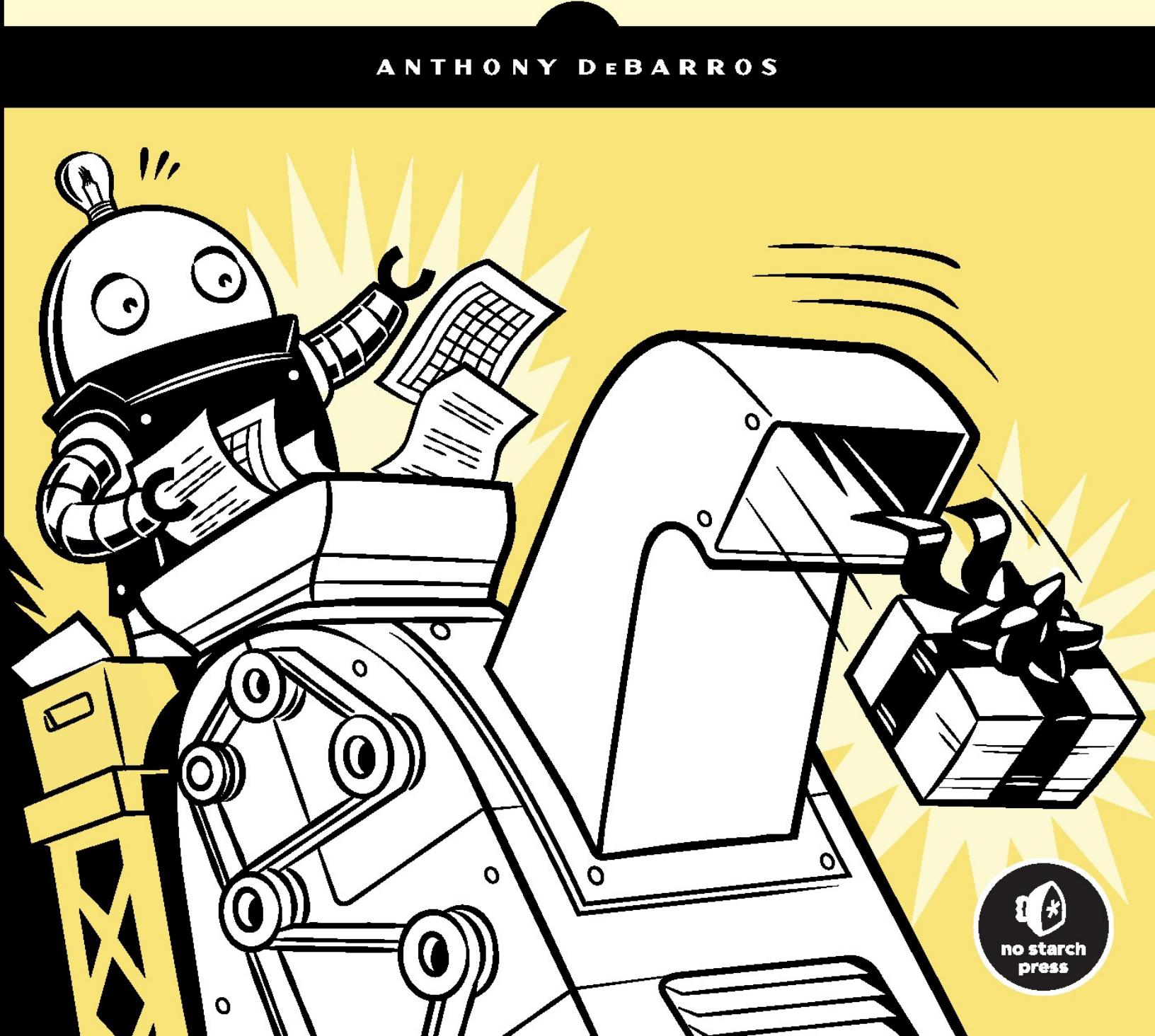


SECOND EDITION

PRACTICAL SQL

A BEGINNER'S GUIDE TO
STORYTELLING WITH DATA

ANTHONY DEBARROS



CONTENTS IN DETAIL

[TITLE PAGE](#)

[COPYRIGHT](#)

[ABOUT THE AUTHOR](#)

[PREFACE TO THE SECOND EDITION](#)

[ACKNOWLEDGMENTS](#)

[INTRODUCTION](#)

[What Is SQL?](#)

[Why SQL?](#)

[Who Is This Book For?](#)

[What You'll Learn](#)

[CHAPTER 1: SETTING UP YOUR CODING ENVIRONMENT](#)

[Installing a Text Editor](#)

[Downloading Code and Data from GitHub](#)

[Installing PostgreSQL and pgAdmin](#)

[Windows Installation](#)

[macOS Installation](#)

[Linux Installation](#)

[Working with pgAdmin](#)

[Launching pgAdmin and Setting a Master Password](#)

[Connecting to the Default postgres Database](#)

[Exploring the Query Tool](#)

[Customizing pgAdmin](#)

[Alternatives to pgAdmin](#)

[Wrapping Up](#)

CHAPTER 2: CREATING YOUR FIRST DATABASE AND TABLE

[Understanding Tables](#)

[Creating a Database](#)

[Executing SQL in pgAdmin](#)

[Connecting to the analysis Database](#)

[Creating a Table](#)

[Using the CREATE TABLE Statement](#)

[Making the teachers Table](#)

[Inserting Rows into a Table](#)

[Using the INSERT Statement](#)

[Viewing the Data](#)

[Getting Help When Code Goes Bad](#)

[Formatting SQL for Readability](#)

[Wrapping Up](#)

CHAPTER 3: BEGINNING DATA EXPLORATION WITH SELECT

[Basic SELECT Syntax](#)

[Querying a Subset of Columns](#)

[Sorting Data with ORDER BY](#)

[Using DISTINCT to Find Unique Values](#)

[Filtering Rows with WHERE](#)

[Using LIKE and ILIKE with WHERE](#)

[Combining Operators with AND and OR](#)

[Putting It All Together](#)

[Wrapping Up](#)

CHAPTER 4: UNDERSTANDING DATA TYPES

[Understanding Characters](#)

[Understanding Numbers](#)

[Using Integers](#)

[Auto-Incrementing Integers](#)

[Using Decimal Numbers](#)

[Choosing Your Number Data Type](#)

[Understanding Dates and Times](#)

[Using the interval Data Type in Calculations](#)

[Understanding JSON and JSONB](#)

[Using Miscellaneous Types](#)

[Transforming Values from One Type to Another with CAST](#)

[Using CAST Shortcut Notation](#)

[Wrapping Up](#)

CHAPTER 5: IMPORTING AND EXPORTING DATA

[Working with Delimited Text Files](#)

[Handling Header Rows](#)

[Quoting Columns That Contain Delimiters](#)

[Using COPY to Import Data](#)

[Importing Census Data Describing Counties](#)

[Creating the us_counties_pop_est_2019 Table](#)

[Understanding Census Columns and Data Types](#)

[Performing the Census Import with COPY](#)

[Inspecting the Import](#)

[Importing a Subset of Columns with COPY](#)

[Importing a Subset of Rows with COPY](#)

[Adding a Value to a Column During Import](#)

[Using COPY to Export Data](#)

[Exporting All Data](#)

[Exporting Particular Columns](#)

[Exporting Query Results](#)

[Importing and Exporting Through pgAdmin](#)

[Wrapping Up](#)

CHAPTER 6: BASIC MATH AND STATS WITH SQL

[Understanding Math Operators and Functions](#)

[Understanding Math and Data Types](#)

[Adding, Subtracting, and Multiplying](#)

[Performing Division and Modulo](#)

[Using Exponents, Roots, and Factorials](#)

[Minding the Order of Operations](#)

[Doing Math Across Census Table Columns](#)

[Adding and Subtracting Columns](#)

[Finding Percentages of the Whole](#)

[Tracking Percent Change](#)

[Using Aggregate Functions for Averages and Sums](#)

[Finding the Median](#)

[Finding the Median with Percentile Functions](#)

[Finding Median and Percentiles with Census Data](#)

[Finding Other Quantiles with Percentile Functions](#)

[Finding the Mode](#)

[Wrapping Up](#)

CHAPTER 7: JOINING TABLES IN A RELATIONAL DATABASE

[Linking Tables Using JOIN](#)

[Relating Tables with Key Columns](#)

[Querying Multiple Tables Using JOIN](#)

[Understanding JOIN Types](#)

[JOIN](#)

[LEFT JOIN and RIGHT JOIN](#)

[FULL OUTER JOIN](#)

[CROSS JOIN](#)

[Using NULL to Find Rows with Missing Values](#)

[Understanding the Three Types of Table Relationships](#)

[One-to-One Relationship](#)

[One-to-Many Relationship](#)

[Many-to-Many Relationship](#)

[Selecting Specific Columns in a Join](#)

[Simplifying JOIN Syntax with Table Aliases](#)

[Joining Multiple Tables](#)

[Combining Query Results with Set Operators](#)

[UNION and UNION ALL](#)

[INTERSECT and EXCEPT](#)

[Performing Math on Joined Table Columns](#)

[Wrapping Up](#)

CHAPTER 8: TABLE DESIGN THAT WORKS FOR YOU

[Following Naming Conventions](#)

[Quoting Identifiers Enables Mixed Case](#)

[Pitfalls with Quoting Identifiers](#)

[Guidelines for Naming Identifiers](#)

[Controlling Column Values with Constraints](#)

[Primary Keys: Natural vs. Surrogate](#)

[Foreign Keys](#)

[How to Automatically Delete Related Records with CASCADE](#)

[The CHECK Constraint](#)

[The UNIQUE Constraint](#)

[The NOT NULL Constraint](#)

[How to Remove Constraints or Add Them Later](#)

[Speeding Up Queries with Indexes](#)

[B-Tree: PostgreSQL's Default Index](#)

[Considerations When Using Indexes](#)

[Wrapping Up](#)

CHAPTER 9: EXTRACTING INFORMATION BY GROUPING AND SUMMARIZING

[Creating the Library Survey Tables](#)

[Creating the 2018 Library Data Table](#)

[Creating the 2017 and 2016 Library Data Tables](#)

[Exploring the Library Data Using Aggregate Functions](#)

[Counting Rows and Values Using count\(\)](#)

[Finding Maximum and Minimum Values Using max\(\) and min\(\)](#)

[Aggregating Data Using GROUP BY](#)

[Wrapping Up](#)

CHAPTER 10: INSPECTING AND MODIFYING DATA

[Importing Data on Meat, Poultry, and Egg Producers](#)

[Interviewing the Dataset](#)

[Checking for Missing Values](#)

[Checking for Inconsistent Data Values](#)

[Checking for Malformed Values Using length\(\)](#)

[Modifying Tables, Columns, and Data](#)

[Modifying Tables with ALTER TABLE](#)

[Modifying Values with UPDATE](#)

[Viewing Modified Data with RETURNING](#)

[Creating Backup Tables](#)

[Restoring Missing Column Values](#)

[Updating Values for Consistency](#)

[Repairing ZIP Codes Using Concatenation](#)

[Updating Values Across Tables](#)

[Deleting Unneeded Data](#)

[Deleting Rows from a Table](#)

[Deleting a Column from a Table](#)

[Deleting a Table from a Database](#)

[Using Transactions to Save or Revert Changes](#)

[Improving Performance When Updating Large Tables](#)

[Wrapping Up](#)

[CHAPTER 11: STATISTICAL FUNCTIONS IN SQL](#)

[Creating a Census Stats Table](#)

[Measuring Correlation with corr\(Y, X\)](#)

[Checking Additional Correlations](#)

[Predicting Values with Regression Analysis](#)

[Finding the Effect of an Independent Variable with r-Squared](#)

[Finding Variance and Standard Deviation](#)

[Creating Rankings with SQL](#)

[Ranking with rank\(\) and dense_rank\(\)](#)

[Ranking Within Subgroups with PARTITION BY](#)

[Calculating Rates for Meaningful Comparisons](#)

[Finding Rates of Tourism-Related Businesses](#)

[Smoothing Uneven Data](#)

[Wrapping Up](#)

CHAPTER 12: WORKING WITH DATES AND TIMES

[Understanding Data Types and Functions for Dates and Times](#)

[Manipulating Dates and Times](#)

[Extracting the Components of a timestamp Value](#)

[Creating Datetime Values from timestamp Components](#)

[Retrieving the Current Date and Time](#)

[Working with Time Zones](#)

[Finding Your Time Zone Setting](#)

[Setting the Time Zone](#)

[Performing Calculations with Dates and Times](#)

[Finding Patterns in New York City Taxi Data](#)

[Finding Patterns in Amtrak Data](#)

[Wrapping Up](#)

CHAPTER 13: ADVANCED QUERY TECHNIQUES

[Using Subqueries](#)

[Filtering with Subqueries in a WHERE Clause](#)

[Creating Derived Tables with Subqueries](#)

[Joining Derived Tables](#)

[Generating Columns with Subqueries](#)

[Understanding Subquery Expressions](#)

[Using Subqueries with LATERAL](#)

[Using Common Table Expressions](#)

[Performing Cross Tabulations](#)

[Installing the crosstab\(\) Function](#)

[Tabulating Survey Results](#)

[Tabulating City Temperature Readings](#)

[Reclassifying Values with CASE](#)

[Using CASE in a Common Table Expression](#)

[Wrapping Up](#)

CHAPTER 14: MINING TEXT TO FIND MEANINGFUL DATA

[Formatting Text Using String Functions](#)

[Case Formatting](#)

[Character Information](#)

[Removing Characters](#)

[Extracting and Replacing Characters](#)

[Matching Text Patterns with Regular Expressions](#)

[Regular Expression Notation](#)

[Using Regular Expressions with WHERE](#)

[Regular Expression Functions to Replace or Split Text](#)

[Turning Text to Data with Regular Expression Functions](#)

[Full-Text Search in PostgreSQL](#)

[Text Search Data Types](#)

[Creating a Table for Full-Text Search](#)

[Searching Speech Text](#)

[Ranking Query Matches by Relevance](#)

[Wrapping Up](#)

CHAPTER 15: ANALYZING SPATIAL DATA WITH POSTGIS

[Enabling PostGIS and Creating a Spatial Database](#)

[Understanding the Building Blocks of Spatial Data](#)

[Understanding Two-Dimensional Geometries](#)

[Well-Known Text Formats](#)

[Projections and Coordinate Systems](#)

[Spatial Reference System Identifier](#)
[Understanding PostGIS Data Types](#)
[Creating Spatial Objects with PostGIS Functions](#)
[Creating a Geometry Type from Well-Known Text](#)
[Creating a Geography Type from Well-Known Text](#)
[Using Point Functions](#)
[Using LineString Functions](#)
[Using Polygon Functions](#)
[Analyzing Farmers' Markets Data](#)
[Creating and Filling a Geography Column](#)
[Adding a Spatial Index](#)
[Finding Geographies Within a Given Distance](#)
[Finding the Distance Between Geographies](#)
[Finding the Nearest Geographies](#)
[Working with Census Shapefiles](#)
[Understanding the Contents of a Shapefile](#)
[Loading Shapefiles](#)
[Exploring the Census 2019 Counties Shapefile](#)
[Examining Demographics Within a Distance](#)
[Performing Spatial Joins](#)
[Exploring Roads and Waterways Data](#)
[Joining the Census Roads and Water Tables](#)
[Finding the Location Where Objects Intersect](#)
[Wrapping Up](#)

CHAPTER 16: WORKING WITH JSON DATA

[Understanding JSON Structure](#)
[Considering When to Use JSON with SQL](#)

[Using json and jsonb Data Types](#)
[Importing and Indexing JSON Data](#)
[Using json and jsonb Extraction Operators](#)
[Key Value Extraction](#)
[Array Element Extraction](#)
[Path Extraction](#)
[Containment and Existence](#)
[Analyzing Earthquake Data](#)
[Exploring and Loading the Earthquake Data](#)
[Working with Earthquake Times](#)
[Finding the Largest and Most-Reported Earthquakes](#)
[Converting Earthquake JSON to Spatial Data](#)
[Generating and Manipulating JSON](#)
[Turning Query Results into JSON](#)
[Adding, Updating, and Deleting Keys and Values](#)
[Using JSON Processing Functions](#)
[Finding the Length of an Array](#)
[Returning Array Elements as Rows](#)
[Wrapping Up](#)

CHAPTER 17: SAVING TIME WITH VIEWS, FUNCTIONS, AND TRIGGERS

[Using Views to Simplify Queries](#)
[Creating and Querying Views](#)
[Creating and Refreshing a Materialized View](#)
[Inserting, Updating, and Deleting Data Using a View](#)
[Creating Your Own Functions and Procedures](#)
[Creating the percent_change\(\) Function](#)

[Using the percent_change\(\) Function](#)
[Updating Data with a Procedure](#)
[Using the Python Language in a Function](#)
[Automating Database Actions with Triggers](#)
[Logging Grade Updates to a Table](#)
[Automatically Classifying Temperatures](#)
[Wrapping Up](#)

CHAPTER 18: USING POSTGRESQL FROM THE COMMAND LINE

[Setting Up the Command Line for psql](#)
[Windows psql Setup](#)
[macOS psql Setup](#)
[Linux psql Setup](#)
[Working with psql](#)
[Launching psql and Connecting to a Database](#)
[Running SQL Queries on psql](#)
[Navigating and Formatting Results](#)
[Meta-Commands for Database Information](#)
[Importing, Exporting, and Using Files](#)
[Additional Command Line Utilities to Expedite Tasks](#)
[Adding a Database with createdb](#)
[Loading Shapefiles with shp2pgsql](#)
[Wrapping Up](#)

CHAPTER 19: MAINTAINING YOUR DATABASE

[Recovering Unused Space with VACUUM](#)
[Tracking Table Size](#)
[Monitoring the Autovacuum Process](#)
[Running VACUUM Manually](#)

[Reducing Table Size with VACUUM FULL](#)
[Changing Server Settings](#)
[Locating and Editing postgresql.conf](#)
[Reloading Settings with pg_ctl](#)
[Backing Up and Restoring Your Database](#)
[Using pg_dump to Export a Database or Table](#)
[Restoring a Database Export with pg_restore](#)
[Exploring Additional Backup and Restore Options](#)
[Wrapping Up](#)

CHAPTER 20: TELLING YOUR DATA'S STORY

[Start with a Question](#)
[Document Your Process](#)
[Gather Your Data](#)
[No Data? Build Your Own Database](#)
[Assess the Data's Origins](#)
[Interview the Data with Queries](#)
[Consult the Data's Owner](#)
[Identify Key Indicators and Trends over Time](#)
[Ask Why](#)
[Communicate Your Findings](#)
[Wrapping Up](#)

APPENDIX: ADDITIONAL POSTGRESQL RESOURCES

[PostgreSQL Development Environments](#)
[PostgreSQL Utilities, Tools, and Extensions](#)
[PostgreSQL News and Community](#)
[Documentation](#)

[INDEX](#)

PRACTICAL SQL

2nd Edition

A Beginner's Guide to Storytelling with Data

by Anthony DeBarros



San Francisco

PRACTICAL SQL, 2ND EDITION. Copyright © 2022 by Anthony DeBarros.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

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>

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc.
Other product and company names mentioned herein may be the trademarks of their respective

owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

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:

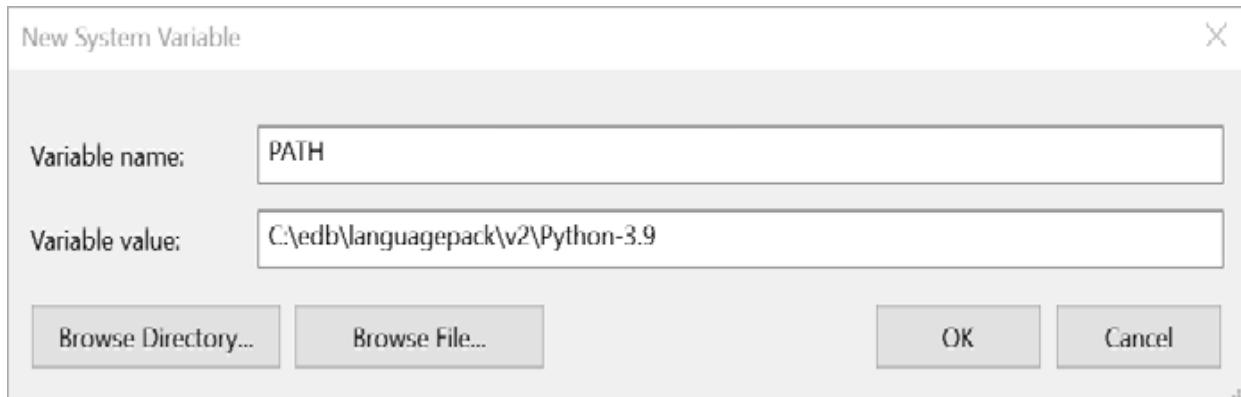
Open the Windows Control Panel by clicking the **Search** icon on the Windows taskbar, entering **Control Panel**, and then clicking the **Control Panel** icon.

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](#).



[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.

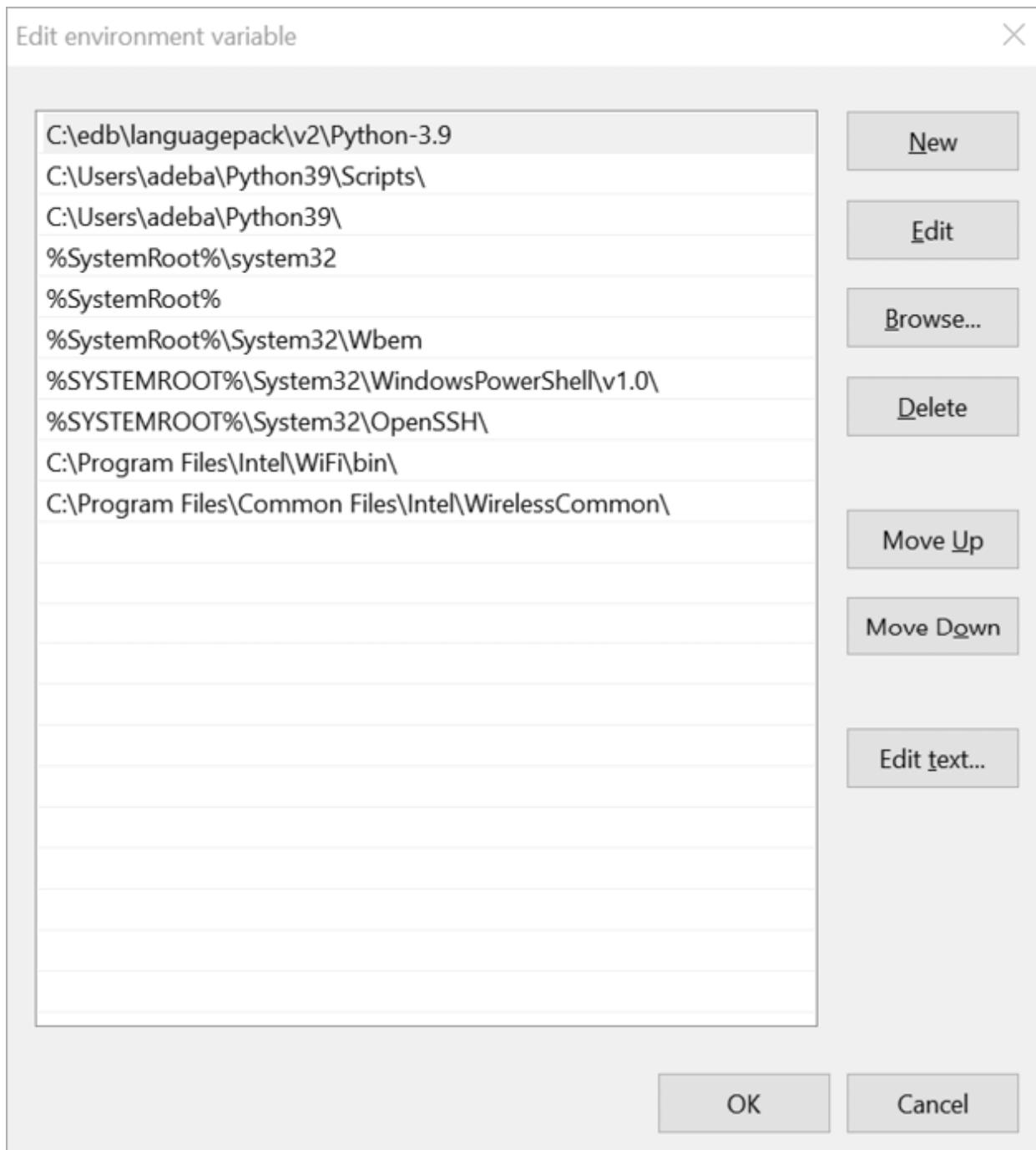


Figure 1-2: Editing existing PATH environment variables in Windows 10

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_release -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.*

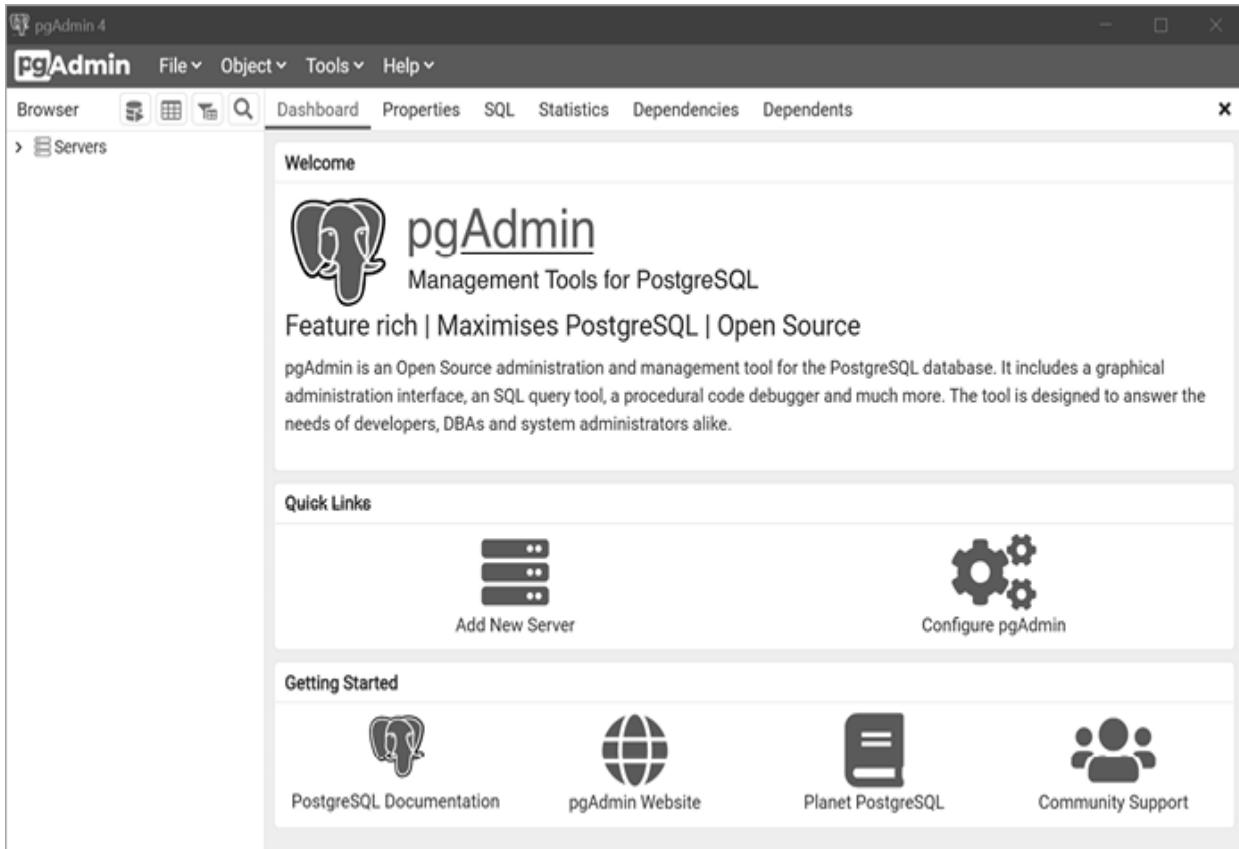


Figure 1-3: The pgAdmin app running on Windows 10

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](#).

