about natural keys include the following:

The data already exists in the table, so you don't need to add a column to create a key.

Because the natural key data has meaning, it can reduce the need to join tables when querying.

If your data changes in a way that violates the requirements for a key—the sudden appearance of duplicate values, for instance—you'll be forced to change the setup of the table.

Here are points to consider about surrogate keys:

Because a surrogate key doesn't have any meaning in itself and its values are independent of the data in the table, you're not limited by the key structure if your data changes later.

Key values are guaranteed to be unique.

Adding a column for a surrogate key requires more space.

In a perfect world, a table should have one or more columns that can serve as a natural key, such as a unique product code in a table of products. But real-world limitations arise all the time. In a table of employees, it might be difficult to find any single column, or even multiple columns, that would be unique on a row-by-row basis to serve as a primary key. In such cases where you can't reconsider the table structure, you may need to use a surrogate key.

Creating a Single-Column Primary Key

Let's work through several primary key examples. In "Understanding JOIN Types" in Chapter 7, you created primary keys on the `district_2020` and `district_2035` tables to try `JOIN` types. In fact, these were surrogate keys: in both tables, you created columns called `id` to use as the key and used the keywords `CONSTRAINT` *`key_name`* `PRIMARY KEY` to declare them as primary keys.

There are two ways to declare constraints: as a column constraint or as a table constraint. In *Listing 8-1*, we try both methods, declaring a primary key on a table similar to the driver's license example mentioned earlier. Because we expect the driver's license IDs to always be unique, we'll use that column as a natural key.

```
CREATE TABLE natural_key_example (
 1 license_id text CONSTRAINT license_key PRIMARY KEY,
    first_name text,
    last_name text
);

2 DROP TABLE natural_key_example;

CREATE TABLE natural_key_example (
    license_id text,
    first_name text,
    last_name text,
 3 CONSTRAINT license_key PRIMARY KEY (license_id)
);
```

Listing 8-1: Declaring a single-column natural key as a primary key

We first create a table called `natural_key_example` and use the column constraint syntax `CONSTRAINT` to declare `license_id` as the primary key 1 followed by a name for the constraint and the keywords `PRIMARY KEY`. This syntax makes it easy to understand at a glance which column is designated as the primary key. Note that you can omit the `CONSTRAINT` keyword and name for the key and simply use `PRIMARY KEY`:

```
license_id text PRIMARY KEY
```

In that case, PostgreSQL will name the primary key on its own, using the convention of the table name followed by `_pkey`.

Next, we delete the table from the database with `DROP TABLE` 2 to prepare for the table constraint example.

To add a table constraint, we declare the `CONSTRAINT` after listing all the columns 3, with the column we want to use as the key in parentheses. (Again, you can omit the `CONSTRAINT` keyword and key name.) In this example, we end up with the same `license_id` column for the primary key. You must use the table constraint syntax when you want to create a primary key using more than one column; in that case, you would list the columns in parentheses, separated by commas. We'll explore that in a moment.

First, let's look at how the qualities of a primary key—unique for every row and no `NULL` values—protect you from harming your data's integrity. Listing 8-2 has two `INSERT` statements.

```
INSERT INTO natural_key_example (license_id, first_name,
last_name)
VALUES ('T229901', 'Gem', 'Godfrey');

INSERT INTO natural_key_example (license_id, first_name,
last_name)
VALUES ('T229901', 'John', 'Mitchell');
```

Listing 8-2: An example of a primary key violation

When you execute the first `INSERT` statement on its own, the server loads a row into the `natural_key_example` table without any issue. When you attempt to execute the second, the server replies with an error:

```
ERROR:  duplicate key value violates unique constraint
"license_key"
DETAIL:  Key (license_id)=(T229901) already exists.
```

Before adding the row, the server checked whether a `license_id` of `T229901` was already present in the table. Because it was and because a primary key by definition must be unique for each row, the server rejected the operation. The rules of the fictional DMV state that no two drivers can have the same license ID, so checking for and rejecting duplicate data is one way for the database to enforce that rule.