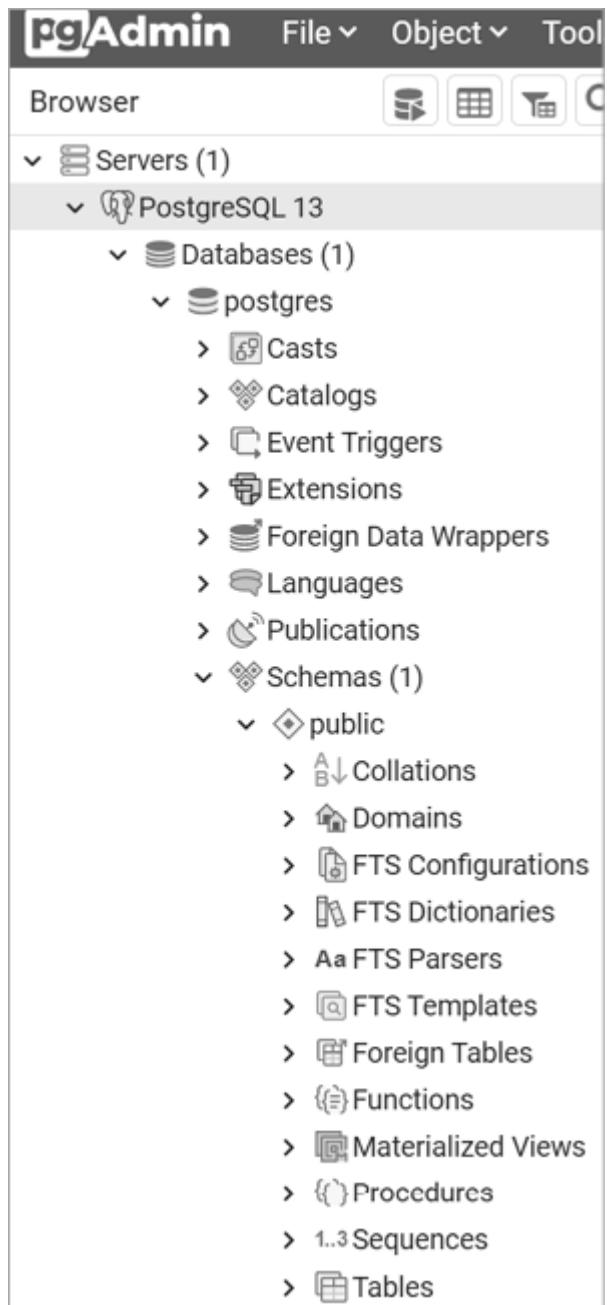


Figure 1-4: The pgAdmin object browser

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**.*

The screenshot shows the pgAdmin Query Tool interface. At the top, there's a menu bar with links to Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, and a file named Chapter_01.sql. Below the menu is a toolbar with various icons for file operations like Open, Save, Print, and a search bar. The main window has tabs for Query Editor (which is active) and Query History. In the Query Editor tab, a code snippet is shown:

```
7  
8 -- Listing 1-1: Checking your PostgreSQL version  
9  
10 SELECT version();  
11  
12  
13  
14  
15  
16  
17
```

Below the code, there are tabs for Data Output, Explain, Messages, and Notifications. The Data Output tab is selected and displays a single row of results:

	version	
1	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	Abrial	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:

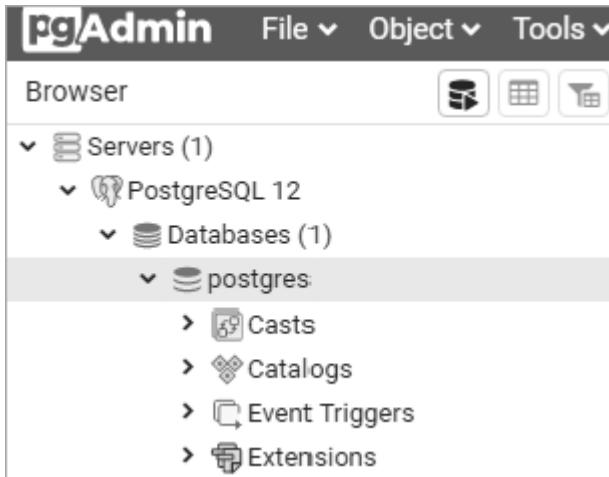
Run PostgreSQL. If you’re using Windows, the installer sets PostgreSQL to launch every time you boot up. On macOS, you must double-click `Postgres.app` in your Applications folder (if you have an elephant icon in your menu bar, it’s already running).

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](#).

The screenshot shows the pgAdmin Query Editor interface. The top menu bar includes Dashboard, Properties, SQL, Statistics, Dependencies, and Dependents. Below the menu is a toolbar with various icons for file operations like Open, Save, and Print. The title bar displays the connection information: postgres/postgres@PostgreSQL 12. The main area has tabs for Query Editor and Query History, with Query Editor selected. A query is entered in the editor:

```
1 CREATE DATABASE analysis;
```

Below the editor, there are tabs for Data Output, Explain, Messages, and Notifications, with Messages selected. The message pane shows the command and its successful execution:

```
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](#).

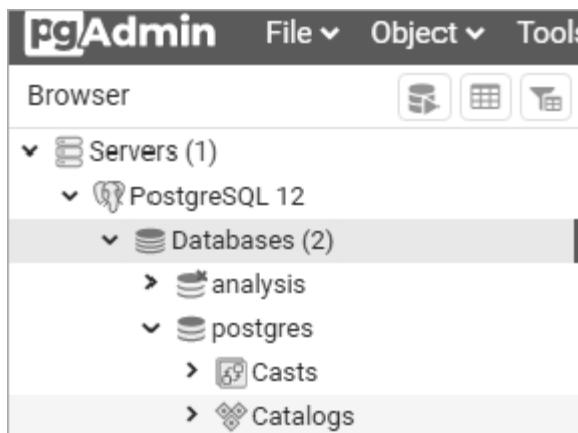


Figure 2-3: The `analysis` database displayed in the object browser

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.

```
| 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
| );
```

[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](#).

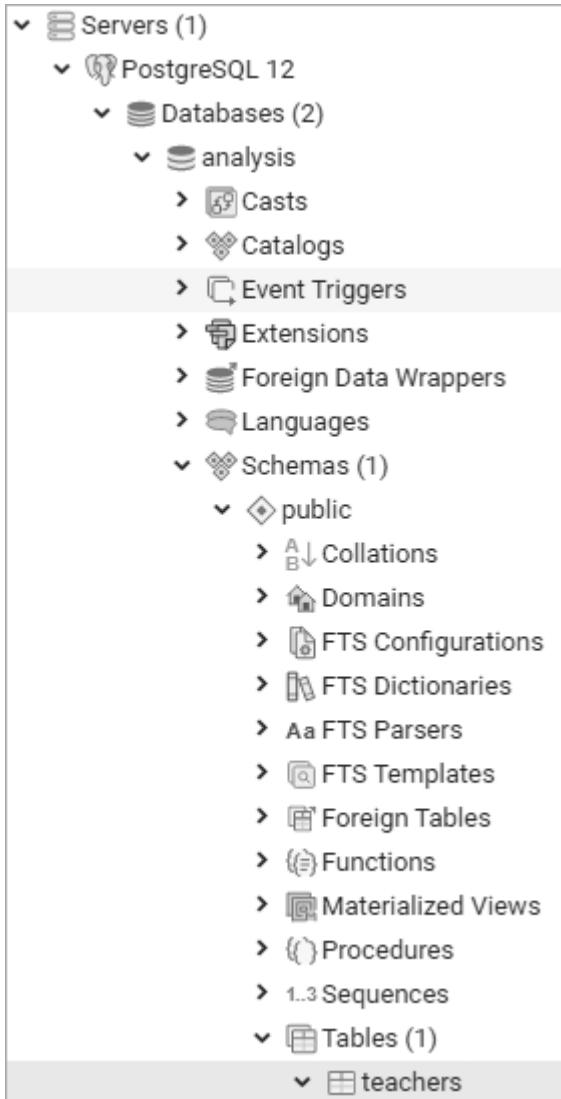


Figure 2-4: The `teachers` table in the object browser

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).

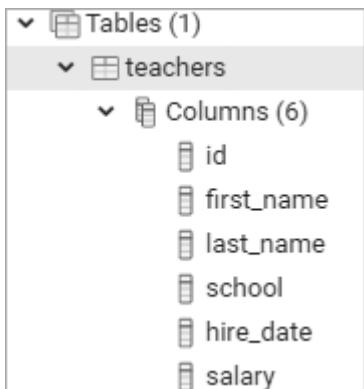


Figure 2-5: Table details for `teachers`

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).

```
| INSERT INTO teachers (first_name, last_name, school, hire_date,
    salary)
?
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.

Data Output Explain Messages Notifications						
	<code>id</code> bigint	<code>first_name</code> character varying (2)	<code>last_name</code> character varying (5)	<code>school</code> character varying (50)	<code>hire_date</code> date	<code>salary</code> 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.

<code>id</code>	<code>first_name</code>	<code>last_name</code>	<code>school</code>
<code>hire_date</code>		<code>salary</code>	

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
| 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
| WHERE first_name LIKE 'sam%';
?
SELECT first_name
FROM teachers
? 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
| WHERE school = 'Myers Middle School'
      AND salary < 40000;

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

SELECT *
FROM teachers
} 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
```

```
) ;  
  
? INSERT INTO char_data_types  
VALUES  
    ('abc', 'abc', 'abc'),  
    ('defghi', 'defghi', 'defghi');  
  
} COPY char_data_types TO 'C:\YourDirectory\typetest.txt'  
| 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:

Iintegers 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
<code>smallint</code>	2 bytes	-32768 to +32767
<code>integer</code>	4 bytes	-2147483648 to +2147483647
<code>bigint</code>	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 bytes	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 (
    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
? 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](#).

```
| CREATE TABLE date_time_types (
|   timestamp_column timestamp with time zone,
|   interval_column interval
| );
|
? 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 01:00:00-05	2 days	2022-12-29
2022-12-31 04:00:00-05 04:00:00-05	1 mon	2022-11-30
2022-12-30 09:00:00-05 09:00:00-05	100 years	1922-12-30
2020-05-31 21:31:15.716063-05 21:31:15.716063-05	7 days	2020-05-24

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.

```
| SELECT timestamp_column, CAST(timestamp_column AS varchar(10))
  FROM date_time_types;
?
? 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 `SELECT` **3** 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:

Obtain the source data in the form of a delimited text file.

Create a table to store the data.

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/wikil/> 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](#).

```
| COPY table_name
| FROM 'C:\YourDirectory\your_file.csv'
| 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,
 2   county_fips text,
 3 region smallint,
 3 state_name text,
 4   county_name text,
 4 area_land bigint,
 5   area_water bigint,
 5 internal_point_lat numeric(10,7),
 6   internal_point_lon numeric(10,7),
 6 pop_est_2018 integer,
 7   pop_est_2019 integer,
 7 births_2019 integer,
 7 deaths_2019 integer,
 7 international_migr_2019 integer,
 7 domestic_migr_2019 integer,
 7 residual_2019 integer,
 7 CONSTRAINT counties_2019_key PRIMARY KEY (state_fips,
 7   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-adata.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 internal_point_lon	state_name	internal_point_lat
Aleutians West Census Area Alaska 179.6211882		51.9489640
Washington County -67.6093542	Maine	44.9670088
Hancock County -68.3707034	Maine	44.5649063
Aroostook County -68.6124095	Maine	46.7091929
Penobscot County -68.6666160	Maine	45.4092843

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.

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

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

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

! 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:

<code>id</code>	<code>town</code>	<code>county</code>	<code>supervisor</code>	<code>start_date</code>	<code>salary</code>
benefits					

```
-- -----  
-- -----  
11 Anytown      Mills      Jones  
67000.00  
12 Bumbleyburg  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](#).

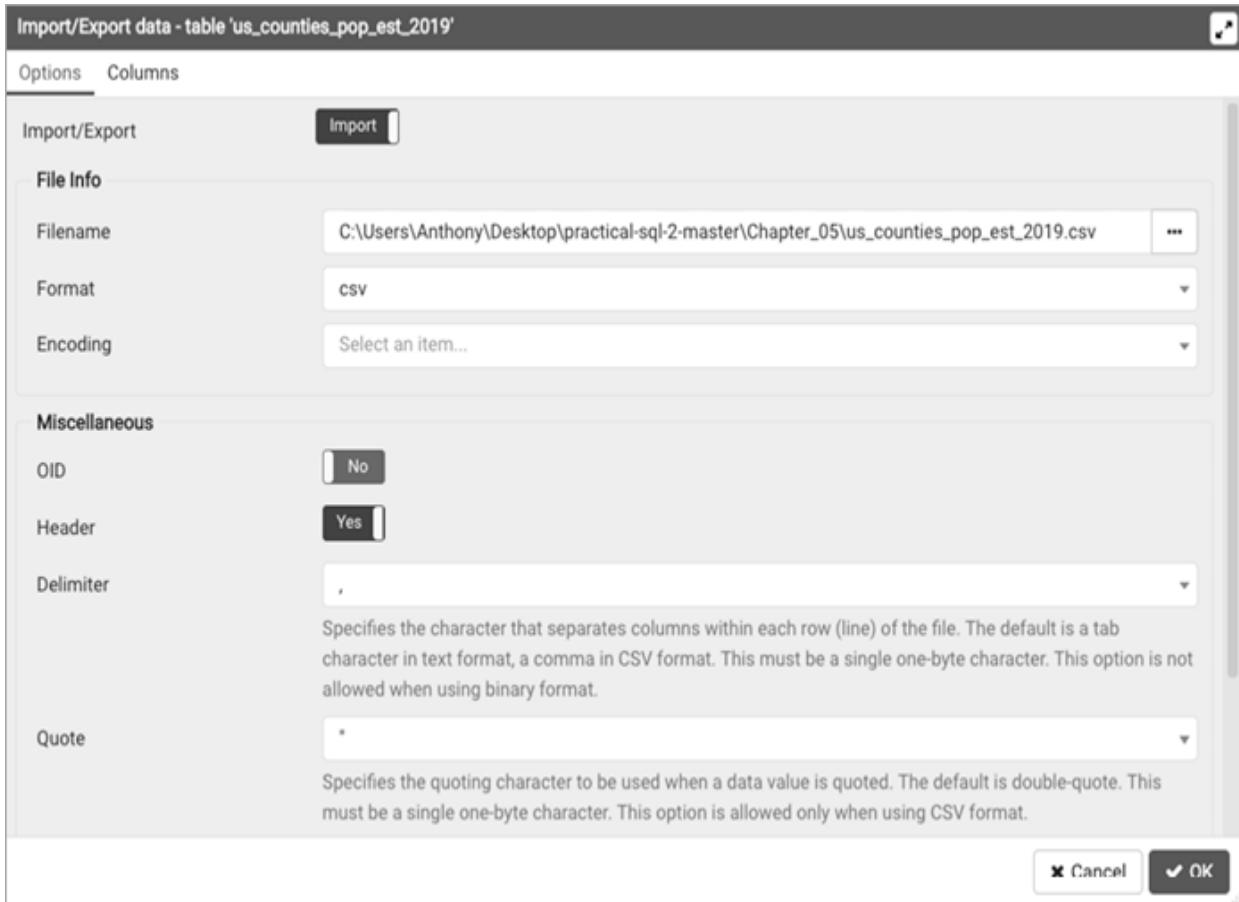


Figure 5-1: The pgAdmin Import/Export dialog

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.

```
| SELECT 2 + 2;  
| SELECT 9 - 1;  
| 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.

```
| SELECT 11 / 6;
| SELECT 11 % 6;
| SELECT 11.0 / 6;
+ 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 $4 \times 3 \times 2 \times 1$. 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 AS 1 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
  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
Alaska	AK	100.000000

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.

```
| CREATE TABLE percent_change (
    department text,
```

```

    spend_2019 numeric(10,2),
    spend_2022 numeric(10,2)
);

? 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](#).

```
| SELECT *
| FROM employees JOIN departments
| 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`:

<code>id</code>	<code>school_2020</code>
--	-----
1	Oak Street School
2	Roosevelt High School
5	Dover Middle School
6	Webutuck High School

There are five rows in `district_2035`:

<code>id</code>	<code>school_2035</code>
--	-----
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
| 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:

<code>id</code>	<code>school_2020</code>	<code>id</code>	<code>school_2035</code>
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
| 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:

<code>id</code>	<code>enrollment</code>
--	-----
1	360
2	1001
5	450
6	927

The `district_2020_grades` table contains the grade levels housed in each building:

<code>id</code>	<code>grades</code>
--	-----
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
  | FROM district_2020 AS d20 JOIN district_2020_enrollment AS en
  |   ON d20.id = en.id
? 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:

<code>id</code>	<code>school_2020</code>	<code>enrollment</code>	<code>grades</code>
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`.

```
1 SELECT * FROM district_2020
| UNION
2   SELECT * FROM district_2035
? 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.

```
| 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
| INTERSECT
|   SELECT * FROM district_2035
|   ORDER BY id;

SELECT * FROM district_2020
? 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](#).

```
| 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)
);

? COPY us_counties_pop_est_2010
  FROM 'C:\YourDirectory\us_counties_pop_est_2010.csv'
  WITH (FORMAT CSV, HEADER);

} 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
} ON c2019.state_fips = c2010.state_fips
   AND c2019.county_fips = c2010.county_fips
} 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 pct_change	state_name	pop_2019	pop_2010	raw_change
McKenzie County 136.3	North Dakota	15024	6359	8665
Loving County 106.1	Texas	169	82	87
Williams County 67.8	North Dakota	37589	22399	15190
Hays County 46.5	Texas	230191	157103	73088
Wasatch County 44.9	Utah	34091	23525	10566

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:

<code>driver_id</code>	<code>st</code>	<code>first_name</code>	<code>last_name</code>
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
| OVERRIDING SYSTEM VALUE
|   VALUES (4, 'Chicken Coop', '2021-09-03 10:33-07');

? ALTER TABLE surrogate_key_example ALTER COLUMN order_number
  RESTART WITH 5;

} 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)
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
    -> Parallel Seq Scan on new_york_addresses
      (cost=0.00..13873.78 1
        rows=1293 width=46) (actual time=2.362..367.258 rows=1112
        loops=3)
          Filter: (street = 'BROADWAY'::text)
          Rows Removed by Filter: 312346
      Planning Time: 0.401 ms
      Execution Time: 389.232 ms 2
```

Not all the output is relevant here, so I won't decode it all, but two lines are pertinent. The first indicates that to find any rows where `street = 'BROADWAY'`, the database will conduct a sequential scan **1** of the table. That's a synonym for a full table scan: the database will examine each row and remove any where `street` doesn't match `BROADWAY`. The execution time (on my computer about 389 milliseconds) **2** is how long the query took to run. Your time will depend on factors including your computer hardware.

For the test, run each query in [Listing 8-13](#) several times and record the fastest execution time for each. You'll notice that execution times for the same query will vary slightly on each run. That can be the result of several factors, from other processes running on the server to the effect of data being held in memory after a prior run of the query.

Adding the Index

Now, let's see how adding an index changes the query's search method and execution time. [Listing 8-14](#) shows the SQL statement for creating the index with PostgreSQL.

```
CREATE INDEX street_idx ON new_york_addresses (street);
```

[Listing 8-14](#): Creating a B-tree index on the `new_york_addresses` table

Notice that it's similar to the commands for creating constraints. We give the `CREATE INDEX` keywords followed by a name we choose for the index, in this case `street_idx`. Then `ON` is added, followed by the target table and column.

Execute the `CREATE INDEX` statement, and PostgreSQL will scan the values in the `street` column and build the index from them. We need to create the index only once. When the task finishes, rerun each of the three queries in [Listing 8-13](#) and record the execution times reported by `EXPLAIN ANALYZE`. Here's an example:

```
Bitmap Heap Scan on new_york_addresses  (cost=76.47..6389.39
rows=3103 width=46)  (actual time=1.355..4.802 rows=3336
loops=1)
    Recheck Cond: (street = 'BROADWAY'::text)
    Heap Blocks: exact=2157
    -> Bitmap Index Scan on street_idx  (cost=0.00..75.70
rows=3103 width=0)  1
        (actual time=0.950..0.950 rows=3336 loops=1)
            Index Cond: (street = 'BROADWAY'::text)
Planning Time: 0.109 ms
Execution Time: 5.113 ms  2
```

Do you notice a change? First, we see that the database is now using an index scan on `street_idx` 1 instead of visiting each row in a sequential scan. Also, the query speed is now markedly faster 2. [Table 8-1](#) shows the fastest execution times (rounded) from my computer before and after adding the index.

Table 8-1: Measuring Index Performance

Query filter	Before index	After index
WHERE street = 'BROADWAY'	92 ms	5 ms
WHERE street = '52 STREET'	94 ms	1 ms
WHERE street = 'ZWICKY AVENUE'	93 ms	<1 ms

The execution times are much, much better, nearly a tenth of a second faster or more per query. Is a tenth of a second that impressive? Well, whether you’re seeking answers in data using repeated querying or creating a database system for thousands of users, the time savings adds up.

If you ever need to remove an index from a table—perhaps if you’re testing the performance of several index types—use the `DROP INDEX` command followed by the name of the index to remove.