Creating a Composite Primary Key

If a single column doesn't meet the requirements for a primary key, we can create a *composite primary key*.

We'll make a table that tracks student school attendance. The combination of `student_id` and `school_day` columns gives us a unique value for each row, which records whether a student was in school on that day in a column called `present`. To create a composite primary key, you must declare it using the table constraint syntax, as shown in Listing 8-3.

```
CREATE TABLE natural_key_composite_example (
    student_id text,
    school_day date,
    present boolean,
    CONSTRAINT student_key PRIMARY KEY (student_id,
school_day)
);
```

Listing 8-3: Declaring a composite primary key as a natural key

Here we pass two (or more) columns as arguments rather than one. We'll simulate a key violation by attempting to insert a row where the combination of values in the two key columns—`student_id` and `school_day`—is not unique to the table. Run the `INSERT` statements in Listing 8-4 one at a time (by highlighting them in pgAdmin before clicking **Execute/Refresh**).

```
INSERT INTO natural_key_composite_example (student_id,
school_day, present)
VALUES(775, '2022-01-22', 'Y');

INSERT INTO natural_key_composite_example (student_id,
school_day, present)
VALUES(775, '2022-01-23', 'Y');

INSERT INTO natural_key_composite_example (student_id,
school_day, present)
VALUES(775, '2022-01-23', 'N');
```

Listing 8-4: Example of a composite primary key violation

The first two `INSERT` statements execute fine because there's no duplication of values in the combination of the key columns. But the third statement causes an error because the `student_id` and `school_day` values it contains match a combination that already exists in the table:

```
ERROR:  duplicate key value violates unique constraint
"student_key"
DETAIL:  Key (student_id, school_day)=(775, 2022-01-23)
already exists.
```

You can create composite keys with more than two columns. The limit to the number of columns you can use depends on your database.

Creating an Auto-Incrementing Surrogate Key

As you learned in "Auto-Incrementing Integers" in Chapter 4, there are two ways to have a PostgreSQL database add an automatically increasing unique value to a column. The first is to set the column to one of the PostgreSQL-specific serial data types: `smallserial`, `serial`, and `bigserial`. The second is to use the `IDENTITY` syntax; because it is part of the ANSI SQL standard, we'll employ this for our examples.

Use `IDENTITY` with one of the integer types `smallint`, `integer`, and `bigint`. For a primary key, it may be tempting to try to save disk space by using `integer`, which handles numbers as large as 2,147,483,647. But many a database developer has received a late-night call from a user frantic to know why an application is broken, only to discover that the database is trying to generate a number one greater than the data type's maximum. So, if it's remotely possible that your table will grow past 2.147 billion rows, it's wise to use `bigint`, which accepts numbers as high as 9.2 *quintillion*. You can set it and forget it, as shown in the first column defined in Listing 8-5.

```
CREATE TABLE surrogate_key_example (
  1 order_number bigint GENERATED ALWAYS AS IDENTITY,
    product_name text,
    order_time timestamp with time zone,
  2 CONSTRAINT order_number_key PRIMARY KEY (order_number)
);

3 INSERT INTO surrogate_key_example (product_name, order_time)
  VALUES ('Beachball Polish', '2020-03-15 09:21-07'),
         ('Wrinkle De-Atomizer', '2017-05-22 14:00-07'),
         ('Flux Capacitor', '1985-10-26 01:18:00-07');

  SELECT * FROM surrogate_key_example;
```

Listing 8-5: Declaring a `bigint` column as a surrogate key using `IDENTITY`

Listing 8-5 shows how to declare an auto-incrementing `bigint` 1 column called `order_number` using the `IDENTITY` syntax and then set the column as the primary key 2. When you insert data into the table 3, you omit `order_number` from the list of columns and values. The database will create a new value for that column as each row is inserted, and that value will be one greater than the largest already created for the column.

Run `SELECT * FROM surrogate_key_example;` to see how the column fills in automatically:

```
order_number    product_name             order_time
------------ ------------------- ----------------------
           1 Beachball Polish    2020-03-15 09:21:00-07
           2 Wrinkle De-Atomizer 2017-05-22 14:00:00-07
           3 Flux Capacitor      1985-10-26 01:18:00-07
```

We see these sorts of auto-incrementing order numbers reflected in the receipts for the purchases we make every day. Now you know how it's done.