Considerations When Using Indexes

You’ve seen that indexes have significant performance benefits, so does that mean you should add an index to every column in a table? Not so fast! Indexes are valuable, but they’re not always needed. In addition, they do enlarge the database and impose a maintenance cost on writing data. Here are a few tips for judging when to use indexes:

Consult the documentation for the database system you’re using to learn about the kinds of indexes available and which to use on particular data types.

PostgreSQL, for example, has five more index types in addition to B-tree. One, called GiST, is particularly suited to the geometry data types discussed later in the book. Full-text search, which you’ll learn in Chapter 14, also benefits from indexing.

Consider adding indexes to columns you’ll use in table joins. Primary keys are indexed by default in PostgreSQL, but foreign key columns in related tables are not and are a good target for indexes.

An index on a foreign key will help avoid an expensive sequential scan during a cascading delete.

Add indexes to columns that will frequently end up in a query `WHERE` clause. As you’ve seen, search performance is significantly improved via indexes.

Use `EXPLAIN ANALYZE` to test the performance under a variety of configurations. Optimization is a process! If an index isn't being used by the database—and it's not backing up a primary key or other constraint—you can drop it to reduce the size of your database and speed up inserts, updates, and deletes.

Wrapping Up

With the tools you've added to your toolbox in this chapter, you're ready to ensure that the databases you build or inherit are best suited for your collection and exploration of data. It's crucial to define constraints that match the data and the expectation of users by not allowing values that don't make sense, making sure values are filled in, and setting up proper relationships between tables. You've also learned how to make your queries run faster and how to consistently organize your database objects. That's a boon for you and for others who share your data.

This chapter concludes the first part of the book, which focused on giving you the essentials to dig into SQL databases. We'll continue building on these foundations as we explore more complex queries and strategies for data analysis. In the next chapter, we'll use SQL aggregate functions to assess the quality of a dataset and get usable information from it.

TRY IT YOURSELF

Are you ready to test yourself on the concepts covered in this chapter? Consider the following two tables from a database you're making to keep track of your vinyl LP collection. Start by reviewing these CREATE TABLE statements:

```
CREATE TABLE albums (
    album_id bigint GENERATED ALWAYS AS IDENTITY,
    catalog_code text,
    title text,
    artist text,
    release_date date,
    genre text,
    description text
);

CREATE TABLE songs (
    song_id bigint GENERATED ALWAYS AS IDENTITY,
    title text,
    composers text,
    album_id bigint
);
```

The `albums` table includes information specific to the overall collection of songs on the disc. The `songs` table catalogs each track on the album. Each song has a title and a column for its composers, who might be different than the album artist.

Use the tables to answer these questions:

Modify these CREATE TABLE statements to include primary and foreign keys plus additional constraints on both tables. Explain why you made your choices.

Instead of using `album_id` as a surrogate key for your primary key, are there any columns in `albums` that could be useful as a natural key? What would you have to know to decide?

To speed up queries, which columns are good candidates for indexes?

9

EXTRACTING INFORMATION BY GROUPING AND SUMMARIZING



Every dataset tells a story, and it's the data analyst's job to find it. In Chapter 3, you learned about interviewing data using SELECT statements by sorting columns, finding distinct values, and filtering results. You've also learned the fundamentals of SQL math, data types, table design, and joining tables. With these tools under your belt, you're ready to glean more insights by using *grouping* and *aggregate functions* to summarize your data.

By summarizing data, we can identify useful information we wouldn't see just by scanning the rows of a table. In this chapter, we'll use the well-known institution of your local library as our example.

Libraries remain a vital part of communities worldwide, but the internet and advancements in library technology have changed how we use them. For example, ebooks and online access to digital materials now have a permanent place in libraries along with books and periodicals.

In the United States, the Institute of Museum and Library Services (IMLS) measures library activity as part of its annual Public Libraries Survey. The survey collects data from about 9,000 library administrative entities, defined by the survey as agencies that provide library services to a particular locality. Some agencies are county library systems, and others are part of school districts. Data

on each agency includes the number of branches, staff, books, hours open per year, and so on. The IMLS has been collecting data each year since 1988 and includes all public library agencies in the 50 states plus the District of Columbia and US territories such as American Samoa. (Read more about the program at <https://www.imls.gov/research-evaluation/data-collection/public-libraries-survey>.)

For this exercise, we'll assume the role of an analyst who just received a fresh copy of the library dataset to produce a report describing trends from the data. We'll create three tables to hold data from the 2018, 2017, and 2016 surveys. (Often, it's helpful to assess multiple years of data to discern trends.) Then we'll summarize the more interesting data in each table and join the tables to see how measures changed over time.

Creating the Library Survey Tables

Let's create the three library survey tables and import the data. We'll use appropriate data types and constraints for each column and add indexes where appropriate. The code and three CSV files are available in the book's resources.

Creating the 2018 Library Data Table

We'll start by creating the table for the 2018 library data. Using the CREATE TABLE statement, [Listing 9-1](#) builds `pls_fy2018_libraries`, a table for the fiscal year 2018 Public Library System Data File from the Public Libraries Survey. The Public Library System Data File summarizes data at the agency level, counting activity at all agency outlets, which include central libraries, branch libraries, and bookmobiles. The annual survey generates two additional files we won't use: one summarizes data at the state level, and the other has data on individual outlets. For this exercise, those files are redundant, but you can read about the data they contain at

https://www.imls.gov/sites/default/files/2018_pls_data_file_documentation.pdf.

For convenience, I've created a naming scheme for the tables: `pls` refers to the survey title, `fy2018` is the fiscal year the data covers, and `libraries` is the name of the particular file from the survey. For simplicity, I've selected 47 of the more relevant columns from the 166 in the original survey file to fill the `pls_fy2018_libraries` table, excluding data such as the codes that explain the source of individual responses. When a library didn't provide data, the agency derived the data using other means, but we don't need that information for this exercise.

[Listing 9-1](#) is abbreviated for convenience, as indicated by the `--snip--` noted in the code, but the full version is included with the book's resources.

```
CREATE TABLE pls_fy2018_libraries (
    stabr text NOT NULL,
    1 fscskey text CONSTRAINT fscskey_2018_pkey PRIMARY KEY,
    libid text NOT NULL,
    libname text NOT NULL,
    address text NOT NULL,
    city text NOT NULL,
    zip text NOT NULL,
    --snip--
    longitude numeric(10,7) NOT NULL,
    latitude numeric(10,7) NOT NULL
);

2 COPY pls_fy2018_libraries
    FROM 'C:\YourDirectory\pls_fy2018_libraries.csv'
    WITH (FORMAT CSV, HEADER);

3 CREATE INDEX libname_2018_idx ON pls_fy2018_libraries
    (libname);
```

[Listing 9-1](#): Creating and filling the 2018 Public Libraries Survey table

After finding the code and data file for [Listing 9-1](#), connect to your analysis database in pgAdmin and run it. Make sure you remember to change `C:\YourDirectory\` to the path where you saved the `pls_fy2018_libraries.csv` file.

First, the code makes the table via `CREATE TABLE`. We assign a primary key constraint to the column named `fscskey` 1, a unique code the data dictionary says is assigned to each library. Because it's unique, present in each row, and unlikely to change, it can serve as a natural primary key.

The definition for each column includes the appropriate data type and `NOT NULL` constraints where the columns have no missing values. The `startdate` and `enddate` columns contain dates, but we've set their data type to `text` in the code; in the CSV file, those columns include nondate values, and our import will fail if we try to use a `date` data type. In Chapter 10, you'll learn how to clean up cases like these. For now, those columns are fine as is.

After creating the table, the `COPY` statement 2 imports the data from a CSV file named `pls_fy2018_libraries.csv` using the file path you provide. We add an index 3

to the libname column to provide faster results when we search for a particular library.

Creating the 2017 and 2016 Library Data Tables

Creating the tables for the 2017 and 2016 library surveys follows similar steps. I've combined the code to create and fill both tables in [Listing 9-2](#). Note again that the listing shown is truncated, but the full code is in the book's resources at <https://nostarch.com/practical-sql-2nd-edition/>.

Update the file paths in the COPY statements for both imports and execute the code.

```
CREATE TABLE pls_fy2017_libraries (
    stabr text NOT NULL,
    fscskey text CONSTRAINT fscskey_17_pkey PRIMARY KEY,
    libid text NOT NULL,
    libname text NOT NULL,
    address text NOT NULL,
    city text NOT NULL,
    zip text NOT NULL,
    --snip--
    longitude numeric(10,7) NOT NULL,
    latitude numeric(10,7) NOT NULL
);

CREATE TABLE pls_fy2016_libraries (
    stabr text NOT NULL,
    fscskey text CONSTRAINT fscskey_16_pkey PRIMARY KEY,
    libid text NOT NULL,
    libname text NOT NULL,
    address text NOT NULL,
    city text NOT NULL,
    zip text NOT NULL,
    --snip--
    longitude numeric(10,7) NOT NULL,
    latitude numeric(10,7) NOT NULL
);

? COPY pls_fy2017_libraries
  FROM 'C:\YourDirectory\pls_fy2017_libraries.csv'
  WITH (FORMAT CSV, HEADER);

COPY pls_fy2016_libraries
  FROM 'C:\YourDirectory\pls_fy2016_libraries.csv'
```

```
WITH (FORMAT CSV, HEADER);  
  
} CREATE INDEX libname_2017_idx ON pls_fy2017_libraries  
    (libname);  
CREATE INDEX libname_2016_idx ON pls_fy2016_libraries  
    (libname);
```

[Listing 9-2: Creating and filling the 2017 and 2016 Public Libraries Survey tables](#)

We start by creating the two tables, and in both we again use `fscskey` **1** as the primary key. Next, we run `COPY` commands **2** to import the CSV files to the tables, and, finally, we create an index on the `libname` column **3** in both tables.

As you review the code, you'll notice that the three tables have an identical structure. Most ongoing surveys will have a handful of year-to-year changes because the makers of the survey either think of new questions or modify existing ones, but the columns I've selected for these three tables are consistent. The documentation for the survey years is at <https://www.imls.gov/research-evaluation/data-collection/public-libraries-survey/>. Now, let's mine this data to discover its story.

Exploring the Library Data Using Aggregate Functions

Aggregate functions combine values from multiple rows, perform an operation on those values, and return a single result. For example, you might return the average of values with the `avg()` aggregate function, as you learned in Chapter 6. Some aggregate functions are part of the SQL standard, and others are specific to PostgreSQL and other database managers. Most of the aggregate functions used in this chapter are part of standard SQL (a full list of PostgreSQL aggregates is at <https://www.postgresql.org/docs/current/functions-aggregate.html>).

In this section, we'll work through the library data using aggregates on single and multiple columns and then explore how you can expand their use by grouping the results they return with values from additional columns.

Counting Rows and Values Using `count()`

After importing a dataset, a sensible first step is to make sure the table has the expected number of rows. The IMLS documentation says the file we imported for the 2018 data has 9,261 rows; 2017 has 9,245; and 2016 has 9,252. The difference likely reflects library openings, closings, or mergers. When we count the number of rows in those tables, the results should match those counts.

The `count()` aggregate function, which is part of the ANSI SQL standard, makes it easy to check the number of rows and perform other counting tasks. If we supply an asterisk as an input, such as `count(*)`, the asterisk acts as a wildcard, so the function returns the number of table rows regardless of whether they include `NULL` values. We do this in all three statements in [*Listing 9-3*](#).

```
SELECT count(*)
FROM pls_fy2018_libraries;

SELECT count(*)
FROM pls_fy2017_libraries;

SELECT count(*)
FROM pls_fy2016_libraries;
```

[*Listing 9-3: Using `count\(\)` for table row counts*](#)

Run each of the commands in [*Listing 9-3*](#) one at a time to see the table row counts. For `pls_fy2018_libraries`, the result should be as follows:

```
count
-----
9261
```

For `pls_fy2017_libraries`, you should see the following:

```
count
-----
9245
```

Finally, the result for `pls_fy2016_libraries` should be this:

```
count
-----
9252
```

All three results match the number of rows we expected. This is a good first step because it will alert us to issues such as missing rows or a case where we might have imported the wrong file.

NOTE

*You can also check the row count using the pgAdmin interface, but it's clunky. Right-clicking the table name in pgAdmin's object browser and selecting **View/Edit Data** ► **All Rows** executes a SQL query for all rows. Then, a pop-up message in the results pane shows the row count, but it disappears after a few seconds.*

Counting Values Present in a Column

If we supply a column name instead of an asterisk to `count()`, it will return the number of rows that are not `NULL`. For example, we can count the number of non-`NULL` values in the `phone` column of the `pls_fy2018_libraries` table using `count()` as in [Listing 9-4](#).

```
SELECT count(phone)
FROM pls_fy2018_libraries;
```

[Listing 9-4](#): Using `count()` for the number of values in a column

The result shows 9,261 rows have a value in `phone`, the same as the total rows we found earlier.

```
count
-----
9261
```

This means every row in the `phone` column has a value. You may have suspected this already, given that the column has a `NOT NULL` constraint in the `CREATE TABLE` statement. But running this check is worthwhile because the absence of values might influence your decision on whether to proceed with analysis at all. To fully vet the data, checking with topical experts and digging deeper into the data is usually a good idea; I recommend seeking expert advice as part of a broader analysis methodology (for more on this topic, see Chapter 20).

Counting Distinct Values in a Column

In Chapter 3, I covered the `DISTINCT` keyword—part of the SQL standard—which with `SELECT` returns a list of unique values. We can use it to see unique values in a single column, or we can see unique combinations of values from multiple columns. We also can add `DISTINCT` to the `count()` function to return a count of distinct values from a column.

[Listing 9-5](#) shows two queries. The first counts all values in the 2018 table’s `libname` column. The second does the same but includes `DISTINCT` in front of the column name. Run them both, one at a time.

```
SELECT count(libname)
FROM pls_fy2018_libraries;

SELECT count(DISTINCT libname)
FROM pls_fy2018_libraries;
```

[Listing 9-5:](#) Using `count()` for the number of distinct values in a column

The first query returns a row count that matches the number of rows in the table that we found using [Listing 9-3](#):

```
count
-----
9261
```

That’s good. We expect to have the library agency name listed in every row. But the second query returns a smaller number:

```
count
-----
8478
```

Using `DISTINCT` to remove duplicates reduces the number of library names to the 8,478 that are unique. Closer inspection of the data shows that 526 library agencies in the 2018 survey shared their name with one or more other agencies. Ten library agencies are named `OXFORD PUBLIC LIBRARY`, each one in a city or town named Oxford in different states, including Alabama, Connecticut, Kansas, and Pennsylvania, among others. We’ll write a query to see combinations of distinct values in the “Aggregating Data Using GROUP BY” section.

Finding Maximum and Minimum Values Using `max()` and `min()`

The `max()` and `min()` functions give us the largest and smallest values in a column and are useful for a couple of reasons. First, they help us get a sense of the scope of the values reported. Second, the functions can reveal unexpected issues with data, as you'll see now.

Both `max()` and `min()` work the same way, with the name of a column as input. [Listing 9-6](#) uses `max()` and `min()` on the 2018 table, taking the `visits` column that records the number of annual visits to the library agency and all of its branches. Run the code.

```
SELECT max(visits), min(visits)
FROM pls_fy2018_libraries;
```

[Listing 9-6](#): *Finding the most and fewest visits using `max()` and `min()`*

The query returns the following results:

max	min
-----	---
16686945	-3

Well, that's interesting. The maximum value of more than 16.6 million is reasonable for a large city library system, but -3 as the minimum? On the surface, that result seems like a mistake, but it turns out that the creators of the library survey are employing a common but potentially problematic convention in data collection by placing a negative number or some artificially high value in a column to indicate some condition.

In this case, negative values in number columns indicate the following:

A value of -1 indicates a “nonresponse” to that question.

A value of -3 indicates “not applicable” and is used when a library agency has closed either temporarily or permanently.

We'll need to account for and exclude negative values as we explore the data, because summing a column and including the negative values will result in an incorrect total. We can do this using a `WHERE` clause to filter them. It's a good reminder to always read the documentation for the data to get ahead of the

issue instead of having to backtrack after spending a lot of time on deeper analysis!

NOTE

A better alternative for this negative value scenario is to use NULL in rows in the visits column where response data is absent and then create a separate visits_flag column to hold codes explaining why.

Aggregating Data Using GROUP BY

When you use the GROUP BY clause with aggregate functions, you can group results according to the values in one or more columns. This allows us to perform operations such as `sum()` or `count()` for every state in the table or for every type of library agency.

Let's explore how using GROUP BY with aggregate functions works. On its own, GROUP BY, which is also part of standard ANSI SQL, eliminates duplicate values from the results, similar to DISTINCT. [Listing 9-7](#) shows the GROUP BY clause in action.

```
SELECT stabr
  FROM pls_fy2018_libraries
  | GROUP BY stabr
  | ORDER BY stabr;
```

[Listing 9-7:](#) Using GROUP BY on the stabr column

We add the GROUP BY clause 1 after the FROM clause and include the column name to group. In this case, we're selecting `stabr`, which contains the state abbreviation, and grouping by that same column. We then use `ORDER BY stabr` as well so that the grouped results are in alphabetical order. This will yield a result with unique state abbreviations from the 2018 table. Here's a portion of the results:

```
stabr
-----
AK
AL
AR
```

```
AS  
AZ  
CA  
--snip--  
WV  
WY
```

Notice that there are no duplicates in the 55 rows returned. These standard two-letter postal abbreviations include the 50 states plus Washington, DC, and several US territories, such as Guam and the US Virgin Islands.

You're not limited to grouping just one column. In [Listing 9-8](#), we use the GROUP BY clause on the 2018 data to specify the city and stabr columns for grouping.

```
SELECT city, stabr  
FROM pls_fy2018_libraries  
GROUP BY city, stabr  
ORDER BY city, stabr;
```

[Listing 9-8:](#) Using GROUP BY on the city and stabr columns

The results get sorted by city and then by state, and the output shows unique combinations in that order:

city	stabr
ABBEVILLE	AL
ABBEVILLE	LA
ABBEVILLE	SC
ABBOTSFORD	WI
ABERDEEN	ID
ABERDEEN	SD
ABERNATHY	TX

--snip--

This grouping returns 9,013 rows, 248 fewer than the total table rows. The result indicates that the file includes multiple instances where there's more than one library agency for a particular city and state combination.

Combining GROUP BY with count()

If we combine GROUP BY with an aggregate function, such as count(), we can pull more descriptive information from our data. For example, we know 9,261 library agencies are in the 2018 table. We can get a count of agencies by state and sort them to see which states have the most. [Listing 9-9](#) shows how to do this.

```
| 1 SELECT stabr, count(*)
|   FROM pls_fy2018_libraries
| 2 GROUP BY stabr
| 3 ORDER BY count(*) DESC;
```

[Listing 9-9](#): Using GROUP BY with count() on the stabr column

We're now asking for the values in the stabr column and a count of how many rows have a given stabr value. In the list of columns to query 1, we specify stabr and count() with an asterisk as its input, which will cause count() to include NULL values. Also, when we select individual columns along with an aggregate function, we must include the columns in a GROUP BY clause 2. If we don't, the database will return an error telling us to do so, because you can't group values by aggregating and have ungrouped column values in the same query.

To sort the results and have the state with the largest number of agencies at the top, we can use an ORDER BY clause 3 that includes the count() function and the DESC keyword.

Run the code in [Listing 9-9](#). The results show New York, Illinois, and Texas as the states with the greatest number of library agencies in 2018:

stabr	count
-----	-----
NY	756
IL	623
TX	560
IA	544
PA	451
MI	398
WI	381
MA	369
--snip--	

Remember that our table represents library agencies that serve a locality. Just because New York, Illinois, and Texas have the greatest number of library agencies doesn't mean they have the greatest number of outlets where you can walk in and peruse the shelves. An agency might have one central library only, or it might have no central libraries but 23 branches spread around a county. To count outlets, each row in the table also has values in the columns `centlib` and `branlib`, which record the number of central and branch libraries, respectively. To find totals, we would use the `sum()` aggregate function on both columns.

Using GROUP BY on Multiple Columns with count()

We can glean yet more information from our data by combining `GROUP BY` with `count()` and multiple columns. For example, the `stataddr` column in all three tables contains a code indicating whether the agency's address changed in the last year. The values in `stataddr` are as follows:

00 No change from last year

07 Moved to a new location

15 Minor address change

[*Listing 9-10*](#) shows the code for counting the number of agencies in each state that moved, had a minor address change, or had no change using `GROUP BY` with `stabr` and `stataddr` and adding `count()`.

```
| SELECT stabr, stataddr, count(*)
|   FROM pls_fy2018_libraries
? GROUP BY stabr, stataddr
? ORDER BY stabr, stataddr;
```

[*Listing 9-10: Using GROUP BY with count\(\) of the stabr and stataddr columns*](#)

The key sections of the query are the column names and the `count()` function after `SELECT 1`, and making sure both columns are reflected in the `GROUP BY` clause **2** to ensure that `count()` will show the number of unique combinations of `stabr` and `stataddr`.

To make the output easier to read, let's sort first by the state and address status codes in ascending order **3**. Here are the results:

stabr	stataddr	count
AK	00	82
AL	00	220
AL	07	3
AL	15	1
AR	00	58
AR	07	1
AR	15	1
AS	00	1
--snip--		

The first few rows show that code 00 (no change in address) is the most common value for each state. We'd expect that because it's likely there are more library agencies that haven't changed address than those that have. The result helps assure us that we're analyzing the data in a sound way. If code 07 (moved to a new location) was the most frequent in each state, that would raise a question about whether we've written the query correctly or whether there's an issue with the data.

Revisiting sum() to Examine Library Activity

Now let's expand our techniques to include grouping and aggregating across joined tables using the 2018, 2017, and 2016 libraries data. Our goal is to identify trends in library visits spanning that three-year period. To do this, we need to calculate totals using the `sum()` aggregate function.

Before we dig into these queries, let's address the values -3 and -1, which indicate "not applicable" and "nonresponse." To prevent these negative numbers from affecting the analysis, we'll filter them out using a `WHERE` clause to limit the queries to rows where values in `visits` are zero or greater.

Let's start by calculating the sum of annual visits to libraries from the individual tables. Run each `SELECT` statement in [Listing 9-11](#) separately.

```
SELECT sum(visits) AS visits_2018
FROM pls_fy2018_libraries
WHERE visits >= 0;

SELECT sum(visits) AS visits_2017
FROM pls_fy2017_libraries
WHERE visits >= 0;
```

```
SELECT sum(visits) AS visits_2016
FROM pls_fy2016_libraries
WHERE visits >= 0;
```

[Listing 9-11](#): Using the `sum()` aggregate function to total visits to libraries in 2016, 2017, and 2018

For 2018, visits totaled approximately 1.29 billion:

```
visits_2018
-----
1292348697
```

For 2017, visits totaled approximately 1.32 billion:

```
visits_2017
-----
1319803999
```

And for 2016, visits totaled approximately 1.36 billion:

```
visits_2016
-----
1355648987
```

We're onto something here, but it may not be good news for libraries. The trend seems to point downward with visits dropping about 5 percent from 2016 to 2018.

Let's refine this approach. These queries sum visits recorded in each table. But from the row counts we ran earlier in the chapter, we know that each table contains a different number of library agencies: 9,261 in 2018; 9,245 in 2017; and 9,252 in 2016. The differences are likely due to agencies opening, closing, or merging. So, let's determine how the sum of visits will differ if we limit the analysis to library agencies that exist in all three tables and have a non-negative value for visits. We can do that by joining the tables, as shown in [*Listing 9-12*](#).

```
| SELECT sum(pls18.visits) AS visits_2018,
      sum(pls17.visits) AS visits_2017,
      sum(pls16.visits) AS visits_2016
?
| FROM pls_fy2018_libraries pls18
```

```

        JOIN pls_fy2017_libraries pls17 ON pls18.fscskey =
pls17.fscskey
        JOIN pls_fy2016_libraries pls16 ON pls18.fscskey =
pls16.fscskey
} WHERE pls18.visits >= 0
        AND pls17.visits >= 0
        AND pls16.visits >= 0;

```

[Listing 9-12](#): Using `sum()` to total visits on joined 2018, 2017, and 2016 tables

This query pulls together a few concepts we covered in earlier chapters, including table joins. At the top, we use the `sum()` aggregate function **1** to total the `visits` columns from each of the three tables. When we join the tables on the tables' primary keys, we're declaring table aliases **2** as we explored in Chapter 7—and here, we're omitting the optional `AS` keyword in front of each alias. For example, we declare `pls18` as the alias for the 2018 table to avoid having to write its lengthier full name throughout the query.

Note that we use a standard `JOIN`, also known as an `INNER JOIN`, meaning the query results will only include rows where the values in the `fscskey` primary key match in all three tables.

As we did in [Listing 9-11](#), we specify with a `WHERE` clause **3** that the result should include only those `rows` where `visits` are greater than or equal to 0 in the tables. This will prevent the artificial negative values from impacting the sums.