NOTE

In your query results, the timestamps in the `order_time` *column will vary based on the time zone configuration of your PostgreSQL installation, as discussed in Chapter 4.*

A few details worth noting: if you delete a row, the database won't fill the gap in the `order_number` sequence, nor will it change any of the existing values in that column. It will generally add one to the largest existing value in the sequence (though there are exceptions related to operations, including restoring a database from a backup). Also, we used the syntax `GENERATED ALWAYS AS IDENTITY`. As discussed in Chapter 4, this prevents a user from inserting a value in `order_number` without manually overriding the setting. Generally, you want to prevent such meddling to avoid problems. Let's say a user were to manually insert a value of `4` into the `order_number` column of your existing `surrogate_key_example` table. That manual insert will not increment the `IDENTITY` sequence for the `order_number` column; that occurs only when the database generates a new value. Thus, on the next row insert, the database also would try to also insert a `4`, as that's the next number in the sequence. The result will be an error, because a duplicate value violates the primary key constraint.

You can, however, allow manual insertions by restarting the `IDENTITY` sequence. You might allow this in case you need to insert a row that was mistakenly deleted. Listing 8-6 shows how to add a row to the table that has an `order_number` of `4`, which is the next value in the sequence.

```
 INSERT INTO surrogate_key_example
1 OVERRIDING SYSTEM VALUE
 VALUES (4, 'Chicken Coop', '2021-09-03 10:33-07');

2 ALTER TABLE surrogate_key_example ALTER COLUMN order_number
 RESTART WITH 5;

3 INSERT INTO surrogate_key_example (product_name, order_time)
 VALUES ('Aloe Plant', '2020-03-15 10:09-07');
```

Listing 8-6: Restarting an `IDENTITY` sequence

You start with an `INSERT` statement that includes the keywords `OVERRIDING SYSTEM VALUE` 1. Next we include the `VALUES` clause and specify the integer `4` for the first column, `order_number`, in the `VALUES` list, which overrides the `IDENTITY` restriction. We're using `4`, but we could choose any number that's not already present in the column.

After the insert, you need to reset the `IDENTITY` sequence so that it begins at a number larger than the `4` you just inserted. To do this, use an `ALTER TABLE ... ALTER COLUMN` statement 2 that includes the keywords `RESTART WITH 5`. An `ALTER TABLE` modifies tables and columns in various ways, which we'll explore more thoroughly in Chapter 10, "Inspecting and Modifying Data." Here, you use it to change the beginning number of the `IDENTITY` sequence; so, when the next row gets added to the table, the value for `order_number` will be `5`. Finally, insert a new row 3 and omit a value for the `order_number`, as you did in Listing 8-5.

If you select all rows again from the `surrogate_key_example` table, you'll see that the `order_number` column populated as intended:

```
order_number    product_name           order_time
------------ -------------------- ----------------------
           1 Beachball Polish     2020-03-15 09:21:00-07
           2 Wrinkle De-Atomizer  2017-05-22 14:00:00-07
           3 Flux Capacitor       1985-10-26 01:18:00-07
           4 Chicken Coop         2021-09-03 10:33:00-07
           5 Aloe Plant           2020-03-15 10:09:00-07
```

This task isn't one you necessarily want to tackle often, but it's good to know if the need arises.

Foreign Keys

We use *foreign keys* to establish relationships between tables. A foreign key is one or more columns whose values match those in another table's primary key or other unique key. Foreign key values must already exist in the primary key or other unique key of the table it references. If not, the value is rejected. With this constraint, SQL enforces *referential integrity*—ensuring that data in related tables doesn't end up unrelated, or orphaned.

We won't end up with rows in one table that have no relation to rows in the other tables we can join them to.

Listing 8-7 shows two tables from a hypothetical database tracking motor vehicle activity.

```
CREATE TABLE licenses (
    license_id text,
    first_name text,
    last_name text,
 1 CONSTRAINT licenses_key PRIMARY KEY (license_id)
);

CREATE TABLE registrations (
    registration_id text,
    registration_date timestamp with time zone,
 2 license_id text REFERENCES licenses (license_id),
    CONSTRAINT registration_key PRIMARY KEY (registration_id,
license_id)
);

3 INSERT INTO licenses (license_id, first_name, last_name)
  VALUES ('T229901', 'Steve', 'Rothery');

4 INSERT INTO registrations (registration_id, registration_date,
  license_id)
  VALUES ('A203391', '2022-03-17', 'T229901');

5 INSERT INTO registrations (registration_id, registration_date,
  license_id)
  VALUES ('A75772', '2022-03-17', 'T000001');
```

Listing 8-7: *A foreign key example*

The first table, `licenses`, uses a driver's unique `license_id` 1 as a natural primary key. The second table, `registrations`, is for tracking vehicle registrations. A single license ID might be connected to multiple vehicle registrations, because each licensed driver can register multiple vehicles—this is called a *one-to-many relationship* (Chapter 7).

Here's how that relationship is expressed via SQL: in the `registrations` table, we designate the column `license_id` 2 as a foreign key by adding the

REFERENCES keyword, followed by the table name and column for it to reference.

Now, when we insert a row into registrations, the database will test whether the value inserted into license_id already exists in the license_id primary key column of the licenses table. If it doesn't, the database returns an error, which is important. If any rows in registrations didn't correspond to a row in licenses, we'd have no way to write a query to find the person who registered the vehicle.

To see this constraint in action, create the two tables and execute the INSERT statements one at a time. The first adds a row to licenses 3 that includes the value T229901 for the license_id. The second adds a row to registrations 4 where the foreign key contains the same value. So far, so good, because the value exists in both tables. But we encounter an error with the third insert, which tries to add a row to registrations 5 with a value for license_id that's not in licenses:

```
ERROR:  insert or update on table "registrations" violates
foreign key constraint "registrations_license_id_fkey"
DETAIL:  Key (license_id)=(T000001) is not present in table
"licenses".
```

The resulting error is actually helpful: the database is enforcing referential integrity by preventing a registration for a nonexistent license holder. But it also indicates a few practical implications. First, it affects the order in which we insert data. We cannot add data to a table that contains a foreign key before the other table referenced by the key has the related records, or we'll get an error. In this example, we'd have to create a driver's license record before inserting a related registration record (if you think about it, that's what your local department of motor vehicles probably does).

Second, the reverse applies when we delete data. To maintain referential integrity, the foreign key constraint prevents us from deleting a row from licenses before removing any related rows in registrations, because doing so would leave an orphaned record. We would have to delete the related row in registrations first and then delete the row in licenses.

However, ANSI SQL provides a way to handle this order of operations automatically using the `ON DELETE CASCADE` keywords.

How to Automatically Delete Related Records with CASCADE

To delete a row in `licenses` and have that action automatically delete any related rows in `registrations`, we can specify that behavior by adding `ON DELETE CASCADE` when defining the foreign key constraint.

Here's how we would modify the *Listing 8-7* `CREATE TABLE` statement for `registrations`, adding the keywords at the end of the definition of the `license_id` column:

```
CREATE TABLE registrations (
    registration_id text,
    registration_date date,
    license_id text REFERENCES licenses (license_id) ON
DELETE CASCADE,
    CONSTRAINT registration_key PRIMARY KEY (registration_id,
license_id)
);
```

Deleting a row in `licenses` should also delete all related rows in `registrations`. This allows us to delete a driver's license without first having to manually remove any registrations linked to it. It also maintains data integrity by ensuring deleting a license doesn't leave orphaned rows in `registrations`.

The CHECK Constraint

A `CHECK` constraint evaluates whether data added to a column meets the expected criteria, which we specify with a logical test. If the criteria aren't met, the database returns an error. The `CHECK` constraint is extremely valuable because it can prevent columns from getting loaded with nonsensical data. For example, a baseball player's total number of hits shouldn't be negative, so you should limit that data to values of zero or greater. Or, in most schools, `Z` isn't a valid letter grade for a course (although my barely passing algebra grade felt like it), so we might insert constraints that only accept the values A–F.