Run the query. The results should look like this:

visits_2018	visits_2017	visits_2016
1278148838	1319325387	1355078384

The results are similar to what we found by querying the tables separately, although these totals are as much as 14 million smaller in 2018. Still, the downward trend holds.

For a full picture of how library use is changing, we'd want to run a similar query on all of the columns that contain performance indicators to chronicle the trend in each. For example, the column `wifisess` shows how many times users connected to the library's wireless internet. If we use `wifisess` instead of `visits` in [Listing 9-11](#), we get this result:

wifi_2018	wifi_2017	wifi_2016
349767271	311336231	234926102

So, though visits were down, libraries saw a sharp increase in Wi-Fi network use. That provides a keen insight into how the role of libraries is changing.

NOTE

Although we joined the tables on fscskey, it's entirely possible that some library agencies that appear in all three tables merged or split during the three years. A call to the IMLS asking about caveats for working with this data is a good idea.

Grouping Visit Sums by State

Now that we know library visits dropped for the United States as a whole between 2016 and 2018, you might ask yourself, “Did every part of the country see a decrease, or did the degree of the trend vary by region?” We can answer this question by modifying our preceding query to group by the state code. Let’s also use a percent-change calculation to compare the trend by state. [Listing 9-13](#) contains the full code.

```
| SELECT pls18.stabr,
|       sum(pls18.visits) AS visits_2018,
|       sum(pls17.visits) AS visits_2017,
|       sum(pls16.visits) AS visits_2016,
|       round( (sum(pls18.visits)::numeric) - sum(pls17.visits))
|
|   2    sum(pls17.visits) * 100, 1 ) AS chg_2018_17,
|       round( (sum(pls17.visits)::numeric) - sum(pls16.visits))
|
|           sum(pls16.visits) * 100, 1 ) AS chg_2017_16
FROM pls_fy2018_libraries pls18
JOIN pls_fy2017_libraries pls17 ON pls18.fscskey =
pls17.fscskey
JOIN pls_fy2016_libraries pls16 ON pls18.fscskey =
pls16.fscskey
WHERE pls18.visits >= 0
AND pls17.visits >= 0
AND pls16.visits >= 0
```

```
3 GROUP BY pls18.stabr  
4 ORDER BY chg_2018_17 DESC;
```

[Listing 9-13](#): Using GROUP BY to track percent change in library visits by state

We follow the `SELECT` keyword with the `stabr` column **1** from the 2018 table; that same column appears in the `GROUP BY` clause **3**. It doesn't matter which table's `stabr` column we use because we're only querying agencies that appear in all three tables. After the `visits` columns, we include the now-familiar percent-change calculation you learned in Chapter 6. We use this twice, giving the aliases `chg_2018_17` **2** and `chg_2017_16` for clarity. We end the query with an `ORDER BY` clause **4**, sorting by the `chg_2018_17` column alias.

When you run the query, the top of the results shows 10 states with an increase in visits from 2017 to 2018. The rest of the results show a decline. American Samoa, at the bottom of the ranking, had a 28 percent drop!

stabr	visits_2018	visits_2017	visits_2016	chg_2018_17
	chg_2017_16			
<hr/>				
SD	3824804	3699212	3722376	3.4
-0.6				
MT	4332900	4215484	4298268	2.8
-1.9				
FL	68423689	66697122	70991029	2.6
-6.0				
ND	2216377	2162189	2201730	2.5
-1.8				
ID	8179077	8029503	8597955	1.9
-6.6				
DC	3632539	3593201	3930763	1.1
-8.6				
ME	6746380	6731768	6811441	0.2
-1.2				
NH	7045010	7028800	7236567	0.2
-2.9				
UT	15326963	15295494	16096911	0.2
-5.0				
DE	4122181	4117904	4125899	0.1
-0.2				
OK	13399265	13491194	13112511	-0.7
2.9				
WY	3338772	3367413	3536788	-0.9

-4 .8				
MA	39926583	40453003	40427356	-1 .3
0 .1				
WA	37338635	37916034	38634499	-1 .5
-1 .9				
MN	22952388	23326303	24033731	-1 .6
-2 .9				
--snip--				
GA	26835701	28816233	27987249	-6 .9
3 .0				
AR	9551686	10358181	10596035	-7 .8
-2 .2				
GU	75119	81572	71813	-7 .9
13 .6				
MS	7602710	8581994	8915406	-11 .4
-3 .7				
HI	3456131	4135229	4490320	-16 .4
-7 .9				
AS	48828	67848	63166	-28 .0
7 .4				

It's helpful, for context, to also see the percent change in visits from 2016 to 2017. Many of the states, such as Minnesota, show consecutive declines. Others, including several at the top of the list, show gains after substantial decreases the year prior.

This is when it's a good idea investigate what's driving the changes. Data analysis can sometimes raise as many questions as it answers, but that's part of the process. It's always worth a phone call to a person who works closely with the data to review your findings. Sometimes, they'll have a good explanation. Other times, an expert will say, "That doesn't sound right." That answer might send you back to the keeper of the data or the documentation to find out if you overlooked a code or a nuance with the data.

Filtering an Aggregate Query Using HAVING

To refine our analysis, we can examine a subset of states and territories that share similar characteristics. With percent change in visits, it makes sense to separate large states from small states. In a small state like Rhode Island, a single library closing for six months for repairs could have a significant effect. A single closure in California might be scarcely noticed in a statewide count. To look at states with a similar volume in visits, we could sort the results by either

of the visits columns, but it would be cleaner to get a smaller result set by filtering our query.

To filter the results of aggregate functions, we need to use the HAVING clause that's part of standard ANSI SQL. You're already familiar with using WHERE for filtering, but aggregate functions, such as sum(), can't be used within a WHERE clause because they operate at the row level, and aggregate functions work across rows. The HAVING clause places conditions on groups created by aggregating. The code in [Listing 9-14](#) modifies the query in [Listing 9-13](#) by inserting the HAVING clause after GROUP BY.

```
SELECT pls18.stabr,
       sum(pls18.visits) AS visits_2018,
       sum(pls17.visits) AS visits_2017,
       sum(pls16.visits) AS visits_2016,
       round( (sum(pls18.visits::numeric) - sum(pls17.visits))
/
           sum(pls17.visits) * 100, 1 ) AS chg_2018_17,
       round( (sum(pls17.visits::numeric) - sum(pls16.visits))
/
           sum(pls16.visits) * 100, 1 ) AS chg_2017_16
FROM pls_fy2018_libraries pls18
    JOIN pls_fy2017_libraries pls17 ON pls18.fscskey =
pls17.fscskey
    JOIN pls_fy2016_libraries pls16 ON pls18.fscskey =
pls16.fscskey
WHERE pls18.visits >= 0
      AND pls17.visits >= 0
      AND pls16.visits >= 0
GROUP BY pls18.stabr
| HAVING sum(pls18.visits) > 50000000
ORDER BY chg_2018_17 DESC;
```

[Listing 9-14](#): Using a HAVING clause to filter the results of an aggregate query

In this case, we've set our query results to include only rows with a sum of visits in 2018 greater than 50 million. That's an arbitrary value I chose to show only the very largest states. Adding the HAVING clause 1 reduces the number of rows in the output to just six. In practice, you might experiment with various values. Here are the results:

stabr	visits_2018	visits_2017	visits_2016	chg_2018_17	chg_2017_16
-------	-------------	-------------	-------------	-------------	-------------

FL	68423689	66697122	70991029	2.6
-6.0				
NY	97921323	100012193	103081304	-2.1
-3.0				
CA	146656984	151056672	155613529	-2.9
-2.9				
IL	63466887	66166082	67336230	-4.1
-1.7				
OH	68176967	71895854	74119719	-5.2
-3.0				
TX	66168387	70514138	70975901	-6.2
-0.7				

All but one of the six states experienced a decline in visits, but notice that the percent-change variation isn't as wide as in the full set of states and territories. Depending on what we learn from library experts, looking at the states with the most activity as a group might be helpful in describing trends, as would looking at other groupings. Think of a sentence you might write that would say, “Among states with the most library visits, Florida was the only one to see an increase in activity between 2017 and 2018; the rest saw visits decrease between 2 percent and 6 percent.” You could write similar sentences about medium-sized states and small states.

Wrapping Up

If you're now inspired to visit your local library and check out a couple of books, ask a librarian whether their branch has seen a rise or drop in visits over the last few years. You can probably guess the answer. In this chapter, you learned how to use standard SQL techniques to summarize data in a table by grouping values and using a handful of aggregate functions. By joining datasets, you were able to identify some interesting trends.

You also learned that data doesn't always come perfectly packaged. The presence of negative values in columns, used as an indicator rather than as an actual numeric value, forced us to filter out those rows. Unfortunately, those sorts of challenges are part of the data analyst's everyday world, so we'll spend the next chapter learning how to clean up a dataset that has a number of issues. Later in the book, you'll also discover more aggregate functions to help you find the stories in your data.

TRY IT YOURSELF

Put your grouping and aggregating skills to the test with these challenges:

We saw that library visits have declined recently in most places. But what is the pattern in library employment? All three library survey tables contain the column `totstaff`, which is the number of paid full-time equivalent employees. Modify the code in Listings 9-13 and 9-14 to calculate the percent change in the sum of the column over time, examining all states as well as states with the most visitors. Watch out for negative values!

The library survey tables contain a column called `obereg`, a two-digit Bureau of Economic Analysis Code that classifies each library agency according to a region of the United States, such as New England, Rocky Mountains, and so on. Just as we calculated the percent change in visits grouped by state, do the same to group percent changes in visits by US region using `obereg`. Consult the survey documentation to find the meaning of each region code. For a bonus challenge, create a table with the `obereg` code as the primary key and the region name as text, and join it to the summary query to group by the region name rather than the code.

Thinking back to the types of joins you learned in Chapter 7, which join type will show you all the rows in all three tables, including those without a match? Write such a query and add an `IS NULL` filter in a `WHERE` clause to show agencies not included in one or more of the tables.

10

INSPECTING AND MODIFYING DATA



If I were to propose a toast to a newly minted class of data analysts, I'd raise my glass and say, "May your data arrive perfectly structured and free of errors!" In reality, you'll sometimes receive data in such a sorry state that it's hard to analyze without modifying it.

This is called *dirty data*, a general label for data with errors, missing values, or poor organization that makes standard queries ineffective. In this chapter, you'll use SQL to clean a set of dirty data and perform other useful maintenance tasks to make data workable.

Dirty data can have multiple origins. Converting data from one file type to another or giving a column the wrong data type can cause information to be lost. People also can be careless when inputting or editing data, leaving behind typos and spelling inconsistencies. Whatever the cause may be, dirty data is the bane of the data analyst.

You'll learn how to examine data to assess its quality and how to modify data and tables to make analysis easier. But the techniques you'll learn will be useful for more than just cleaning data. The ability to make changes to data and tables gives you options for updating or adding new information to your database as it becomes available, elevating your database from a static collection to a living record.

Let's begin by importing our data.

Importing Data on Meat, Poultry, and Egg Producers

For this example, we'll use a directory of US meat, poultry, and egg producers. The Food Safety and Inspection Service (FSIS), an agency within the US Department of Agriculture, compiles and updates this database regularly. The FSIS is responsible for inspecting animals and food at more than 6,000 meat processing plants, slaughterhouses, farms, and the like. If inspectors find a problem, such as bacterial contamination or mislabeled food, the agency can issue a recall. Anyone interested in agriculture business, food supply chain, or outbreaks of foodborne illnesses will find the directory useful. Read more about the agency on its site at <https://www.fsis.usda.gov/>.

The data we'll use comes from <https://www.data.gov/>, a website run by the US federal government that catalogs thousands of datasets from various federal agencies (<https://catalog.data.gov/dataset/fsis-meat-poultry-and-egg-inspection-directory-by-establishment-name/>). I've converted the Excel file posted on the site to CSV format, and you'll find a link to the file *MPI_Directory_by_Establishment_Name.csv* along with other resources for this book at <https://nostarch.com/practical-sql-2nd-edition/>.

NOTE

Because FSIS updates the data regularly, you will see different results than those shown in this chapter if you download directly from <https://www.data.gov/>.

To import the file into PostgreSQL, use the code in [Listing 10-1](#) to create a table called `meat_poultry_egg_establishments` and use `COPY` to add the CSV file to the table. As in previous examples, use pgAdmin to connect to your analysis database, and then open the Query Tool to run the code. Remember to change the path in the `COPY` statement to reflect the location of your CSV file.

```
CREATE TABLE meat_poultry_egg_establishments (
    establishment_number text CONSTRAINT est_number_key
    PRIMARY KEY,
    company text,
    street text,
```

```

    city text,
    st text,
    zip text,
    phone text,
    grant_date date,
2 activities text,
    dbas text
);

3 COPY meat_poultry_egg_establishments
  FROM
    'C:\YourDirectory\MPI_Directory_by_Establishment_Name.csv'
  WITH (FORMAT CSV, HEADER);

4 CREATE INDEX company_idx ON meat_poultry_egg_establishments
  (company);

```

[Listing 10-1: Importing the FSIS Meat, Poultry, and Egg Inspection Directory](#)

The table has 10 columns. We add a natural primary key constraint to the establishment_number column **1**, which will hold unique values that identify each establishment. Most of the remaining columns relate to the company's name and location. You'll use the activities column **2**, which describes activities at the company, in the "Try It Yourself" exercise at the end of this chapter. We set most columns to text. In PostgreSQL, text is a varying length data type that affords us up to 1GB of data (see Chapter 4). The column dbas contains strings of more than 1,000 characters in its rows, so we're prepared to handle that. We import the CSV file **3** and then create an index on the company column **4** to speed up searches for particular companies.

For practice, let's use the count() aggregate function introduced in Chapter 9 to check how many rows are in the meat_poultry_egg_establishments table:

```
SELECT count(*) FROM meat_poultry_egg_establishments;
```

The result should show 6,287 rows. Now let's find out what the data contains and determine whether we can glean useful information from it as is, or if we need to modify it in some way.

Interviewing the Dataset

Interviewing data is my favorite part of analysis. We interview a dataset to discover its details—what it holds, what questions it can answer, and how suitable it is for our purposes—the same way a job interview reveals whether a candidate has the skills required.

The aggregate queries from Chapter 9 are a useful interviewing tool because they often expose the limitations of a dataset or raise questions you may want to ask before drawing conclusions and assuming the validity of your findings.

For example, the `meat_poultry_egg_establishments` table's rows describe food producers. At first glance, we might assume that each company in each row operates at a distinct address. But it's never safe to assume in data analysis, so let's check using the code in [Listing 10-2](#).

```
SELECT company,
       street,
       city,
       st,
       count(*) AS address_count
  FROM meat_poultry_egg_establishments
 GROUP BY company, street, city, st
 HAVING count(*) > 1
 ORDER BY company, street, city, st;
```

[Listing 10-2](#): Finding multiple companies at the same address

Here, we group companies by unique combinations of the `company`, `street`, `city`, and `st` columns. Then we use `count(*)`, which returns the number of rows for each combination of those columns and gives it the alias `address_count`. Using the `HAVING` clause introduced in Chapter 9, we filter the results to show only cases where more than one row has the same combination of values. This should return all duplicate addresses for a company.

The query returns 23 rows, which means there are close to two dozen cases where the same company is listed multiple times at the same address:

company	street	city
st	address_count	
--	-----	-----
Acre Station Meat Farm	17076 Hwy 32 N	Pinetown

```
NC          2
Beltex Corporation      3801 North Grove Street   Fort
Worth TX                2
Cloverleaf Cold Storage 111 Imperial Drive     Sanford
NC          2
--snip--
```

This is not necessarily a problem. There may be valid reasons for a company to appear multiple times at the same address. For example, two types of processing plants could exist with the same name. On the other hand, we may have found data entry errors. Either way, it's a wise practice to eliminate concerns about the validity of a dataset before relying on it, and this result should prompt us to investigate individual cases before we draw conclusions. However, this dataset has other issues that we need to look at before we can get meaningful information from it. Let's work through a few examples.

Checking for Missing Values

Next, we'll check whether we have values from all states and whether any rows are missing a state code by asking a basic question: How many meat, poultry, and egg processing companies are there in each state? We'll use the aggregate function `count()` along with `GROUP BY` to determine this, as shown in [Listing 10-3](#).

```
SELECT st,
       count(*) AS st_count
  FROM meat_poultry_egg_establishments
 GROUP BY st
 ORDER BY st;
```

[Listing 10-3: Grouping and counting states](#)

The query is a simple count that tallies the number of times each state postal code (`st`) appears in the table. Your result should include 57 rows, grouped by the state postal code in the column `st`. Why more than the 50 US states? Because the data includes Puerto Rico and other unincorporated US territories, such as Guam and American Samoa. Alaska (`AK`) is at the top of the results with a count of 17 establishments:

st	st_count
--	-----

```
AK          17
AL          93
AR          87
AS           1
--snip--
WA         139
WI          184
WV           23
WY           1
            3
```

However, the row at the bottom of the list has a `NULL` value in the `st` column and a `3` in `st_count`. That means three rows have a `NULL` in `st`. To see the details of those facilities, let's query those rows.

NOTE

Depending on the database implementation, `NULL` values will appear either first or last in a sorted column. In PostgreSQL, they appear last by default. The ANSI SQL standard doesn't specify one or the other, but it lets you add `NULLS FIRST` or `NULLS LAST` to an `ORDER BY` clause to specify a preference. For example, to make `NULL` values appear first in the preceding query, the clause would read `ORDER BY st NULLS FIRST`.

In [Listing 10-4](#), we add a `WHERE` clause with the `st` column and the `IS NULL` keywords to find which rows are missing a state code.

```
SELECT establishment_number,
       company,
       city,
       st,
       zip
  FROM meat_poultry_egg_establishments
 WHERE st IS NULL;
```

[Listing 10-4](#): Using `IS NULL` to find missing values in the `st` column

This query returns three rows that don't have a value in the `st` column:

est_number	company	city
st	zip	

```
--  
V18677A          Atlas Inspection, Inc.      Blaine  
55449  
M45319+P45319   Hall-Namie Packing Company, Inc  
36671  
M263A+P263A+V263A Jones Dairy Farm  
53538
```

That's a problem, because any counts that include the `st` column will be incorrect, such as the number of establishments per state. When you spot an error such as this, it's worth making a quick visual check of the original file you downloaded. Unless you're working with files in the gigabyte range, you can usually open a CSV file in one of the text editors I noted in Chapter 1 and search for the row. If you're working with larger files, you might be able to examine the source data using utilities such as `grep` (on Linux and macOS) or `findstr` (on Windows). In this case, a visual check of the file from <https://www.data.gov/> confirms that, indeed, there was no state listed in those rows in the file, so the error is organic to the data, not one introduced during import.

In our interview of the data so far, we've discovered that we'll need to add missing values to the `st` column to clean up this table. Let's look at what other issues exist in our dataset and make a list of cleanup tasks.

Checking for Inconsistent Data Values

Inconsistent data is another factor that can hamper our analysis. We can check for inconsistently entered data within a column by using `GROUP BY` with `count()`. When you scan the unduplicated values in the results, you might be able to spot variations in the spelling of names or other attributes.

For example, many of the 6,200 companies in our table are multiple locations owned by just a few multinational food corporations, such as Cargill or Tyson Foods. To find out how many locations each company owns, we count the values in the `company` column. Let's see what happens when we do, using the query in [Listing 10-5](#).

```
SELECT company,  
       count(*) AS company_count  
FROM meat_poultry_egg_establishments
```

```
GROUP BY company  
ORDER BY company ASC;
```

[Listing 10-5](#): Using `GROUP BY` and `count()` to find inconsistent company names

Scrolling through the results reveals a number of cases in which a company's name is spelled in several different ways. For example, notice the entries for the Armour-Eckrich brand:

company	company_count
<hr/>	
--snip--	
Armour - Eckrich Meats, LLC	1
Armour-Eckrich Meats LLC	3
Armour-Eckrich Meats, Inc.	1
Armour-Eckrich Meats, LLC	2
--snip--	

At least four different spellings are shown for seven establishments that are likely owned by the same company. If we later perform any aggregation by company, it would help to standardize the names so all the items counted or summed are grouped properly. Let's add that to our list of items to fix.

Checking for Malformed Values Using `length()`

It's a good idea to check for unexpected values in a column that should be consistently formatted. For example, each entry in the `zip` column in the `meat_poultry_egg_establishments` table should be formatted in the style of US ZIP codes with five digits. However, that's not what is in our dataset.

Solely for the purpose of this example, I replicated a common error I've committed before. When I converted the original Excel file to a CSV file, I stored the ZIP code in the default "General" number format instead of as a text value, and any ZIP code that begins with a zero lost its leading zero because an integer can't start with a zero. As a result, 07502 appears in the table as 7502. You can make this error in a variety of ways, including by copying and pasting data into Excel columns set to "General." After being burned a few times, I learned to take extra caution with numbers that should be formatted as text.

My deliberate error appears when we run the code in [Listing 10-6](#). The example introduces `length()`, a *string function* that counts the number of characters in a

string. We combine `length()` with `count()` and `GROUP BY` to determine how many rows have five characters in the `zip` field and how many have a value other than five. To make it easy to scan the results, we use `length()` in the `ORDER BY` clause.

```
SELECT length(zip),
       count(*) AS length_count
  FROM meat_poultry_egg_establishments
 GROUP BY length(zip)
 ORDER BY length(zip) ASC;
```

[Listing 10-6](#): Using `length()` and `count()` to test the `zip` column

The results confirm the formatting error. As you can see, 496 of the ZIP codes are four characters long, and 86 are three characters long, which likely means these numbers originally had two leading zeros that my conversion erroneously eliminated:

length	length_count
3	86
4	496
5	5705

Using the `WHERE` clause, we can see which states these shortened ZIP codes correspond to, as shown in [*Listing 10-7*](#).

```
SELECT st,
       count(*) AS st_count
  FROM meat_poultry_egg_establishments
 WHERE length(zip) < 5
 GROUP BY st
 ORDER BY st ASC;
```

[Listing 10-7](#): Filtering with `length()` to find short `zip` values

We use the `length()` function inside the `WHERE` clause **1** to return a count of rows where the ZIP code is less than five characters for each state code. The result is what we would expect. The states are largely in the Northeast region of the United States where ZIP codes often start with a zero:

st	st_count
--	-----
CT	55
MA	101
ME	24
NH	18
NJ	244
PR	84
RI	27
VI	2
VT	27

Obviously, we don't want this error to persist, so we'll add it to our list of items to correct. So far, we need to correct the following issues in our dataset:

Missing values for three rows in the st column

Inconsistent spelling of at least one company's name

Inaccurate ZIP codes due to file conversion

Next, we'll look at how to use SQL to fix these issues by modifying your data.

Modifying Tables, Columns, and Data

Almost nothing in a database, from tables to columns and the data types and values they contain, is set in concrete after it's created. As your needs change, you can use SQL to add columns to a table, change data types on existing columns, and edit values. Given the issues we discovered in the meat_poultry_egg_establishments table, being able to modify our database will come in handy.

We'll use two SQL commands. The first, `ALTER TABLE`, is part of the ANSI SQL standard and provides options to `ADD COLUMN`, `ALTER COLUMN`, and `DROP COLUMN`, among others.

NOTE

Typically, PostgreSQL and other databases include implementation-specific extensions to ALTER TABLE that provide an array of options for managing database objects (see <https://www.postgresql.org/docs/current/sql-altertable.html>). For our exercises, we'll stick with the core options.

The second command, UPDATE, also included in the SQL standard, allows you to change values in a table's columns. You can supply criteria using WHERE to choose which rows to update.

Let's explore the basic syntax and options for both commands and then use them to fix the issues in our dataset.

WHEN TO TOSS YOUR DATA

If your interview of the data reveals too many missing values or values that defy common sense—such as numbers ranging in the billions when you expected thousands—it's time to reevaluate your use of it. The data may not be reliable enough to serve as the foundation of your analysis.

If you suspect as much, the first step is to revisit the original data file. Make sure you imported it correctly and that values in all the source columns are located in the same columns in the table. You might need to open the original spreadsheet or CSV file and do a visual comparison. The second step is to call the agency or company that produced the data to confirm what you see and seek an explanation. You might also ask for advice from others who have used the same data.

More than once I've had to toss a dataset after determining that it was poorly assembled or simply incomplete. Sometimes, the amount of work required to make a dataset usable undermines its usefulness. These situations require you to make a tough judgment call. But it's better to start over or find an alternative than to use bad data that can lead to faulty conclusions.