As with primary keys, we can implement a `CHECK` constraint at the column or table level. For a column constraint, declare it in the `CREATE TABLE` statement after the column name and data type: `CHECK (`*`logical expression`*`)`. As a table constraint, use the syntax `CONSTRAINT` *`constraint_name`* `CHECK (`*`logical expression`*`)` after all columns are defined.

Listing 8-8 shows a `CHECK` constraint applied to two columns in a table we might use to track the user role and salary of employees within an organization. It uses the table constraint syntax for the primary key and the `CHECK` constraint.

```
CREATE TABLE check_constraint_example (
    user_id bigint GENERATED ALWAYS AS IDENTITY,
    user_role text,
    salary numeric(10,2),
    CONSTRAINT user_id_key PRIMARY KEY (user_id),
 1 CONSTRAINT check_role_in_list CHECK (user_role IN('Admin',
'Staff')),
 2 CONSTRAINT check_salary_not_below_zero CHECK (salary >= 0)
);
```

Listing 8-8: *Examples of `CHECK` constraints*

We create the table and set the `user_id` column as an auto-incrementing surrogate primary key. The first `CHECK` 1 tests whether values entered into the `user_role` column match one of two predefined strings, `Admin` or `Staff`, by using the SQL `IN` operator. The second `CHECK` 2 tests whether values entered in the `salary` column are greater than or equal to 0, because a negative amount wouldn't make sense. Both tests are an example of a *Boolean expression*, a statement that evaluates as either true or false. If a value tested by the constraint evaluates as `true`, the check passes.

NOTE

Developers may differ on whether check logic belongs in the database, in the application in front of the database, such as a human resources system, or both. One advantage of checks in the database is that the database will maintain data integrity in the case of changes to the application, new applications that use the database, or users directly accessing the database.

When values are inserted or updated, the database checks them against the constraint. If the values in either column violate the constraint—or, for that matter, if the primary key constraint is violated—the database will reject the change.

If we use the table constraint syntax, we also can combine more than one test in a single `CHECK` statement. Say we have a table related to student achievement. We could add the following:

```
CONSTRAINT grad_check CHECK (credits >= 120 AND tuition =
'Paid')
```

Notice that we combine two logical tests by enclosing them in parentheses and connecting them with `AND`. Here, both Boolean expressions must evaluate as `true` for the entire check to pass. You can also test values across columns, as in the following example where we want to make sure an item's sale price is a discount on the original, assuming we have columns for both values:

```
CONSTRAINT sale_check CHECK (sale_price < retail_price)
```

Inside the parentheses, the logical expression checks that the sale price is less than the retail price.

The UNIQUE Constraint

We can also ensure that a column has a unique value in each row by using the `UNIQUE` constraint. If ensuring unique values sounds similar to the purpose of a primary key, it is. But `UNIQUE` has one important difference. In a primary key, no values can be `NULL`, but a `UNIQUE` constraint permits multiple `NULL` values in a column. This is useful in cases where we won't always have values but want to ensure that the ones we do have are unique.

To show the usefulness of `UNIQUE`, look at the code in *Listing 8-9*, which is a table for tracking contact info.

```
CREATE TABLE unique_constraint_example (
    contact_id bigint GENERATED ALWAYS AS IDENTITY,
    first_name text,
    last_name text,
    email text,
    CONSTRAINT contact_id_key PRIMARY KEY (contact_id),
 1 CONSTRAINT email_unique UNIQUE (email)
);

INSERT INTO unique_constraint_example (first_name, last_name,
email)
VALUES ('Samantha', 'Lee', 'slee@example.org');

INSERT INTO unique_constraint_example (first_name, last_name,
email)
VALUES ('Betty', 'Diaz', 'bdiaz@example.org');

INSERT INTO unique_constraint_example (first_name, last_name,
email)
2 VALUES ('Sasha', 'Lee', 'slee@example.org');
```

Listing 8-9: A `UNIQUE` constraint example

In this table, `contact_id` serves as a surrogate primary key, uniquely identifying each row. But we also have an `email` column, the main point of contact with each person. We'd expect this column to contain only unique email addresses, but those addresses might change over time. So, we use `UNIQUE` 1 to ensure that any time we add or update a contact's email, we're not providing one that already exists. If we try to insert an email that already exists 2, the database will return an error:

```
ERROR:  duplicate key value violates unique constraint
"email_unique"
DETAIL:  Key (email)=(slee@example.org) already exists.
```

Again, the error shows the database is working for us.

The NOT NULL Constraint

In Chapter 7, you learned about `NULL`, a special SQL value that represents missing data or unknown values. We know that `NULL` is not allowed for primary key values because they need to uniquely identify each row in a table. But there may be other times when you'll want to disallow empty values in a column. For example, in a table listing each student in a school, requiring that columns containing first and last names be filled for each row makes sense. To require a value in a column, SQL provides the `NOT NULL` constraint, which simply prevents a column from accepting empty values.

Listing 8-10 demonstrates the `NOT NULL` syntax.

```
CREATE TABLE not_null_example (
    student_id bigint GENERATED ALWAYS AS IDENTITY,
    first_name text NOT NULL,
    last_name text NOT NULL,
    CONSTRAINT student_id_key PRIMARY KEY (student_id)
);
```

Listing 8-10: A `NOT NULL` constraint example

Here, we declare `NOT NULL` for the `first_name` and `last_name` columns because it's likely we'd require those pieces of information in a table tracking student information. If we attempt an `INSERT` on the table and don't include values for those columns, the database will notify us of the violation.

How to Remove Constraints or Add Them Later

You can remove a constraint or later add one to an existing table using `ALTER TABLE`, the command you used earlier in the chapter in "Creating an Auto-incrementing Surrogate Key" to reset the `IDENTITY` sequence.

To remove a primary key, foreign key, or `UNIQUE` constraint, you write an `ALTER TABLE` statement in this format:

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name;
```

To drop a `NOT NULL` constraint, the statement operates on the column, so you must use the additional `ALTER COLUMN` keywords, like so:

```
ALTER TABLE table_name ALTER COLUMN column_name DROP NOT
NULL;
```

Let's use these statements to modify the `not_null_example` table you just made, as shown in Listing 8-11.