Modifying Tables with ALTER TABLE

We can use the `ALTER TABLE` statement to modify the structure of tables. The following examples show standard ANSI SQL syntax for common operations, starting with the code for adding a column to a table:

```
ALTER TABLE table ADD COLUMN column data_type;
```

We can remove a column with the following syntax:

```
ALTER TABLE table DROP COLUMN column;
```

To change the data type of a column, we would use this code:

```
ALTER TABLE table ALTER COLUMN column SET DATA TYPE data_type;
```

We add a NOT NULL constraint to a column like so:

```
ALTER TABLE table ALTER COLUMN column SET NOT NULL;
```

Note that in PostgreSQL and some other systems, adding a constraint to the table causes all rows to be checked to see whether they comply with the constraint. If the table has millions of rows, this could take a while.

Removing the NOT NULL constraint looks like this:

```
ALTER TABLE table ALTER COLUMN column DROP NOT NULL;
```

When you execute `ALTER TABLE` with the placeholders filled in, you should see a message that reads `ALTER TABLE` in the pgAdmin output screen. If an operation violates a constraint or if you attempt to change a column's data type and the existing values in the column won't conform to the new data type, PostgreSQL returns an error. But PostgreSQL won't give you any warning about deleting data when you drop a column, so use extra caution before dropping a column.

Modifying Values with UPDATE

The `UPDATE` statement, part of the ANSI SQL standard, modifies the data in a column that meets a condition. It can be applied to all rows or a subset of rows. Its basic syntax for updating the data in every row in a column follows this form:

```
UPDATE table
SET column = value;
```

We first pass UPDATE the name of the table. Then to SET we pass the column we want to update. The new *value* to place in the column can be a string, number, the name of another column, or even a query or expression that generates a value. The new value must be compatible with the column data type.

We can update values in multiple columns by adding additional columns and source values and separating each with a comma:

```
UPDATE table
SET column_a = value,
    column_b = value;
```

To restrict the update to particular rows, we add a WHERE clause with some criteria that must be met before the update can happen, such as rows where values equal a date or match a string:

```
UPDATE table
SET column = value
WHERE criteria;
```

We can also update one table with values from another table. Standard ANSI SQL requires that we use a *subquery*, a query inside a query, to specify which values and rows to update:

```
UPDATE table
SET column = (SELECT column
                FROM table_b
                WHERE table.column = table_b.column)
WHERE EXISTS (SELECT column
                FROM table_b
                WHERE table.column = table_b.column);
```

The value portion of SET, inside the parentheses, is a subquery. A SELECT statement inside parentheses generates the values for the update by joining columns in both tables on matching row values. Similarly, the WHERE EXISTS clause uses a SELECT statement to ensure that we only update rows where both tables have matching values. If we didn't use WHERE EXISTS, we might

inadvertently set some values to NULL without planning to. (If this syntax looks somewhat complicated, that's okay. I'll cover subqueries in detail in Chapter 13.)

Some database managers offer additional syntax for updating across tables. PostgreSQL supports the ANSI standard but also a simpler syntax using a FROM clause:

```
UPDATE table
SET column = table_b.column
FROM table_b
WHERE table.column = table_b.column;
```

When you execute an UPDATE statement, you'll get a message stating UPDATE along with the number of rows affected.

Viewing Modified Data with RETURNING

If you add an optional RETURNING clause to UPDATE, you can view the values that were modified without having to run a second, separate query. The syntax of the clause uses the RETURNING keyword followed by a list of columns or a wildcard in the same manner that we name columns following SELECT. Here's an example:

```
UPDATE table
SET column_a = value
RETURNING column_a, column_b, column_c;
```

Instead of just noting the number of rows modified, RETURNING directs the database to show the columns you specify for the rows modified. This is a PostgreSQL-specific implementation that you also can use with INSERT and DELETE FROM. We'll try it with some of our examples.

Creating Backup Tables

Before modifying a table, it's a good idea to make a copy for reference and backup in case you accidentally destroy some data. [Listing 10-8](#) shows how to use a variation of the familiar CREATE TABLE statement to make a new table from the table we want to duplicate.

```
CREATE TABLE meat_poultry_egg_establishments_backup AS
SELECT * FROM meat_poultry_egg_establishments;
```

[Listing 10-8](#): Backing up a table

The result should be a pristine copy of your table with the new specified name. You can confirm this by counting the number of records in both tables at once:

```
SELECT
    (SELECT count(*) FROM meat_poultry_egg_establishments) AS original,
    (SELECT count(*) FROM
meat_poultry_egg_establishments_backup) AS backup;
```

The results should return the same count from both tables, like this:

original	backup
-----	-----
6287	6287

If the counts match, you can be sure your backup table is an exact copy of the structure and contents of the original table. As an added measure and for easy reference, we'll use `ALTER TABLE` to make copies of column data within the table we're updating.

NOTE

Indexes are not copied when creating a table backup using the `CREATE TABLE` statement. If you decide to run queries on the backup, be sure to create a separate index on that table.

Restoring Missing Column Values

The query in [Listing 10-4](#) earlier revealed that three rows in the `meat_poultry_egg_establishments` table don't have a value in the `st` column:

est_number	company	city
st	zip	-----
-----	-----	-----
V18677A	Atlas Inspection, Inc.	Blaine
55449		

M45319+P45319	Hall-Namie Packing Company, Inc
36671	
M263A+P263A+V263A	Jones Dairy Farm
53538	

To get a complete count of establishments in each state, we need to fill those missing values using an UPDATE statement.

Creating a Column Copy

Even though we've backed up this table, let's take extra caution and make a copy of the `st` column within the table so we still have the original data if we make some dire error somewhere. Let's create the copy and fill it with the existing `st` column values as in [Listing 10-9](#).

```
| ALTER TABLE meat_poultry_egg_establishments ADD COLUMN st_copy
|   text;
|
| UPDATE meat_poultry_egg_establishments
? SET st_copy = st;
```

[Listing 10-9](#): Creating and filling the `st_copy` column with `ALTER TABLE` and `UPDATE`

The `ALTER TABLE` statement 1 adds a column called `st_copy` using the same `text` data type as the original `st` column. Next, the `SET` clause 2 in `UPDATE` fills our new `st_copy` column with the values in column `st`. Because we don't specify any criteria using `WHERE`, values in every row are updated, and PostgreSQL returns the message `UPDATE 6287`. Again, it's worth noting that on a very large table, this operation could take some time and also substantially increase the table's size. Making a column copy in addition to a table backup isn't entirely necessary, but if you're the patient, cautious type, it can be worthwhile.

We can confirm the values were copied properly with a simple `SELECT` query on both columns, as in [Listing 10-10](#).

```
SELECT st,
       st_copy
  FROM meat_poultry_egg_establishments
```

```
WHERE st IS DISTINCT FROM st_copy
ORDER BY st;
```

[Listing 10-10](#): Checking values in the st and st_copy columns

To check for differences between values in the columns, we use `IS DISTINCT FROM` in the `WHERE` clause. You've used `DISTINCT` before to find unique values in a column (Chapter 3); in this context, `IS DISTINCT FROM` tests whether values in `st` and `st_copy` are different. This keeps us from having to scan every row ourselves. Running this query will return zero rows, meaning the values match throughout the table.

NOTE

Because `IS DISTINCT FROM` treats `NULL` as a known value, comparisons between values always will evaluate to `true` or `false`. That's different than the `<>` operator, in which a comparison that includes a `NULL` will return `NULL`. Run `SELECT 'a' <> NULL;` to see this behavior.

Now, with our original data safely stored, we can update the three rows with missing state codes. This is now our in-table backup, so if something goes drastically wrong while we're updating the original column, we can easily copy the original data back in. I'll show you how after we apply the first updates.

Updating Rows Where Values Are Missing

To update those rows' missing values, we first find the values we need with a quick online search: Atlas Inspection is located in Minnesota; Hall-Namie Packing is in Alabama; and Jones Dairy is in Wisconsin. We add those states to the appropriate rows in [Listing 10-11](#).

```
UPDATE meat_poultry_egg_establishments
SET st = 'MN'
| WHERE establishment_number = 'V18677A';

UPDATE meat_poultry_egg_establishments
SET st = 'AL'
WHERE establishment_number = 'M45319+P45319';

UPDATE meat_poultry_egg_establishments
```

```
SET st = 'WI'  
WHERE establishment_number = 'M263A+P263A+V263A'  
? RETURNING establishment_number, company, city, st, zip;
```

[Listing 10-11](#): Updating the st column for three establishments

Because we want each UPDATE statement to affect a single row, we include a WHERE clause **1** for each that identifies the company's unique establishment_number, which is the table's primary key. When we run the first two queries, PostgreSQL responds with the message UPDATE 1, showing that only one row was updated for each query. When we run the third, the RETURNING clause **2** directs the database to show several columns from the row that was updated:

establishment_number	company	city	st	zip
M263A+P263A+V263A	Jones Dairy Farm		WI	53538

If we rerun the code in [Listing 10-4](#) to find rows where st is NULL, the query should return nothing. Success! Our count of establishments by state is now complete.

Restoring Original Values

What happens if we botch an update by providing the wrong values or updating the wrong rows? We'll just copy the data back from either the full table backup or the column backup. [Listing 10-12](#) shows the two options.

```
| UPDATE meat_poultry_egg_establishments  
|   SET st = st_copy;  
  
? UPDATE meat_poultry_egg_establishments original  
?   SET st = backup.st  
?   FROM meat_poultry_egg_establishments_backup backup  
?   WHERE original.establishment_number =  
?         backup.establishment_number;
```

[Listing 10-12](#): Restoring original st column values

To restore the values from the backup column in meat_poultry_egg_establishments, run an UPDATE query 1 that sets st to the values in st_copy. Both columns should again have the identical original values. Alternatively, you can create an UPDATE 2 that sets st to values in the st column from the meat_poultry_egg_establishments_backup table you made in [Listing 10-8](#). This will obviate the fixes you made to add missing state values, so if you want to try this query, you'll need to redo the fixes using [Listing 10-11](#).

Updating Values for Consistency

In [Listing 10-5](#), we discovered several cases where a single company's name was entered inconsistently. These inconsistencies will hinder us if we want to aggregate data by company name, so we'll fix them.

Here are the spelling variations of Armour-Eckrich Meats in [Listing 10-5](#):

```
--snip--  
Armour - Eckrich Meats, LLC  
Armour-Eckrich Meats LLC  
Armour-Eckrich Meats, Inc.  
Armour-Eckrich Meats, LLC  
--snip--
```

We can standardize the spelling using an UPDATE statement. To protect our data, we'll create a new column for the standardized spellings, copy the names in company into the new column, and work in the new column. [Listing 10-13](#) has the code for both actions.

```
ALTER TABLE meat_poultry_egg_establishments ADD COLUMN  
company_standard text;  
  
UPDATE meat_poultry_egg_establishments  
SET company_standard = company;
```

[Listing 10-13](#): Creating and filling the company_standard column

Now, let's say we want any name in company that starts with the string Armour to appear in company_standard as Armour-Eckrich Meats. (This assumes we've checked all Armour entries and want to standardize them.) With [Listing 10-14](#), we can update all the rows matching the string Armour using WHERE.

```
UPDATE meat_poultry_egg_establishments
SET company_standard = 'Armour-Eckrich Meats'
| WHERE company LIKE 'Armour%'
? RETURNING company, company_standard;
```

Listing 10-14: Using an UPDATE statement to modify column values that match a string

The important piece of this query is the WHERE clause that uses the LIKE keyword 1 for case-sensitive pattern matching introduced in Chapter 3. Including the wildcard syntax % at the end of the string Armour updates all rows that start with those characters regardless of what comes after them. The clause lets us target all the varied spellings used for the company's name. The RETURNING clause 2 causes the statement to provide the results of the updated company_standard column next to the original company column:

company	company_standard
-----	-----
Armour-Eckrich Meats LLC	Armour-Eckrich Meats
Armour - Eckrich Meats, LLC	Armour-Eckrich Meats
Armour-Eckrich Meats LLC	Armour-Eckrich Meats
Armour-Eckrich Meats LLC	Armour-Eckrich Meats
Armour-Eckrich Meats, Inc.	Armour-Eckrich Meats
Armour-Eckrich Meats, LLC	Armour-Eckrich Meats
Armour-Eckrich Meats, LLC	Armour-Eckrich Meats

The values for Armour-Eckrich in company_standard are now standardized with consistent spelling. To standardize other company names in the table, we would create an UPDATE statement for each case. We would also keep the original company column for reference.

Repairing ZIP Codes Using Concatenation

Our final fix repairs values in the zip column that lost leading zeros. Zip codes in Puerto Rico and the US Virgin Islands begin with two zeros, so we need to restore two leading zeros to the values in zip. For the other states, located mostly in New England, we'll restore a single leading zero.

We'll use UPDATE in conjunction with the double-pipe *string concatenation operator* (||). Concatenation combines two string values into one (it will also combine a string and a number into a string). For example, inserting || between the strings abc and xyz results in abcxxyz. The double-pipe operator is a SQL

standard for concatenation supported by PostgreSQL. You can use it in many contexts, such as UPDATE queries and SELECT, to provide custom output from existing as well as new data.

First, [Listing 10-15](#) makes a backup copy of the zip column as we did earlier.

```
ALTER TABLE meat_poultry_egg_establishments ADD COLUMN
zip_copy text;

UPDATE meat_poultry_egg_establishments
SET zip_copy = zip;
```

[Listing 10-15](#): Creating and filling the zip_copy column

Next, we use the code in [Listing 10-16](#) to perform the first update.

```
UPDATE meat_poultry_egg_establishments
| SET zip = '00' || zip
? WHERE st IN('PR', 'VI') AND length(zip) = 3;
```

[Listing 10-16](#): Modifying codes in the zip column missing two leading zeros

We use SET to set the value in the zip column **1** to the result of the concatenation of 00 and the existing value. We limit the UPDATE to only those rows where the st column has the state codes PR and VI **2** using the IN comparison operator from Chapter 3 and add a test for rows where the length of zip is 3. This entire statement will then only update the zip values for Puerto Rico and the Virgin Islands. Run the query; PostgreSQL should return the message UPDATE 86, which is the number of rows we expect to change based on our earlier count in [Listing 10-6](#).

Let's repair the remaining ZIP codes using a similar query in [Listing 10-17](#).

```
UPDATE meat_poultry_egg_establishments
SET zip = '0' || zip
WHERE st IN('CT', 'MA', 'ME', 'NH', 'NJ', 'RI', 'VT') AND
length(zip) = 4;
```

[Listing 10-17](#): Modifying codes in the zip column missing one leading zero

PostgreSQL should return the message UPDATE 496. Now, let's check our progress. Earlier in [Listing 10-6](#), when we aggregated rows in the `zip` column by length, we found 86 rows with three characters and 496 with four.

Using the same query now returns a more desirable result: all the rows have a five-digit ZIP code.

length	count
5	6287

I'll discuss additional string functions in Chapter 14 when we consider advanced techniques for working with text.

Updating Values Across Tables

In “Modifying Values with UPDATE” earlier in the chapter, I showed the standard ANSI SQL and PostgreSQL-specific syntax for updating values in one table based on values in another. This syntax is particularly valuable in a relational database where primary keys and foreign keys establish table relationships. In those cases, we may need information in one table to update values in another table.

Let's say we're setting an inspection deadline for each of the companies in our table. We want to do this by US regions, such as Northeast, Pacific, and so on, but those regional designations don't exist in our table. However, they *do* exist in the file `state_regions.csv`, included with the book's resources, that contains matching `st` state codes. Once we load that file into a table, we can use that data in an UPDATE statement. Let's begin with the New England region to see how this works.

Enter the code in [Listing 10-18](#), which contains the SQL statements to create a `state_regions` table and fill the table with data:

```
CREATE TABLE state_regions (
    st text CONSTRAINT st_key PRIMARY KEY,
    region text NOT NULL
);

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

[Listing 10-18](#): Creating and filling a state_regions table

We'll create two columns in a `state_regions` table: one containing the two-character state code `st` and the other containing the region name. We set the primary key constraint to the `st` column, which holds a unique `st_key` value to identify each state. In the data you're importing, each state is present and assigned to a census region, and territories outside the United States are labeled as outlying areas. We'll update the table one region at a time.

Next, let's return to the `meat_poultry_egg_establishments` table, add a column for inspection dates, and then fill in that column with the New England states. [*Listing 10-19*](#) shows the code.

```
ALTER TABLE meat_poultry_egg_establishments
    ADD COLUMN inspection_deadline timestamp with time zone;

| UPDATE meat_poultry_egg_establishments establishments
? SET inspection_deadline = '2022-12-01 00:00 EST'
3 WHERE EXISTS (SELECT state_regions.region
                  FROM state_regions
                 WHERE establishments.st = state_regions.st
                   AND state_regions.region = 'New England');
```

[Listing 10-19](#): Adding and updating an inspection_deadline column

The `ALTER TABLE` statement creates the `inspection_deadline` column in the `meat_poultry_egg_establishments` table. In the `UPDATE` statement, we give the table an alias of `establishments` to make the code easier to read **1** (and do so omitting the optional `AS` keyword). Next, `SET` assigns a timestamp value of `2022-12-01 00:00 EST` to the new `inspection_deadline` column **2**. Finally, `WHERE EXISTS` includes a subquery that connects the `meat_poultry_egg_establishments` table to the `state_regions` table we created in [*Listing 10-18*](#) and specifies which rows to update **3**. The subquery (in parentheses, beginning with `SELECT`) looks for rows in the `state_regions` table where the `region` column matches the string `New England`. At the same time, it joins the `meat_poultry_egg_establishments` table with the `state_regions` table using the `st` column from both tables. In effect, the query is telling the database to find all the `st` codes that correspond to the `New England` region and use those codes to filter the update.

When you run the code, you should receive a message of UPDATE 252, which is the number of companies in New England states. You can use the code in [*Listing 10-20*](#) to see the effect of the change.

```
SELECT st, inspection_deadline
FROM meat_poultry_egg_establishments
GROUP BY st, inspection_deadline
ORDER BY st;
```

[*Listing 10-20*](#): Viewing updated `inspection_date` values

The results should show the updated inspection deadlines for all New England companies. The top of the output shows Connecticut has received a deadline timestamp, for example, but states outside New England remain NULL because we haven't updated them yet:

st	inspection_deadline
--	-----
--snip--	
CA	
CO	
CT	2022-12-01 00:00:00-05
DC	
--snip--	

To fill in deadlines for additional regions, substitute a different region for New England in [*Listing 10-19*](#) and rerun the query.

Deleting Unneeded Data

The most irrevocable way to modify data is to remove it entirely. SQL includes options to remove rows and columns along with options to delete an entire table or database. We want to perform these operations with caution, removing only data or tables we don't need. Without a backup, your data is gone for good.

NOTE

It's easy to exclude unwanted data in queries using a WHERE clause, so decide whether you truly need to delete the data or can just filter it out. Cases where deleting may be the best solution include data with errors, data imported incorrectly, or almost no disk space.

In this section, we'll use a variety of SQL statements to delete data. If you didn't back up the meat_poultry_egg_establishments table using [Listing 10-8](#), now is a good time to do so.

Writing and executing these statements is fairly simple, but doing so comes with a caveat. If deleting rows, a column, or a table would cause a violation of a constraint, such as the foreign key constraint covered in Chapter 8, you need to deal with that constraint first. That might involve removing the constraint, deleting data in another table, or deleting another table. Each case is unique and will require a different way to work around the constraint.

Deleting Rows from a Table

To remove rows from a table, we can use either `DELETE FROM` or `TRUNCATE`, which are both part of the ANSI SQL standard. Each offers options that are useful depending on your goals.

Using `DELETE FROM`, we can remove all rows from a table, or we can add a `WHERE` clause to delete only the portion that matches an expression we supply. To delete all rows from a table, use the following syntax:

```
DELETE FROM table_name;
```

To remove only selected rows, add a `WHERE` clause along with the matching value or pattern to specify which ones you want to delete:

```
DELETE FROM table_name WHERE expression;
```

For example, to exclude US territories from our processors table, we can remove the companies in those locations using the code in [Listing 10-21](#).

```
DELETE FROM meat_poultry_egg_establishments  
WHERE st IN('AS','GU','MP','PR','VI');
```

[Listing 10-21](#): Deleting rows matching an expression

Run the code; PostgreSQL should return the message `DELETE 105`. This means the 105 rows where the `st` column held any of the codes designating a territory that you supplied via the `IN` keyword have been removed from the table.

With large tables, using `DELETE FROM` to remove all rows can be inefficient because it scans the entire table as part of the process. In that case, you can use `TRUNCATE`, which skips the scan. To empty the table using `TRUNCATE`, use the following syntax:

```
TRUNCATE table_name;
```

A handy feature of `TRUNCATE` is the ability to reset an `IDENTITY` sequence, such as one you may have created to serve as a surrogate primary key, as part of the operation. To do that, add the `RESTART IDENTITY` keywords to the statement:

```
TRUNCATE table_name RESTART IDENTITY;
```

We'll skip truncating any tables for now as we need the data for the rest of the chapter.

Deleting a Column from a Table

Earlier we created a backup `zip` column called `zip_copy`. Now that we've finished working on fixing the issues in `zip`, we no longer need `zip_copy`. We can remove the backup column, including all the data within the column, from the table using the `DROP` keyword in the `ALTER TABLE` statement.

The syntax for removing a column is similar to other `ALTER TABLE` statements:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

The code in [Listing 10-22](#) removes the `zip_copy` column:

```
ALTER TABLE meat_poultry_egg_establishments DROP COLUMN  
zip_copy;
```

[*Listing 10-22*](#): Removing a column from a table using `DROP`

PostgreSQL returns the message `ALTER TABLE`, and the `zip_copy` column should be deleted. The database doesn't actually rewrite the table to remove the column; it just marks the column as deleted in its internal catalog and no longer shows it or adds data to it when new rows are added.

Deleting a Table from a Database

The `DROP TABLE` statement is a standard ANSI SQL feature that deletes a table from the database. This statement might come in handy if, for example, you have a collection of backups, or *working tables*, that have outlived their usefulness. It's also useful when you need to change the structure of a table significantly; in that case, rather than using too many `ALTER TABLE` statements, you can just remove the table and create a fresh one by running a new `CREATE TABLE` statement and re-importing the data.

The syntax for the `DROP TABLE` command is simple:

```
DROP TABLE table_name;
```

For example, [*Listing 10-23*](#) deletes the backup version of the `meat_poultry_egg_establishments` table.

```
DROP TABLE meat_poultry_egg_establishments_backup;
```

[*Listing 10-23*](#): Removing a table from a database using `DROP`

Run the query; PostgreSQL should respond with the message `DROP TABLE` to indicate the table has been removed.

Using Transactions to Save or Revert Changes

So far, our alterations in this chapter have been final. That is, after you run a `DELETE` or `UPDATE` query (or any other query that alters your data or database structure), the only way to undo the change is to restore from a backup. However, there is a way to check your changes before finalizing them and cancel the change if it's not what you intended. You do this by enclosing the SQL statement within a *transaction*, which includes keywords that allow you to

commit your changes if they are successful or roll them back if not. You define a transaction using the following keywords at the beginning and end of the query:

START TRANSACTION Signals the start of the transaction block. In PostgreSQL, you can also use the non-ANSI SQL `BEGIN` keyword.

COMMIT Signals the end of the block and saves all changes.

ROLLBACK Signals the end of the block and reverts all changes.

You can include multiple statements between `BEGIN` and `COMMIT` to define a sequence of operations that perform one unit of work in a database. An example is when you buy concert tickets, which might involve two steps: charging your credit card and reserving your seats so someone else can't buy them. A database programmer would want either both steps in the transaction to happen (say, when your card charge goes through) or neither to happen (if you cancel at checkout). Defining both steps as one transaction—also called a *transaction block*—keeps them as a unit; if one step is canceled or throws an error, the other gets canceled too. You can learn more details about transactions and PostgreSQL at <https://www.postgresql.org/docs/current/tutorial-transactions.html>.

We can use a transaction block to review changes a query makes and then decide whether to keep or discard them. In our table, let's say we're cleaning dirty data related to the company AGRO Merchants Oakland LLC. The table has three rows listing the company, but one row has an extra comma in the name:

Company

AGRO Merchants Oakland LLC
AGRO Merchants Oakland LLC
AGRO Merchants Oakland, LLC

We want the name to be consistent, so we'll remove the comma from the third row using an `UPDATE` query, as we did earlier. But this time we'll check the result of our update before we make it final (and we'll purposely make a mistake we want to discard). [Listing 10-24](#) shows how to do this using a transaction block.

```
| START TRANSACTION;
| UPDATE meat_poultry_egg_establishments
```

```
2 SET company = 'AGRO Merchantss Oakland LLC'  
    WHERE company = 'AGRO Merchants Oakland, LLC';  
  
3 SELECT company  
    FROM meat_poultry_egg_establishments  
    WHERE company LIKE 'AGRO%'  
    ORDER BY company;  
  
4 ROLLBACK;
```

[Listing 10-24](#): Demonstrating a transaction block

Beginning with `START TRANSACTION; 1`, we'll run each statement separately. The database responds with the message `START TRANSACTION`, letting you know that any succeeding changes you make to data will not be made permanent unless you issue a `COMMIT` command. Next, we run the `UPDATE` statement, which changes the company name in the row where it has an extra comma. I intentionally added an extra `s` in the name used in the `SET` clause **2** to introduce a mistake.

When we view the names of companies starting with the letters `AGRO` using the `SELECT` statement **3**, we see that, oops, one company name is misspelled now.

Company

AGRO Merchants Oakland LLC
AGRO Merchants Oakland LLC
AGRO Merchantss Oakland LLC

Instead of rerunning the `UPDATE` statement to fix the typo, we can simply discard the change by running the `ROLLBACK; 4` command. When we rerun the `SELECT` statement to view the company names, we're back to where we started:

Company

AGRO Merchants Oakland LLC
AGRO Merchants Oakland LLC
AGRO Merchants Oakland, LLC

From here, you correct your `UPDATE` statement by removing the extra `s` and rerun it, beginning with the `START TRANSACTION` statement again. If you're

happy with the changes, run `COMMIT;` to make them permanent.

NOTE

When you start a transaction in PostgreSQL, any changes you make to the data aren't visible to other database users until you execute `COMMIT`. Other databases may behave differently depending on their settings.

Transaction blocks are often used for more complex situations rather than checking simple changes. Here you've used them to test whether a query behaves as desired, saving you time and headaches. Next, let's look at another way to save time when updating lots of data.

Improving Performance When Updating Large Tables

With PostgreSQL, adding a column to a table and filling it with values can quickly inflate the table's size because the database creates a new version of the existing row each time a value is updated, but it doesn't delete the old row version. That essentially doubles the table's size. (You'll learn how to clean up these old rows when I discuss database maintenance in "Recovering Unused Space with VACUUM" in Chapter 19.) For small datasets, the increase is negligible, but for tables with hundreds of thousands or millions of rows, the time required to update rows and the resulting extra disk usage can be substantial.

Instead of adding a column and filling it with values, we can save disk space by copying the entire table and adding a populated column during the operation. Then, we rename the tables so the copy replaces the original, and the original becomes a backup. Thus, we have a fresh table without the added old rows.

[Listing 10-25](#) shows how to copy `meat_poultry_egg_establishments` into a new table while adding a populated column. To do this, if you didn't already drop the `meat_poultry_egg_establishments_backup` table as shown in [Listing 10-23](#), go ahead and drop it. Then run the `CREATE TABLE` statement.

```
CREATE TABLE meat_poultry_egg_establishments_backup AS
| SELECT *,
```

```
2 '2023-02-14 00:00 EST'::timestamp with time zone AS
reviewed_date
FROM meat_poultry_egg_establishments;
```

[Listing 10-25](#): Backing up a table while adding and filling a new column

The query is a modified version of the backup script in [Listing 10-8](#). Here, in addition to selecting all the columns using the asterisk wildcard **1**, we also add a column called `reviewed_date` by providing a value cast as a `timestamp` data type **2** and the `AS` keyword. That syntax adds and fills `reviewed_date`, which we might use to track the last time we checked the status of each plant.

Then we use [Listing 10-26](#) to swap the table names.

```
1 ALTER TABLE meat_poultry_egg_establishments
    RENAME TO meat_poultry_egg_establishments_temp;
2 ALTER TABLE meat_poultry_egg_establishments_backup
    RENAME TO meat_poultry_egg_establishments;
3 ALTER TABLE meat_poultry_egg_establishments_temp
    RENAME TO meat_poultry_egg_establishments_backup;
```

[Listing 10-26](#): Swapping table names using `ALTER TABLE`

Here we use `ALTER TABLE` with a `RENAME TO` clause to change a table name. The first statement changes the original table name to one that ends with `_temp` **1**. The second statement renames the copy we made with [Listing 10-24](#) to the original name of the table **2**. Finally, we rename the table that ends with `_temp` to the ending `_backup` **3**. The original table is now called `meat_poultry_egg_establishments_backup`, and the copy with the added column is called `meat_poultry_egg_establishments`. This process avoids updating rows and thus inflating the table.

Wrapping Up

Gleaning useful information from data sometimes requires modifying the data to remove inconsistencies, fix errors, and make it more suitable for supporting an accurate analysis. In this chapter you learned some useful tools to help you assess dirty data and clean it up. In a perfect world, all datasets would arrive

with everything clean and complete. But such a perfect world doesn't exist, so the ability to alter, update, and delete data is indispensable.

Let me restate the important tasks of working safely. Be sure to back up your tables before you start making changes. Make copies of your columns, too, for an extra level of protection. When I discuss database maintenance for PostgreSQL later in the book, you'll learn how to back up entire databases. These few steps of precaution will save you a world of pain.

In the next chapter, we'll return to math to explore some of SQL's advanced statistical functions and techniques for analysis.

TRY IT YOURSELF

In this exercise, you'll turn the `meat_poultry_egg_establishments` table into useful information. You need to answer two questions: how many of the plants in the table process meat, and how many process poultry?

The answers to these two questions lie in the `activities` column. Unfortunately, the column contains an assortment of text with inconsistent input. Here's an example of the kind of text you'll find in the `activities` column:

Poultry Processing, Poultry Slaughter
Meat Processing, Poultry Processing
Poultry Processing, Poultry Slaughter

The mishmash of text makes it impossible to perform a typical count that would allow you to group processing plants by activity. However, you can make some modifications to fix this data. Your tasks are as follows:

Create two new columns called `meat_processing` and `poultry_processing` in your table. Each can be of the type `boolean`.

Using `UPDATE`, set `meat_processing = TRUE` on any row in which the `activities` column contains the text *Meat Processing*. Do the same update on the `poultry_processing` column, but this time look for the text *Poultry Processing* in `activities`.

Use the data from the new, updated columns to count how many plants perform each type of activity. For a bonus challenge, count how many plants perform both activities.

11

STATISTICAL FUNCTIONS IN SQL



In this chapter, we'll explore SQL statistical functions along with guidelines for using them. A SQL database usually isn't the first tool a data analyst chooses when they need to do more than calculate sums and averages.

Typically, the software of choice is a full-featured statistics package, such as SPSS or SAS, the programming languages R or Python, or even Excel. But you don't have to discount your database. Standard ANSI SQL, including PostgreSQL's implementation, offers powerful stats functions and capabilities that reveal a lot about your data without having to export your dataset to another program.

Statistics is a vast subject worthy of its own book, so we'll only skim the surface here. Nevertheless, you'll learn how to apply high-level statistical concepts to help you derive meaning from your data using a new dataset from the US Census Bureau. You'll also learn to use SQL to create rankings, calculate rates using data about business establishments, and smooth out time-series data using rolling averages and sums.

Creating a Census Stats Table

Let's return to one of my favorite data sources, the US Census Bureau. This time, you'll use county data from the 2014–2018 American Community Survey

(ACS) 5-Year Estimates, another product from the bureau.

Use the code in [Listing 11-1](#) to create the table `acs_2014_2018_stats` and import the CSV file `acs_2014_2018_stats.csv`. The code and data are available with all the book's resources via <https://nostarch.com/practical-sql-2nd-edition/>. Remember to change `C:\YourDirectory\` to the location of the CSV file.

```
CREATE TABLE acs_2014_2018_stats (
    1 geoid text CONSTRAINT geoid_key PRIMARY KEY,
    county text NOT NULL,
    st text NOT NULL,
    2 pct_travel_60_min numeric(5,2),
    pct_bachelors_higher numeric(5,2),
    pct_masters_higher numeric(5,2),
    median_hh_income integer,
    3 CHECK (pct_masters_higher <= pct_bachelors_higher)
);

COPY acs_2014_2018_stats
FROM 'C:\YourDirectory\acs_2014_2018_stats.csv'
WITH (FORMAT CSV, HEADER);

| SELECT * FROM acs_2014_2018_stats;
```

[Listing 11-1](#): Creating a 2014–2018 ACS 5-Year Estimates table and importing data

The `acs_2014_2018_stats` table has seven columns. The first three 1 include a unique `geoid` that serves as the primary key, the name of the `county`, and the state name `st`. Both `county` and `st` carry the `NOT NULL` constraint because each row should contain a value. The next four columns display certain percentages 2 I derived for each county from estimates in the ACS release, plus one more economic indicator:

`pct_travel_60_min`

The percentage of workers ages 16 and older who commute more than 60 minutes to work.

`pct_bachelors_higher`

The percentage of people ages 25 and older whose level of education is a bachelor's degree or higher. (In the United States, a bachelor's degree is usually awarded upon completing a four-year college education.)

pct_masters_higher

The percentage of people ages 25 and older whose level of education is a master's degree or higher. (In the United States, a master's degree is the first advanced degree earned after completing a bachelor's degree.)

median_hh_income

The county's median household income in 2018 inflation-adjusted dollars. As you learned in Chapter 6, a median value is the midpoint in an ordered set of numbers, where half the values are larger than the midpoint and half are smaller. Because averages can be skewed by a few very large or very small values, government reporting on economic data, such as income, tends to use medians.

We include a `CHECK` constraint **3** to ensure that the figures for the bachelor's degree are equal to or higher than those for the master's degree, because in the United States, a bachelor's degree is earned before or concurrently with a master's degree. A county showing the opposite could indicate data imported incorrectly or a column mislabeled. Our data checks out: upon import, there are no errors showing a violation of the `CHECK` constraint.

We use the `SELECT` statement **4** to view all 3,142 rows imported, each corresponding to a county surveyed in this census release.

Next, we'll use statistics functions in SQL to better understand the relationships among the percentages.

THE US CENSUS: ESTIMATES VS. THE COMPLETE COUNT

Each US Census Bureau data product has its own methodology. The Decennial Census, the most well-known, is a full count of the US population conducted every 10 years using forms mailed to each household in the country and visits by census workers. One of its primary purposes is to determine the number of seats each state holds in the US House of Representatives. The census population estimates we've used build off the decennial count and use births, deaths, migration, and other factors to create population totals for the years between the decennial counts.

In contrast, the American Community Survey is an ongoing annual survey of about 3.5 million US households. It asks about topics including income, education, employment, ancestry, and housing. Private and public organizations use ACS data to track trends that drive decision-making. Currently, the US Census Bureau packages ACS data into two releases: a one-year dataset with estimates for geographies with populations of 65,000 or more, and a five-year dataset that includes all geographies. Because it's a survey, ACS results are estimates and have a margin of error, which I've omitted for brevity but which you'll see included in a full ACS dataset.

Measuring Correlation with `corr(Y, X)`

Correlation describes the statistical relationship between two variables, measuring the extent to which a change in one is associated with a change in the other. In this section, we'll use the SQL `corr(Y, X)` function to measure what relationship exists, if any, between the percentage of people in a county who've attained a bachelor's degree and the median household income in that county. We'll also determine whether, according to our data, a better-educated population typically equates to higher income and, if it does, the strength of that relationship.

First, some background. The *Pearson correlation coefficient* (generally denoted as r) measures the strength and direction of a *linear relationship* between two variables. Variables that have a strong linear relationship cluster along a line when graphed on a scatterplot. The Pearson value of r falls between -1 and 1 ; either end of the range indicates a perfect correlation, whereas values near zero

indicate a random distribution with little correlation. A positive r value indicates a *direct relationship*: as one variable increases, the other does too. When graphed, the data points representing each pair of values in a direct relationship would slope upward from left to right. A negative r value indicates an *inverse relationship*: as one variable increases, the other decreases. Dots representing an inverse relationship would slope downward from left to right on a scatterplot.

[Table 11-1](#) provides general guidelines for interpreting positive and negative r values, although different statisticians may offer different interpretations.

Table 11-1: Interpreting Correlation Coefficients

Correlation coefficient (+/-)	What it could mean
0	No relationship
.01 to .29	Weak relationship
.3 to .59	Moderate relationship
.6 to .99	Strong to nearly perfect relationship
1	Perfect relationship

In standard ANSI SQL and PostgreSQL, we calculate the Pearson correlation coefficient using `corr(y, x)`. It's one of several *binary aggregate functions* in SQL and is so named because these functions accept two inputs. The input y is the *dependent variable* whose variation depends on the value of another variable, and x is the *independent variable* whose value doesn't depend on another variable.

NOTE

Even though SQL specifies the y and x inputs for the corr() function, correlation calculations don't distinguish between dependent and independent variables. Switching the order of inputs in corr() produces the same result. However, for convenience and readability, these examples order the input variables according to dependent and independent.

We'll use `corr(y, x)` to discover the relationship between education level and income, with income as our dependent variable and education as our independent variable. Enter the code in [Listing 11-2](#) to use `corr(y, x)` with `median_hh_income` and `pct_bachelors_higher` as inputs.

```
SELECT corr(median_hh_income, pct_bachelors_higher)
      AS bachelors_income_r
FROM acs_2014_2018_stats;
```

[Listing 11-2](#): Using $\text{corr}(Y, X)$ to measure the relationship between education and income

Run the query; your result should be an r value of just below 0.70 given as the floating-point double precision data type:

```
bachelors_income_r
-----
0.6999086502599159
```

This positive r value indicates that as a county's educational attainment increases, household income tends to increase. The relationship isn't perfect, but the r value shows the relationship is fairly strong. We can visualize this pattern by plotting the variables on a scatterplot using Excel, as shown in [Figure 11-1](#). Each data point represents one US county; the data point's position on the x-axis shows the percentage of the population ages 25 and older that has a bachelor's degree or higher. The data point's position on the y-axis represents the county's median household income.

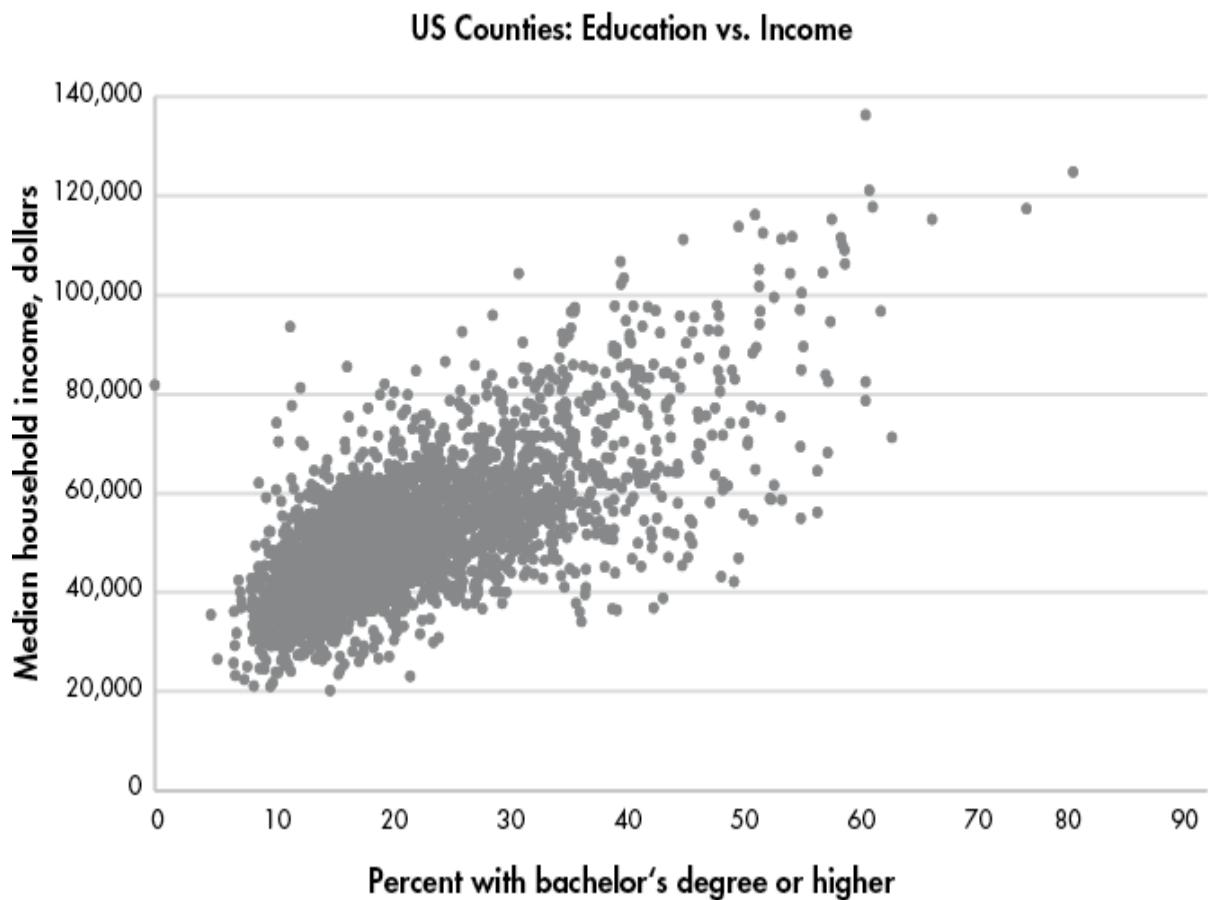


Figure 11-1: A scatterplot showing the relationship between education and income

Notice that although most of the data points are grouped together in the bottom-left corner of the graph, they do generally slope upward from left to right. Also, the points spread out rather than strictly follow a straight line. If they were in a straight line sloping up from left to right, the r value would be 1, indicating a perfect positive linear relationship.

Checking Additional Correlations

Now let's calculate the correlation coefficients for the remaining variable pairs using the code in [Listing 11-3](#).

```

SELECT
  1 round(
    corr(median_hh_income, pct_bachelors_higher)::numeric, 2
  )
  
```

```

        ) AS bachelors_income_r,
    round(
        corr(pct_travel_60_min, median_hh_income)::numeric, 2
    ) AS income_travel_r,
    round(
        corr(pct_travel_60_min, pct_bachelors_higher)::numeric,
2
    ) AS bachelors_travel_r
FROM acs_2014_2018_stats;

```

[Listing 11-3](#): Using `corr(Y, X)` on additional variables

This time we'll round off the decimal values to make the output more readable by wrapping the `corr(Y, X)` function inside SQL's `round()` function 1, which takes two inputs: the numeric value to be rounded and an integer value indicating the number of decimal places to round the first value. If the second parameter is omitted, the value is rounded to the nearest whole integer. Because `corr(Y, X)` returns a floating-point value by default, we cast it to the `numeric` type using the `::` notation you learned in Chapter 4. Here's the output:

bachelors_income_r	income_travel_r	bachelors_travel_r
0.70	0.06	-0.14

The `bachelors_income_r` value is 0.70, which is the same as our first run but rounded up to two decimal places. Compared to `bachelors_income_r`, the other two correlations are weak.

The `income_travel_r` value shows that the correlation between income and the percentage of those who commute more than an hour to work is practically zero. This indicates that a county's median household income bears little connection to how long it takes people to get to work.

The `bachelors_travel_r` value shows that the correlation of bachelor's degrees and lengthy commutes is also low at -0.14. The negative value indicates an inverse relationship: as education increases, the percentage of the population that travels more than an hour to work decreases. Although this is interesting, a correlation coefficient that is this close to zero indicates a weak relationship.

When testing for correlation, we need to note some caveats. The first is that even a strong correlation does not imply causality. We can't say that a change in

one variable causes a change in the other, only that the changes move together. The second is that correlations should be subject to testing to determine whether they're statistically significant. Those tests are beyond the scope of this book but worth studying on your own.

Nevertheless, the SQL `corr(y, x)` function is a handy tool for quickly checking correlations between variables.

Predicting Values with Regression Analysis

Researchers also want to predict values using available data. For example, let's say 30 percent of a county's population has a bachelor's degree or higher. Given the trend in our data, what would we expect that county's median household income to be? Likewise, for each percent increase in education, how much increase, on average, would we expect in income?

We can answer both questions using *linear regression*. Simply put, the regression method finds the best linear equation, or straight line, that describes the relationship between an independent variable (such as education) and a dependent variable (such as income). We can then look at points along this line to predict values where we don't have observations. Standard ANSI SQL and PostgreSQL include functions that perform linear regression.

[Figure 11-2](#) shows our previous scatterplot with a regression line added.

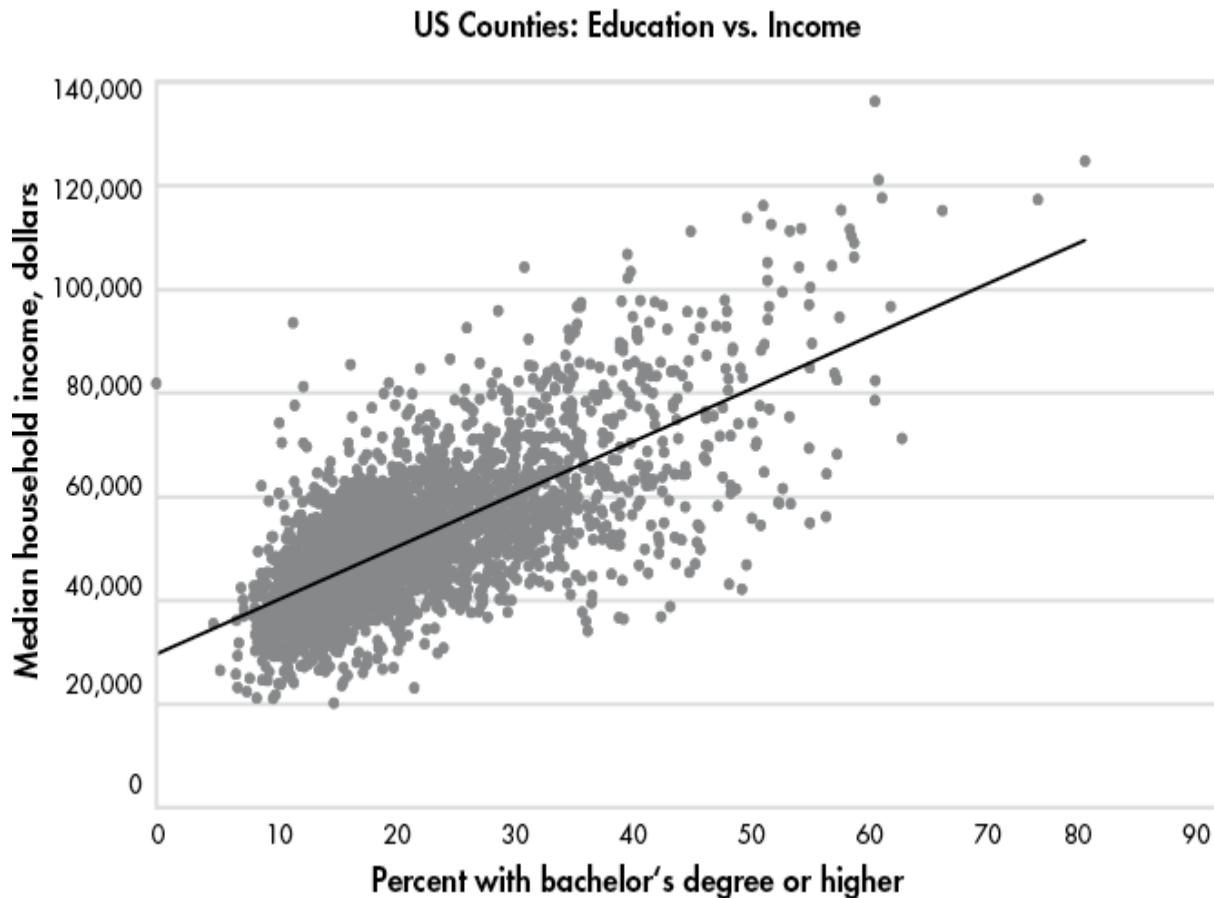


Figure 11-2: Scatterplot with least squares regression line showing the relationship between education and income

The straight line running through the middle of all the data points is called the *least squares regression line*, which approximates the “best fit” for a straight line that best describes the relationship between the variables. The equation for the regression line is like the *slope-intercept* formula you might remember from high school math but written using differently named variables: $Y = bX + a$. Here are the formula’s components:

Y is the predicted value, which is also the value on the y-axis, or dependent variable.

b is the slope of the line, which can be positive or negative. It measures how many units the y-axis value will increase or decrease for each unit of the x-axis value.

X represents a value on the x-axis, or independent variable.

a is the y-intercept, the value at which the line crosses the y-axis when the X value is zero.

Let's apply this formula using SQL. Earlier, we questioned the expected median household income in a county where than 30 percent or more of the population had a bachelor's degree. In our scatterplot, the percentage with bachelor's degrees falls along the x-axis, represented by X in the calculation. Let's plug that value into the regression line formula in place of X :

$$Y = b(30) + a$$

To calculate Y , which represents the predicted median household income, we need the line's slope, b , and the y-intercept, a . To get these values, we'll use the SQL functions `regr_slope(Y, X)` and `regr_intercept(Y, X)`, as shown in [Listing 11-4](#).