```
ALTER TABLE not_null_example DROP CONSTRAINT student_id_key;
ALTER TABLE not_null_example ADD CONSTRAINT student_id_key
PRIMARY KEY (student_id);
ALTER TABLE not_null_example ALTER COLUMN first_name DROP NOT
NULL;
ALTER TABLE not_null_example ALTER COLUMN first_name SET NOT
NULL;
```

Listing 8-11: Dropping and adding a primary key and a `NOT NULL` constraint

Execute the statements one at a time. Each time, you can view the changes to the table definition in pgAdmin by clicking the table name once and then clicking the **SQL** tab above the query window. (Note that it will display a more verbose syntax for the table definition than what you used when creating the table.)

With the first `ALTER TABLE` statement, we use `DROP CONSTRAINT` to remove the primary key named `student_id_key`. We then add the primary key back using `ADD CONSTRAINT`. We'd use that same syntax to add a constraint to any existing table.

NOTE

You can add a constraint to an existing table only if the data in the target column obeys the limits of the constraint. For example, you can't place a primary key constraint on a column that has duplicate or empty values.

In the third statement, `ALTER COLUMN` and `DROP NOT NULL` remove the `NOT NULL` constraint from the `first_name` column. Finally, `SET NOT NULL` adds the constraint.

Speeding Up Queries with Indexes

In the same way that a book's index helps you find information more quickly, you can speed up queries by adding an *index*—a separate data structure the database manages—to one or more columns in a table. The database uses the index as a shortcut rather than scanning each row to find data. That's admittedly a simplistic picture of what, in SQL databases, is a nontrivial topic. We could spend several chapters delving into the workings of SQL indexes and tuning databases for performance, but instead I'll offer general guidance on using indexes and a PostgreSQL-specific example that demonstrates their benefits.

NOTE

The ANSI SQL standard doesn't specify a syntax for creating indexes, nor does it specify how a database system should implement them. Nevertheless, indexes are a feature of all major database systems, including Microsoft SQL Server, MySQL, Oracle, and SQLite, with similarities to the syntax and behavior described here.

B-Tree: PostgreSQL's Default Index

You've already created several indexes, perhaps without knowing. Each time you add a primary key or `UNIQUE` constraint, PostgreSQL (as well as most database systems) creates an index on the column or columns included in the constraint. Indexes are stored separately from the table data and are accessed automatically (if needed) when you run a query and updated every time a row is added, removed, or updated.

In PostgreSQL, the default index type is the *B-tree index*. It's created automatically on the columns designated for the primary key or a `UNIQUE` constraint, and it's also the type created by default with the `CREATE INDEX` statement. B-tree, short for *balanced tree*, is so named because when you search for a value, the structure looks from the top of the tree down through branches until it locates the value. (Of course, the process is a lot more complicated than that.) A B-tree index is useful for data that can be ordered and searched using equality and range operators, such as `<`, `<=`, `=`, `>=`, `>`, and `BETWEEN`. It also works with `LIKE` if there's no wildcard in the pattern at the beginning of the search string. An example is `WHERE chips LIKE 'Dorito%'`.

PostgreSQL also supports additional index types, such as the *Generalized Inverted Index (GIN)* and the *Generalized Search Tree (GiST)*. Each has distinct uses, and I'll incorporate them in later chapters on full-text search and queries using geometry types.

For now, let's see a B-tree index speed up a simple search query. For this exercise, we'll use a large dataset comprising more than 900,000 New York City street addresses, compiled by the OpenAddresses project at *https://openaddresses.io/*. The file with the data, *city_of_new_york.csv*, is available for you to download along with all the resources for this book from *https://nostarch.com/practical-sql-2nd-edition/*.

After you've downloaded the file, use the code in *Listing 8-12* to create a `new_york_addresses` table and import the address data. The import will take longer than the tiny datasets you've loaded so far because the CSV file is about 50MB.

```
CREATE TABLE new_york_addresses (
    longitude numeric(9,6),
    latitude numeric(9,6),
    street_number text,
    street text,
    unit text,
    postcode text,
    id integer CONSTRAINT new_york_key PRIMARY KEY
);

COPY new_york_addresses
FROM 'C:\YourDirectory\city_of_new_york.csv'
WITH (FORMAT CSV, HEADER);
```

Listing 8-12: Importing New York City address data

When the data loads, run a quick `SELECT` query to visually check that you have 940,374 rows and seven columns. A common use for this data might be to search for matches in the `street` column, so we'll use that example for exploring index performance.

Benchmarking Query Performance with EXPLAIN

We'll measure the performance before and after adding an index by using the PostgreSQL-specific `EXPLAIN` command, which lists the *query plan* for a specific database query. The query plan might include how the database plans to scan the table, whether or not it will use indexes, and so on. When we add the `ANALYZE` keyword, `EXPLAIN` will carry out the query and show the actual execution time.

Recording Some Control Execution Times

We'll use the three queries in *Listing 8-13* to analyze query performance before and after adding an index. We're using typical `SELECT` queries with a `WHERE` clause with `EXPLAIN ANALYZE` included at the beginning. These keywords tell the database to execute the query and display statistics about the query process and how long it took to execute, rather than show the results.

```
EXPLAIN ANALYZE SELECT * FROM new_york_addresses
WHERE street = 'BROADWAY';

EXPLAIN ANALYZE SELECT * FROM new_york_addresses
WHERE street = '52 STREET';

EXPLAIN ANALYZE SELECT * FROM new_york_addresses
WHERE street = 'ZWICKY AVENUE';
```

Listing 8-13: Benchmark queries for index performance

On my system, the first query returns these stats in the pgAdmin output pane:

```
Gather (cost=1000.00..15184.08 rows=3103 width=46) (actual
time=9.000..388.448 rows=3336 loops=1)
  Workers Planned: 2
  Workers Launched: 2
```