```
SELECT
    round(
        regr_slope(median_hh_income,
pct_bachelors_higher)::numeric, 2
    ) AS slope,
    round(
        regr_intercept(median_hh_income,
pct_bachelors_higher)::numeric, 2
    ) AS y_intercept
FROM acs_2014_2018_stats;
```

[Listing 11-4](#): Regression slope and intercept functions

Using the `median_hh_income` and `pct_bachelors_higher` variables as inputs for both functions, we'll set the resulting value of the `regr_slope(Y, X)` function as `slope` and the output for the `regr_intercept(Y, X)` function as `y_intercept`.

Run the query; the result should show the following:

slope	y_intercept
-----	-----
1016.55	29651.42

The `slope` value shows that for every one-unit increase in bachelor's degree percentage, we can expect a county's median household income will increase by

\$1,016.55. The `y_intercept` value shows that when the regression line crosses the y-axis, where the percentage with bachelor's degrees is at 0, the y-axis value is 29,651.42. Now let's plug both values into the equation to get our predicted value Y :

$$Y = 1016.55(30) + 29651.42 \\ Y = 60147.92$$

Based on our calculation, in a county in which 30 percent of people age 25 and older have a bachelor's degree or higher, we can expect a median household income to be about \$60,148. Of course, our data includes counties whose median income falls above and below that predicted value, but we expect this to be the case because our data points in the scatterplot don't line up perfectly along the regression line. Recall that the correlation coefficient we calculated was 0.70, indicating a strong but not perfect relationship between education and income. Other factors likely contributed to variations in income, such as the types of jobs available in each county.

Finding the Effect of an Independent Variable with r -Squared

Beyond determining the direction and strength of the relationship between two variables, we can also calculate the extent that the variation in the x (independent) variable explains the variation in the y (dependent) variable. To do this we square the r value to find the *coefficient of determination*, better known as *r-squared*. An r -squared indicates the percentage of the variation that is explained by the independent variable, and is a value between zero and one. For example, if r -squared equals 0.1, we would say that the independent variable explains 10 percent of the variation in the dependent variable, or not much at all.

To find r -squared, we use the `regr_r2(y, x)` function in SQL. Let's apply it to our education and income variables using the code in [Listing 11-5](#).

```
SELECT round(
    regr_r2(median_hh_income,
    pct_bachelors_higher)::numeric, 3
) AS r_squared
FROM acs_2014_2018_stats;
```

[Listing 11-5](#): Calculating the coefficient of determination, or r -squared

This time we'll round off the output to the nearest thousandth place and alias the result to `r_squared`. The query should return the following result:

```
r_squared
-----
0.490
```

The *r*-squared value of 0.490 indicates that about 49 percent of the variation in median household income among counties can be explained by the percentage of people with a bachelor's degree or higher in that county. Any number of factors could explain the other 51 percent, and statisticians will typically test numerous combinations of variables to determine what they are.

Before you use these numbers in a headline or presentation, it's worth revisiting the following points:

Correlation doesn't prove causality. For verification, do a Google search on "correlation and causality." Many variables correlate well but have no meaning. (See <https://www.tylervigen.com/spurious-correlations> for examples of correlations that don't prove causality, including the correlation between divorce rate in Maine and margarine consumption.) Statisticians usually perform *significance testing* on the results to make sure values are not simply the result of randomness.

Statisticians also apply additional tests to data before accepting the results of a regression analysis, including whether the variables follow the standard bell curve distribution and meet other criteria for a valid result.

Let's explore two additional concepts before wrapping up our look at statistical functions.

Finding Variance and Standard Deviation

Variance and *standard deviation* describe the degree to which a set of values varies from the average of those values. Variance, often used in finance, is the average of each number's squared distance from the average. The more dispersion in a set of values, the greater the variance. A stock market trader can use variance to measure the volatility of a particular stock—how much its daily closing values tend to vary from the average. That could indicate how risky an investment the stock might be.

Standard deviation is the square root of the variance and is most useful for assessing data whose values form a normal distribution, usually visualized as a symmetrical *bell curve*. In a *normal distribution*, about two-thirds of values fall within one standard deviation of the average; 95 percent are within two standard deviations. The standard deviation of a set of values, therefore, helps us understand how close most of our values are to the average. For example, consider a study that found the average height of adult US women is about 65.5 inches with a standard deviation of 2.5 inches. Given that heights are normally distributed, that means about two-thirds of women are within 2.5 inches of the average, or 63 inches to 68 inches tall.

When calculating variance and standard deviation, note that they report different units. Standard deviation is expressed in the same units as the values, while variance is not—it reports a number that is larger than the units, on a scale of its own.

These are the functions for calculating variance:

var_pop(numeric) Calculates the population variance of the input values. In this context, *population* refers to a dataset that contains all possible values, as opposed to a sample that just contains a portion of all possible values.

var_samp(numeric) Calculates the sample variance of the input values. Use this with data that is sampled from a population, as in a random sample survey.

For calculating standard deviation, we use these:

stddev_pop(numeric) Calculates the population standard deviation.

stddev_samp(numeric) Calculates the sample standard deviation.

With functions covering correlation, regression, and other descriptive statistics, you have a basic toolkit for obtaining a preliminary survey of your data before doing more rigorous analysis. All these topics are worth in-depth study to better understand when you might use them and what they measure. A classic, easy-to-understand resource I recommend is the book *Statistics* by David Freedman, Robert Pisani, and Roger Purves.

Creating Rankings with SQL

Rankings make the news often. You'll see them used anywhere from weekend box-office charts to sports teams' league standings. With SQL you can create numbered rankings in your query results, which are useful for tasks such as

tracking changes over several years. You can also simply use a ranking as a fact on its own in a report. Let's explore how to create rankings using SQL.

Ranking with rank() and dense_rank()

Standard ANSI SQL includes several ranking functions, but we'll just focus on two: `rank()` and `dense_rank()`. Both are *window functions*, which are defined as functions that perform calculations across a set of rows relative to the current row. Unlike aggregate functions, which combine rows to calculate values, with window functions the query first generates a set of rows, and then the window function runs across the result set to calculate the value it will return.

The difference between `rank()` and `dense_rank()` is the way they handle the next rank value after a tie: `rank()` includes a gap in the rank order, but `dense_rank()` does not. This concept is easier to understand in action, so let's look at an example. Consider a Wall Street analyst who covers the highly competitive widget manufacturing market. The analyst wants to rank companies by their annual output. The SQL statements in [Listing 11-6](#) create and fill a table with this data and then rank the companies by widget output.

```
CREATE TABLE widget_companies (
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    company text NOT NULL,
    widget_output integer NOT NULL
);

INSERT INTO widget_companies (company, widget_output)
VALUES
    ('Dom Widgets', 125000),
    ('Ariadne Widget Masters', 143000),
    ('Saito Widget Co.', 201000),
    ('Mal Inc.', 133000),
    ('Dream Widget Inc.', 196000),
    ('Miles Amalgamated', 620000),
    ('Arthur Industries', 244000),
    ('Fischer Worldwide', 201000);

SELECT
    company,
    widget_output,
    1 rank() OVER (ORDER BY widget_output DESC),
    2 dense_rank() OVER (ORDER BY widget_output DESC)
```

```
FROM widget_companies
ORDER BY widget_output DESC;
```

[Listing 11-6](#): Using the `rank()` and `dense_rank()` window functions

Notice the syntax in the `SELECT` statement that includes `rank()` 1 and `dense_rank()` 2. After the function names, we use the `OVER` clause and in parentheses place an expression that specifies the “window” of rows the function should operate on. The *window* is the set of rows relative to the current row, and in this case, we want both functions to work on all rows of the `widget_output` column, sorted in descending order. Here’s the output:

company	widget_output	rank	
dense_rank			
---		----	-----
Miles Amalgamated	620000	1	1
Arthur Industries	244000	2	2
Fischer Worldwide	201000	3	3
Saito Widget Co.	201000	3	3
Dream Widget Inc.	196000	5	4
Ariadne Widget Masters	143000	6	5
Mal Inc.	133000	7	6
Dom Widgets	125000	8	7

The columns produced by `rank()` and `dense_rank()` show each company’s ranking based on the `widget_output` value from highest to lowest, with Miles Amalgamated at number one. To see how `rank()` and `dense_rank()` differ, check the fifth-row listing, Dream Widget Inc.

With `rank()`, Dream Widget Inc. is the fifth-highest-ranking company. Because `rank()` allows a gap in the order when a tie occurs, Dream placing fifth tells us there are four companies with more output. In contrast, `dense_rank()` doesn’t allow a gap in the rank order so it places Dream Widget Inc. in fourth place. This reflects the fact that Dream has the fourth-highest widget output regardless of how many companies produced more.

Both ways of handling ties have merit, but in practice `rank()` is used most often. It’s also what I recommend using, because it more accurately reflects the total number of companies ranked, shown by the fact that Dream Widget Inc. has four companies ahead of it in total output, not three.

Let’s look at a more complex ranking example.

Ranking Within Subgroups with PARTITION BY

The ranking we just did was a simple overall ranking based on widget output. But sometimes you'll want to produce ranks within groups of rows in a table. For example, you might want to rank government employees by salary within each department or rank movies by box-office earnings within each genre.

To use window functions in this way, we'll add `PARTITION BY` to the `OVER` clause. A `PARTITION BY` clause divides table rows according to values in a column we specify.

Here's an example using made-up data about grocery stores. Enter the code in [Listing 11-7](#) to fill a table called `store_sales`.

```
CREATE TABLE store_sales (
    store text NOT NULL,
    category text NOT NULL,
    unit_sales bigint NOT NULL,
    CONSTRAINT store_category_key PRIMARY KEY (store,
category)
);

INSERT INTO store_sales (store, category, unit_sales)
VALUES
    ('Broders', 'Cereal', 1104),
    ('Wallace', 'Ice Cream', 1863),
    ('Broders', 'Ice Cream', 2517),
    ('Cramers', 'Ice Cream', 2112),
    ('Broders', 'Beer', 641),
    ('Cramers', 'Cereal', 1003),
    ('Cramers', 'Beer', 640),
    ('Wallace', 'Cereal', 980),
    ('Wallace', 'Beer', 988);

SELECT
    category,
    store,
    unit_sales,
    1 rank() OVER (PARTITION BY category ORDER BY unit_sales
DESC)
FROM store_sales
? ORDER BY category, rank() OVER (PARTITION BY category
        ORDER BY unit_sales DESC);
```

[Listing 11-7](#): Applying `rank()` within groups using `PARTITION BY`

In the table, each row includes a store's product category and sales for that category. The final `SELECT` statement creates a result set showing how each store's sales ranks within each category. The new element is the addition of `PARTITION BY` in the `OVER` clause **1**. In effect, the clause tells the program to create rankings one category at a time, using the store's unit sales in descending order.

To display the results by category and rank, we add an `ORDER BY` clause **2** that includes the `category` column and the same `rank()` function syntax. Here's the output:

category	store	unit_sales	rank
Beer	Wallace	988	1
Beer	Broders	641	2
Beer	Cramers	640	3
Cereal	Broders	1104	1
Cereal	Cramers	1003	2
Cereal	Wallace	980	3
Ice Cream	Broders	2517	1
Ice Cream	Cramers	2112	2
Ice Cream	Wallace	1863	3

Rows for each category are ordered by category unit sales with the `rank` column displaying the ranking.

Using this table, we can see at a glance how each store ranks in a food category. For instance, Broders tops sales for cereal and ice cream, but Wallace wins in the beer category. You can apply this concept to many other scenarios: for each auto manufacturer, finding the vehicle with the most consumer complaints; figuring out which month had the most rainfall in each of the last 20 years; finding the team with the most wins against left-handed pitchers; and so on.

Calculating Rates for Meaningful Comparisons

Rankings based on raw counts aren't always meaningful; in fact, they can be misleading. Consider birth statistics: the US National Center for Health Statistics (NCHS) reported that in 2019, there were 377,599 babies born in the state of Texas and 46,826 born in the state of Utah. So, women in Texas are more likely to have babies, right? Not so fast. In 2019, Texas' estimated

population was nine times as much as Utah's. Given that context, comparing the plain number of births in the two states isn't very meaningful.

A more accurate way to compare these numbers is to convert them to rates. Analysts often calculate a rate per 1,000 people, or some multiple of that number, to allow an apples-to-apples comparison. For example, the fertility rate—the number of births per 1,000 women ages 15 to 44—was 62.5 for Texas in 2019 and 66.7 for Utah, according to the NCHS. So, despite the smaller number of births, on a per-1,000 rate, women in Utah actually had more children.

The math behind this is simple. Let's say your town had 115 births and a population of 2,200 women ages 15 to 44. You can find the per-1,000 rate as follows:

$$(115 / 2,200) \times 1,000 = 52.3$$

In your town, there were 52.3 births per 1,000 women ages 15 to 44, which you can now compare to other places regardless of their size.

Finding Rates of Tourism-Related Businesses

Let's try calculating rates using SQL and census data. We'll join two tables: the census population estimates you imported in Chapter 5 plus data I compiled about tourism-related businesses from the census' County Business Patterns program. (You can read about the program methodology at <https://www.census.gov/programs-surveys/cbp/about.html>.)

[Listing 11-8](#) contains the code to create and fill the business patterns table. Remember to point the script to the location in which you've saved the CSV file *cbp_naics_72_establishments.csv*, which you can download from GitHub via the link at <https://nostarch.com/practical-sql-2nd-edition/>.

```
CREATE TABLE cbp_naics_72_establishments (
    state_fips text,
    county_fips text,
    county text NOT NULL,
    st text NOT NULL,
    naics_2017 text NOT NULL,
    naics_2017_label text NOT NULL,
    year smallint NOT NULL,
    establishments integer NOT NULL,
    CONSTRAINT cbp_fips_key PRIMARY KEY (state_fips,
```

```

        county_fips)
    );

COPY cbp_naics_72_establishments
FROM 'C:\YourDirectory\cbp_naics_72_establishments.csv'
WITH (FORMAT CSV, HEADER);

SELECT *
FROM cbp_naics_72_establishments
ORDER BY state_fips, county_fips
LIMIT 5;

```

[Listing 11-8](#): Creating and filling a table for census county business pattern data

Once you've imported the data, run the final SELECT statement to view the first few rows of the table. Each row contains descriptive information about a county along with the number of business establishments that fall under code 72 of the North American Industry Classification System (NAICS). Code 72 covers "Accommodation and Food Services" establishments, mainly hotels, inns, bars, and restaurants. The number of those businesses in a county is a good proxy for the amount of tourist and recreation activity in the area.

Let's find out which counties have the highest concentration of such businesses per 1,000 population, using the code in [Listing 11-9](#).

```

SELECT
    cbp.county,
    cbp.st,
    cbp.establishments,
    pop.pop_est_2018,
    1 round( (cbp.establishments::numeric / pop.pop_est_2018) *
1000, 1 )
        AS estabs_per_1000
FROM cbp_naics_72_establishments cbp JOIN
us_counties_pop_est_2019 pop
    ON cbp.state_fips = pop.state_fips
    AND cbp.county_fips = pop.county_fips
? WHERE pop.pop_est_2018 >= 50000
    ORDER BY cbp.establishments::numeric / pop.pop_est_2018 DESC;

```

[Listing 11-9](#): Finding business rates per thousand population in counties with 50,000 or more people

Overall, this syntax should look familiar. In Chapter 5, you learned that when dividing an integer by an integer, one of the values must be a numeric or decimal for the result to include decimal places. We do that in the rate calculation **1** with PostgreSQL's double-colon shorthand. Because we don't need many decimal places, we wrap the statement in the `round()` function to round off the output to the nearest tenth. Then we give the calculated column an alias of `estabs_per_1000` for easy reference.

Also, we use a `WHERE` clause **2** to limit our results to counties with 50,000 or more people. That's an arbitrary value that lets us see how rates compare within a group of more-populous, better-known counties. Here's a portion of the results, sorted with highest rates at top:

county estabs_per_1000	st	establishments	pop_est_2018
10.0	New Jersey	925	92446
8.7	Maryland	453	51960
7.2	Florida	540	74757
6.6	New York	427	64215
6.4	New York	10428	1629055
6.2	Maine	337	54734
5.8	Tennessee	570	97895
5.6	Colorado	309	54943
--snip--			

The counties that have the highest rates make sense. Cape May County, New Jersey, is home to numerous beach resort towns on the Atlantic Ocean and Delaware Bay. Worcester County, Maryland, contains Ocean City and other beach attractions. And Monroe County, Florida, is best known for its vacation hotspot, the Florida Keys. Sense a pattern?

Smoothing Uneven Data

A *rolling average* is an average calculated for each time period in a dataset, using a moving window of rows as input each time. Think of a hardware store: it might sell 20 hammers on Monday, 15 hammers on Tuesday, and just a few the rest of the week. The next week, hammer sales might spike on Friday. To find the big-picture story in such uneven data, we can smooth numbers by calculating the rolling average, sometimes called a *moving average*.

Here are two weeks of hammer sales at that hypothetical hardware store:

Date	Hammer sales	Seven-day average
2022-05-01	0	
2022-05-02	20	
2022-05-03	15	
2022-05-04	3	
2022-05-05	6	
2022-05-06	1	
2022-05-07	1	6.6
? 2022-05-08	2	6.9
2022-05-09	18	6.6
2022-05-10	13	6.3
2022-05-11	2	6.1
2022-05-12	4	5.9
2022-05-13	12	7.4
2022-05-14	2	7.6

Let's say that for every day we want to know the average sales over the last seven days (we can choose any period, but a week is an intuitive unit). Once we have seven days of data **1**, we calculate the average of sales over the seven-day period that includes the current day. The average of hammer sales from May 1 to May 7, 2022, is 6.6 per day.

The next day **2**, we again average sales over the most recent seven days, from May 2 to May 8, 2022. The result is 6.9 per day. As we continue each day, despite the ups and downs in the daily sales, the seven-day average remains fairly steady. Over a long period of time, we'll be able to better discern a trend.

Let's use the window function syntax again to perform this calculation using the code in [Listing 11-10](#). The code and data are available with all the book's resources in GitHub, available via <https://nostarch.com/practical-sql-2nd-edition/>. Remember to change `C:\YourDirectory\` to the location of the CSV file.

```
| CREATE TABLE us_exports (
    year smallint,
    month smallint,
    citrus_export_value bigint,
    soybeans_export_value bigint
);

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

? SELECT year, month, citrus_export_value
   FROM us_exports
   ORDER BY year, month;

! SELECT year, month, citrus_export_value,
       round(
           5 avg(citrus_export_value)
           6 OVER(ORDER BY year, month
           7 ROWS BETWEEN 11 PRECEDING AND CURRENT
ROW), 0)
       AS twelve_month_avg
   FROM us_exports
   ORDER BY year, month;
```

[Listing 11-10](#): Creating a rolling average for export data

We create a table **1** and use `COPY 2` to insert data from `us_exports.csv`. This file contains data showing the monthly dollar value of US exports of citrus fruit and soybeans, two commodities whose sales are tied to the growing season. The data comes from the US Census Bureau's international trade division at <https://usatrade.census.gov/>.

The first `SELECT` statement **3** lets you view the monthly citrus export data, which covers every month from 2002 through summer 2020. The last dozen rows should look like this:

year	month	citrus_export_value
2019	9	14012305
2019	10	26308151

--snip--

2019	11	60885676
2019	12	84873954
2020	1	110924836
2020	2	171767821
2020	3	201231998
2020	4	122708243
2020	5	75644260
2020	6	36090558
2020	7	20561815
2020	8	15510692

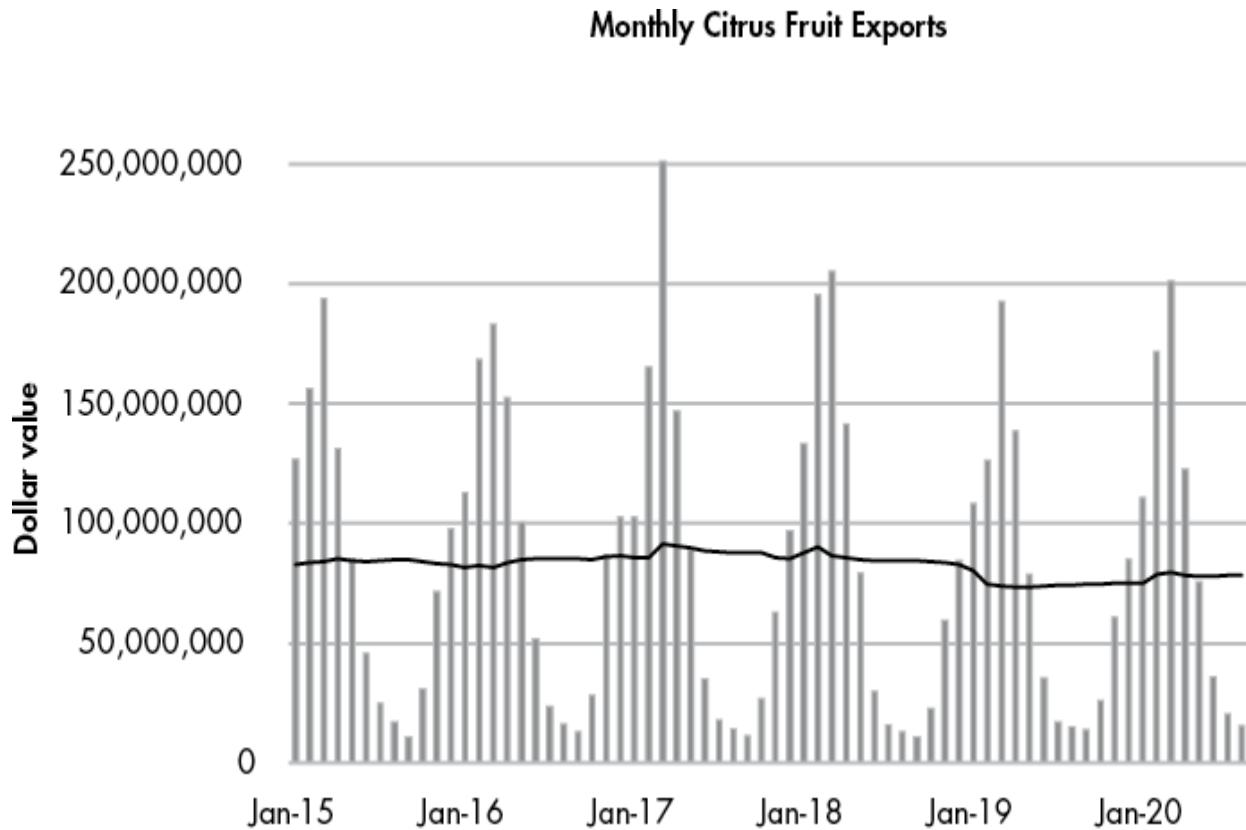
Notice the pattern: the value of citrus fruit exports is highest in winter months, when the growing season is paused in the northern hemisphere and countries need imports to meet demand. We'll use the second SELECT statement **4** to compute a 12-month rolling average so we can see, for each month, the annual trend in exports.

In the SELECT values list, we place an `avg()` **5** function to calculate the average of the values in the `citrus_export_value` column. We follow the function with an `OVER` clause **6** that has two elements in parentheses: an `ORDER BY` clause that sorts the data for the period we plan to average, and the number of rows to average, using the keywords `ROWS BETWEEN 11 PRECEDING AND CURRENT ROW` **7**. This tells PostgreSQL to limit the window to the current row and the 11 rows before it—12 total.

We wrap the entire statement, from the `avg()` function through the `OVER` clause, in a `round()` function to limit the output to whole numbers. The last dozen rows of your query result should be as follows:

year	month	citrus_export_value	twelve_month_avg
<hr/>			
--snip--			
2019	9	14012305	74465440
2019	10	26308151	74756757
2019	11	60885676	74853312
2019	12	84873954	74871644
2020	1	110924836	75099275
2020	2	171767821	78874520
2020	3	201231998	79593712
2020	4	122708243	78278945
2020	5	75644260	77999174
2020	6	36090558	78045059
2020	7	20561815	78343206
2020	8	15510692	78376692

Notice the 12-month average is far more consistent. If we want to see the trend, it's helpful to graph the results using Excel or a stats program. [Figure 11-3](#) shows the monthly totals from 2015 through August 2020 in bars, with the 12-month average as a line.



[Figure 11-3: Monthly citrus fruit exports with 12-month rolling average](#)

Based on the rolling average, citrus fruit exports were generally steady until 2019 and then trended down before recovering slightly in 2020. It's difficult to discern that movement from the monthly data, but the rolling average makes it apparent.

The window function syntax offers multiple options for analysis. For example, instead of calculating a rolling average, you could substitute the `sum()` function to find the rolling total over a time period. If you calculated a seven-day rolling sum, you'd know the weekly total ending on any day in your dataset.

NOTE

Calculating rolling averages or sums works best when there are no breaks in the time periods in your data. A missing month, for example, will turn a 12-month sum into a 13-month sum because the window function pays attention to rows, not dates.

SQL offers additional window functions. Check the official PostgreSQL documentation at <https://www.postgresql.org/docs/current/tutorial-window.html> for an overview of window functions, and check <https://www.postgresql.org/docs/current/functions-window.html> for a listing of window functions.

Wrapping Up

Now your SQL analysis toolkit includes ways to find relationships among variables using statistical functions, create rankings from ordered data, smooth spiky data to find trends, and properly compare raw numbers by turning them into rates. That toolkit is starting to look impressive!

Next, we'll dive deeper into date and time data, using SQL functions to extract the information we need.

TRY IT YOURSELF

Test your new skills with the following questions:

In [Listing 11-2](#), the correlation coefficient, or `r` value, of the variables `pct_bachelors_higher` and `median_hh_income` was about 0.70. Write a query using the same dataset to show the correlation between `pct_masters_higher` and `median_hh_income`. Is the `r` value higher or lower? What might explain the difference?

Using the exports data, create a 12-month rolling sum using the values in the column `soybeans_export_value` and the query pattern from [Listing 11-8](#). Copy and paste the results from the pgAdmin output pane and graph the values using Excel. What trend do you see?

As a bonus challenge, revisit the libraries data in the table `pls_fy2018_libraries` in Chapter 9. Rank library agencies based on the rate of visits per 1,000 population (column `popu_1sa`), and limit the query to agencies serving 250,000 people or more.

12

WORKING WITH DATES AND TIMES



Columns filled with dates and times can indicate *when* events happened or *how long* they took, and that can lead to interesting lines of inquiry. What patterns exist in the moments on a timeline? Which events were shortest or longest? What relationships exist between a particular activity and the time of day or season in which it occurred?

In this chapter, we'll explore these kinds of questions using SQL data types for dates and times and their related functions. We'll start with a closer look at data types and functions related to dates and times. Then we'll explore a dataset on trips by New York City taxicabs to look for patterns and try to discover what, if any, story the data tells. We'll also explore time zones using Amtrak data to calculate the duration of train trips across the United States.

Understanding Data Types and Functions for Dates and Times

Chapter 4 explored primary SQL data types, but to review, here are the four data types related to dates and times:

`timestamp` Records date and time. You will almost always want to add the keywords `with time zone` to ensure that times stored include time zone information. Otherwise, times recorded 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`. You can specify time zones in three different formats: its UTC offset, an area/location designator, or a standard abbreviation. If you supply a time without a time zone to a `timestamptz` column, the database will add time zone information using your server's default setting.

date Records only the date and is part of the SQL standard. PostgreSQL accepts several date formats. For example, valid formats for adding the 21st day of September 2022 are `September 21, 2022` or `9/21/2022`. I recommend using `YYYY-MM-DD` (or `2022-09-21`), which is the ISO 8601 international standard format and also the default PostgreSQL date output. Using the ISO format helps avoid confusion when sharing data internationally.

time Records only the time and is part of the SQL standard. Adding `with time zone` makes the column time zone aware, but without a date the time zone will be meaningless. Given that, using `time with time zone` and its PostgreSQL shortcut `timetz` is strongly discouraged. The ISO 8601 format is `HH:MM:SS`, where `HH` represents the hour, `MM` the minutes, and `ss` the seconds.

interval Holds a value that represents a unit of time expressed in the format `quantity unit`. It doesn't record the start or end of a period, only its duration. Examples include 12 days or 8 hours. It's also part of the SQL standard, although PostgreSQL-specific syntax offers more options.

The first three data types, `date`, `time`, and `timestamp with time zone` (or `timestamptz`), are known as *datetime types* whose values are called *datetimes*. The `interval` value is an *interval type* whose values are *intervals*. All four data types can track the system clock and the nuances of the calendar. For example, `date` and `timestamp with time zone` recognize that June has 30 days. If you try to use June 31, PostgreSQL will display an error: `date/time field value out of range`. Likewise, the date February 29 is valid only in a leap year, such as 2024.

Manipulating Dates and Times

We can use SQL functions to perform calculations on dates and times or extract their components. For example, we can retrieve the day of the week from a timestamp or extract just the month from a date. ANSI SQL outlines a handful of functions for this purpose, but many database managers (including

MySQL and Microsoft SQL Server) deviate from the standard to implement their own date and time data types, syntax, and function names. If you’re using a database other than PostgreSQL, check its documentation.

Let’s review how to manipulate dates and times using PostgreSQL functions.

Extracting the Components of a timestamp Value

It’s not unusual to need just one piece of a date or time value for analysis, particularly when you’re aggregating results by month, year, or even minute. We can extract these components using the PostgreSQL `date_part()` function. Its format looks like this:

```
date_part(text, value)
```

The function takes two inputs. The first is a string in `text` format that represents the part of the date or time to extract, such as `hour`, `minute`, or `week`. The second is the `date`, `time`, or `timestamp` value. To see the `date_part()` function in action, we’ll execute it multiple times on the same value using the code in [Listing 12-1](#).

```
SELECT
    date_part('year', '2022-12-01 18:37:12 EST'::timestamptz) AS year,
    date_part('month', '2022-12-01 18:37:12 EST'::timestamptz) AS month,
    date_part('day', '2022-12-01 18:37:12 EST'::timestamptz) AS day,
    date_part('hour', '2022-12-01 18:37:12 EST'::timestamptz) AS hour,
    date_part('minute', '2022-12-01 18:37:12 EST'::timestamptz) AS minute,
    date_part('seconds', '2022-12-01 18:37:12 EST'::timestamptz) AS seconds,
    date_part('timezone_hour', '2022-12-01 18:37:12 EST'::timestamptz) AS tz,
    date_part('week', '2022-12-01 18:37:12 EST'::timestamptz) AS week,
    date_part('quarter', '2022-12-01 18:37:12 EST'::timestamptz) AS quarter,
    date_part('epoch', '2022-12-01 18:37:12 EST'::timestamptz) AS epoch;
```

[Listing 12-1](#): Extracting components of a timestamp value using `date_part()`

Each column statement in this SELECT query first uses a string to name the component we want to extract: year, month, day, and so on. The second input uses the string `2022-12-01 18:37:12 EST` cast as a timestamp with time zone with the PostgreSQL double-colon syntax and the `timestamptz` shorthand. We specify that this timestamp occurs in the Eastern time zone using the Eastern Standard Time (EST) designation.

Here's the output as shown on my computer. The database converts the values to reflect your PostgreSQL time zone setting, so your output might be different; for example, if it's set to the US Pacific time zone, the hour will show as 15:

year	month	day	hour	minute	seconds	tz
week	quarter	epoch				
2022	12	1	18	37	12	-5
48	4	1669937832				

Each column contains a single component of the timestamp that represents 6:37:12 PM on December 1, 2022. The first six values are easy to recognize from the original timestamp, but the last four deserve an explanation.

In the `tz` column, PostgreSQL reports back the hours difference, or *offset*, from Coordinated Universal Time (UTC), the time standard for the world. The value of UTC is $+/- 00:00$, so -5 specifies a time zone five hours behind UTC. From November through early March, UTC -5 represents the Eastern time zone. In March, when the Eastern time zone moves to daylight saving time and clocks “spring forward” an hour, its UTC offset changes to -4 . (For a map of UTC time zones, see

https://en.wikipedia.org/wiki/Coordinated_Universal_Time#/media/File:Standard_World_Time_Zones.tif.)

NOTE

You can derive the UTC offset from the time zone but not vice versa. Each UTC offset can refer to multiple named time zones plus standard and daylight saving time variants.

The week column shows that December 1, 2022, falls in the 48th week of the year. This number is determined by ISO 8601 standards, which start each week on a Monday. A week at the end of a year can extend from December into January of the following year.

The quarter column shows that our test date is part of the fourth quarter of the year. The epoch column shows a measurement, which is used in computer systems and programming languages, that represents the number of seconds elapsed before or after 12 AM, January 1, 1970, at UTC 0. A positive value designates a time since that point; a negative value designates a time before it. In this example, 1,669,937,832 seconds elapsed between January 1, 1970, and the timestamp. Epoch can be useful for comparing two timestamps mathematically on an absolute scale.

NOTE

Proceed with caution with epoch times. PostgreSQL's `date_part()` returns epoch time as a double precision type, which is subject to floating-point computational errors (see Chapter 4). Epoch time also faces the so-called Year 2038 problem, when epoch values will grow too large for some computer systems to store.

PostgreSQL also supports the SQL-standard `extract()` function, which parses datetimes in the same way as the `date_part()` function. I've featured `date_part()` here instead for two reasons. First, its name helpfully reminds us what it does. Second, `extract()` isn't widely supported by other database managers. Most notably, it's absent in Microsoft's SQL Server. Nevertheless, if you need to use `extract()`, the syntax takes this form:

```
extract(text from value)
```

To replicate the first `date_part()` example in [Listing 12-1](#) where we pull the year from the timestamp, we'd set up `extract()` like this (note that we don't need single quotes around the time unit, in this case `year`):

```
extract(year from '2022-12-01 18:37:12 EST'::timestamptz)
```

PostgreSQL provides additional components you can extract or calculate from dates and times. For the full list of functions, see the documentation at

<https://www.postgresql.org/docs/current/functions-datetime.html>.

Creating Datetime Values from timestamp Components

It's not unusual to come across a dataset in which the year, month, and day exist in separate columns, and you might want to create a datetime value from these components. To perform calculations on a date, it's helpful to combine and format those pieces correctly into one column.

You can use the following PostgreSQL functions to make datetime objects:

make_date(year, month, day) Returns a value of type date.

make_time(hour, minute, seconds) Returns a value of type time without time zone.

make_timestamptz(year, month, day, hour, minute, second, time zone) Returns a timestamp with time zone.

The variables for these three functions take integer types as input, with two exceptions: seconds are of the type double precision because you can supply fractions of seconds, and time zones must be specified with a text string that names the time zone.

[Listing 12-2](#) shows examples of the three functions in action using components of February 22, 2022, for the date, and 6:04:30.3 PM in Lisbon, Portugal for the time.

```
SELECT make_date(2022, 2, 22);
SELECT make_time(18, 4, 30.3);
SELECT make_timestamptz(2022, 2, 22, 18, 4, 30.3,
'Europe/Lisbon');
```

[Listing 12-2: Three functions for making datetimes from components](#)

When I run each query in order, the output on my computer is as follows. Again, yours may differ depending on your PostgreSQL time zone setting:

```
2022-02-22
18:04:30.3
2022-02-22 13:04:30.3-05
```

Notice that on my computer the timestamp in the third line shows 13:04:30 .3, which is five hours behind the time input to the function: 18:04:30 .3. That output is appropriate because Lisbon’s time zone is at UTC 0, and my PostgreSQL is set to the Eastern time zone, which is UTC –5 in winter months. We’ll explore working with time zones in more detail, and you’ll learn to adjust its display, in the “Working with Time Zones” section.

Retrieving the Current Date and Time

If you need to record the current date or time as part of a query—when updating a row, for example—standard SQL provides functions for that too. The following functions record the time as of the start of the query:

`current_timestamp` Returns the current timestamp with time zone. A shorthand PostgreSQL-specific version is `now()`.

`localtimestamp` Returns the current timestamp without time zone. Avoid using `localtimestamp`, as a timestamp without a time zone can’t be placed in a global location and is thus meaningless.

`current_date` Returns the date.

`current_time` Returns the current time with time zone. Remember, though, without a date, the time alone with a time zone is useless.

`localtime` Returns the current time without time zone.

Because these functions record the time at the start of the query (or a collection of queries grouped under a *transaction*—see Chapter 10), they’ll provide that same time throughout the execution of a query regardless of how long the query runs. So, if your query updates 100,000 rows and takes 15 seconds to run, any timestamp recorded at the start of the query will be applied to each row, and so each row will receive the same timestamp.

If, instead, you want the date and time to reflect how the clock changes during the execution of the query, you can use the PostgreSQL-specific `clock_timestamp()` function to record the current time as it elapses. That way, if you’re updating 100,000 rows and inserting a timestamp each time, each row gets the time the row updated rather than the time at the start of the query. Note that `clock_timestamp()` can slow large queries and may be subject to system limitations.

[Listing 12-3](#) shows `current_timestamp` and `clock_timestamp()` in action when inserting a row in a table.

```
CREATE TABLE current_time_example (
    time_id integer GENERATED ALWAYS AS IDENTITY,
    1 current_timestamp_col timestamptz,
    2 clock_timestamp_col timestamptz
);

INSERT INTO current_time_example
    (current_timestamp_col, clock_timestamp_col)
3 (SELECT current_timestamp,
    clock_timestamp()
    FROM generate_series(1,1000));

SELECT * FROM current_time_example;
```

[Listing 12-3](#): Comparing `current_timestamp` and `clock_timestamp()` during row insert

The code creates a table that includes two `timestamptz` columns (the PostgreSQL shorthand for `timestamp with time zone`). The first holds the result of the `current_timestamp` function 1, which records the time at the start of the `INSERT` statement that adds 1,000 rows to the table. To do that, we use the `generate_series()` function, which returns a set of integers starting with 1 and ending with 1,000. The second column holds the result of the `clock_timestamp()` function 2, which records the time of insertion of each row. You call both functions as part of the `INSERT` statement 3. Run the query, and the result from the final `SELECT` statement should show that the time in the `current_timestamp_col` is the same for all rows, whereas the time in `clock_timestamp_col` increases with each row inserted.

Working with Time Zones

Recording a timestamp is most useful when you know where on the globe that time occurred—whether in Asia, Eastern Europe, or one of the 12 time zones of Antarctica.

Sometimes, however, datasets contain no time zone data in their datetime columns. This isn't always a deal-breaker in terms of analyzing the data. If you

know that every event happened in the same location—for example, readings from a temperature sensor in Bar Harbor, Maine—you can factor that into your analysis. Better, though, during import is to set your session time zone to represent the time zone of the data and load the datetimes into a `timestamptz` column. That strategy helps ward off dangerous misinterpretation of the data later.

Let's look at some strategies for managing how we work with time zones.

Finding Your Time Zone Setting

When working with timestamps that contain time zones, it's important to know your current time zone setting. If you installed PostgreSQL on your own computer, the server's default will be your local time zone. If you're connecting to a PostgreSQL database elsewhere, perhaps on a cloud provider such as Amazon Web Services, its time zone setting may be different than your own. To help avoid confusion, database administrators often set a shared server's time zone to UTC.

[*Listing 12-4*](#) shows two ways to view your current time zone setting: the `SHOW` command with `timezone` keyword and the `current_setting()` function with a `timezone` argument.

```
SHOW timezone;
SELECT current_setting('timezone');
```

[*Listing 12-4: Viewing your current time zone setting*](#)

Running either statement will display your time zone setting, which will vary according to your operating system and locale. Entering the statements in [*Listing 12-4*](#) into pgAdmin and running both my macOS and Linux computers returns `America/New_York`, one of several location names that falls into the Eastern time zone, which encompasses eastern Canada and the United States, the Caribbean, and parts of Mexico. On my Windows machine, the setting shows as `US/Eastern`.

NOTE

You can use `SHOW ALL;` to see the settings of every parameter on your PostgreSQL server.

Though both statements provide the same information, you may find `current_setting()` extra handy as an input to another function such as `make_timestamptz()`:

```
SELECT make_timestamptz(2022, 2, 22, 18, 4, 30.3,  
current_setting('timezone'));
```

[Listing 12-5](#) shows how to retrieve all time zone names, abbreviations, and their UTC offsets.

```
SELECT * FROM pg_timezone_abbrevs ORDER BY abbrev;  
SELECT * FROM pg_timezone_names ORDER BY name;
```

[Listing 12-5: Showing time zone abbreviations and names](#)

You can easily filter either of these `SELECT` statements with a `WHERE` clause to look up specific location names or time zones:

```
SELECT * FROM pg_timezone_names  
WHERE name LIKE 'Europe%'  
ORDER BY name;
```

This code should return a table listing that includes the time zone name, abbreviation, UTC offset, and a boolean column `is_dst` that notes whether the time zone is currently observing daylight saving time:

name	abbrev	utc_offset	is_dst
Europe/Amsterdam	CEST	02:00:00	true
Europe/Andorra	CEST	02:00:00	true
Europe/Astrakhan	+04	04:00:00	false
Europe/Athens	EEST	03:00:00	true
Europe/Belfast	BST	01:00:00	true
--snip--			

This is a faster way of looking up time zones than using Wikipedia. Now let's look at how to set the time zone to a particular value.

Setting the Time Zone

When you installed PostgreSQL, the server's default time zone was set as a parameter in *postgresql.conf*, a file that contains dozens of values read by PostgreSQL each time it starts. The location of *postgresql.conf* in your file system varies depending on your operating system and sometimes on the way you installed PostgreSQL. To make permanent changes to *postgresql.conf*, such as changing your time zone, you need to edit the file and restart the server, which might be impossible if you're not the owner of the machine. Changes to configurations might also have unintended consequences for other users or applications. Instead, we'll look at setting the time zone on a per-session basis, which should last as long as you're connected to the server, and then I'll cover working with *postgresql.conf* in more depth in Chapter 19. This solution is handy when you want to specify how you view a particular table or handle timestamps in a query.

To set the time zone for the current session while using pgAdmin, we use the command `SET TIME ZONE`, as shown in [Listing 12-6](#).

```
| 1 SET TIME ZONE 'US/Pacific';

2 CREATE TABLE time_zone_test (
    test_date timestamp;
);

3 INSERT INTO time_zone_test VALUES ('2023-01-01 4:00');

4 SELECT test_date
    FROM time_zone_test;

5 SET TIME ZONE 'US/Eastern';

6 SELECT test_date
    FROM time_zone_test;

7 SELECT test_date AT TIME ZONE 'Asia/Seoul'
    FROM time_zone_test;
```

[Listing 12-6](#): Setting the time zone for a client session

First, we set the time zone to `US/Pacific` **1**, which designates the Pacific time zone that covers western Canada and the United States along with Baja California in Mexico. The syntax `SET TIME ZONE` is part of the ANSI SQL standard. PostgreSQL also supports the nonstandard syntax `SET timezone TO`.

Second, we create a one-column table **2** with a data type of `timestamptz` and insert a single row to display a test result. Notice that the value inserted, `2023-01-01 4:00`, is a timestamp with no time zone **3**. You'll encounter timestamps with no time zone often, particularly when you acquire datasets restricted to a specific location.

When executed, the first `SELECT` statement **4** returns `2023-01-01 4:00` as a timestamp that now contains time zone data:

```
test_date
-----
2023-01-01 04:00:00-08
```

Here, the `-08` shows that the Pacific time zone is eight hours behind UTC in January, when standard time is in effect. Because we set the pgAdmin client's time zone to `US/Pacific` for this session, any value without a time zone entered into a column that is time zone-aware will be set to Pacific time. If we had entered a date that falls during daylight saving time, the UTC offset would be `-07`.

NOTE

On the server, the timestamp with time zone (or `timestamptz` shorthand) data type always stores data as UTC internally; the time zone setting governs how it's displayed.

Now comes some fun. We change the time zone for this session to the Eastern time zone using the `SET` command **5** and the `US/Eastern` designation. Then, when we execute the `SELECT` statement **6** again, the result should be as follows:

```
test_date
-----
2023-01-01 07:00:00-05
```

In this example, two components of the timestamp have changed: the time is now 07:00, and the UTC offset is -05 because we're viewing the timestamp from the perspective of the Eastern time zone: 4 AM Pacific is 7 AM Eastern. The database converts the original Pacific time value to whatever time zone we set at 5.

Even more convenient is that we can view a timestamp through the lens of any time zone without changing the session setting. The final `SELECT` statement uses the `AT TIME ZONE` keywords 7 to display the timestamp in our session as the Korea standard time (KST) zone by specifying `Asia/Seoul`:

```
timezone
-----
2023-01-01 21:00:00
```

Now we know that the value of 4 AM in `US/Pacific` on January 1, 2023, is equivalent to 9 PM that same day in `Asia/Seoul`. Again, this syntax changes the output data, but the data on the server remains unchanged. When using the `AT TIME ZONE` keywords, also note this quirk: if the original value is a timestamp with time zone, the output is a timestamp with no time zone. If the original value has no time zone, the output is timestamp with time zone.

The ability of databases to track time zones is extremely important for accurate calculations of intervals, as you'll see next.

Performing Calculations with Dates and Times

We can perform simple arithmetic on datetime and interval types the same way we can on numbers. Addition, subtraction, multiplication, and division are all possible in PostgreSQL using the math operators `+`, `-`, `*`, and `/`. For example, you can subtract one date from another date to get an integer that represents the difference in days between the two dates. The following code returns an integer of 3:

```
SELECT '1929-09-30'::date - '1929-09-27'::date;
```

The result indicates that these two dates are exactly three days apart.

Likewise, you can use the following code to add a time interval to a date to return a new date:

```
SELECT '1929-09-30'::date + '5 years'::interval;
```

This code adds five years to the date 1929-09-30 to return a timestamp value of 1934-09-30.

More examples of math functions you can use with dates and times are available in the PostgreSQL documentation at <https://www.postgresql.org/docs/current/functions-datetime.html>. Let's explore some more practical examples using actual transportation data.

Finding Patterns in New York City Taxi Data

When I visit New York City, I usually take at least one ride in one of the thousands of iconic yellow cars that ferry hundreds of thousands of people across the city's five boroughs each day. The New York City Taxi and Limousine Commission releases data on monthly yellow taxi trips plus other for-hire vehicles. We'll use this large, rich dataset to put date functions to practical use.

The *nyc_yellow_taxi_trips.csv* file available from the book's resources on GitHub (via the link at <https://nostarch.com/practical-sql-2nd-edition/>) holds one day of yellow taxi trip records from June 1, 2016. Save the file to your computer and execute the code in [Listing 12-7](#) to build the *nyc_yellow_taxi_trips* table. Remember to change the file path in the COPY command to the location where you've saved the file and adjust the path format to reflect whether you're using Windows, macOS, or Linux.

```
| CREATE TABLE nyc_yellow_taxi_trips (
|   trip_id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
|   vendor_id text NOT NULL,
|   tpep_pickup_datetime timestamp NOT NULL,
|   tpep_dropoff_datetime timestamp NOT NULL,
|   passenger_count integer NOT NULL,
|   trip_distance numeric(8,2) NOT NULL,
|   pickup_longitude numeric(18,15) NOT NULL,
|   pickup_latitude numeric(18,15) NOT NULL,
|   rate_code_id text NOT NULL,
|   store_and_fwd_flag text NOT NULL,
|   dropoff_longitude numeric(18,15) NOT NULL,
|   dropoff_latitude numeric(18,15) NOT NULL,
|   payment_type text NOT NULL,
|   fare_amount numeric(9,2) NOT NULL,
```

```

extra numeric(9,2) NOT NULL,
mta_tax numeric(5,2) NOT NULL,
tip_amount numeric(9,2) NOT NULL,
tolls_amount numeric(9,2) NOT NULL,
improvement_surcharge numeric(9,2) NOT NULL,
total_amount numeric(9,2) NOT NULL
);

2 COPY nyc_yellow_taxi_trips (
    vendor_id,
    tpep_pickup_datetime,
    tpep_dropoff_datetime,
    passenger_count,
    trip_distance,
    pickup_longitude,
    pickup_latitude,
    rate_code_id,
    store_and_fwd_flag,
    dropoff_longitude,
    dropoff_latitude,
    payment_type,
    fare_amount,
    extra,
    mta_tax,
    tip_amount,
    tolls_amount,
    improvement_surcharge,
    total_amount
)
FROM 'C:\YourDirectory\nyc_yellow_taxi_trips.csv'
WITH (FORMAT CSV, HEADER);

3 CREATE INDEX tpep_pickup_idx
ON nyc_yellow_taxi_trips (tpep_pickup_datetime);

```

[Listing 12-7](#): Creating a table and importing NYC yellow taxi data

The code in [Listing 12-7](#) builds the table **1**, imports the rows **2**, and creates an index **3**. In the COPY statement, we provide the names of columns because the input CSV file doesn't include the trip_id column that exists in the target table. That column is of type bigint and set as an auto-incrementing surrogate primary key. After your import is complete, you should have 368,774 rows, one for each yellow cab ride on June 1, 2016. You can count the rows in your table using the following code:

```
SELECT count(*) FROM nyc_yellow_taxi_trips;
```

Each row includes data on the number of passengers, the location of pickup and drop-off in latitude and longitude, and the fare and tips in US dollars. The data dictionary that describes all columns and codes is available at

https://www1.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf.

For these exercises, we're most interested in the timestamp columns `tpep_pickup_datetime` and `tpep_dropoff_datetime`, which represent the start and end times of the ride. (The Technology Passenger Enhancements Project [TPEP] is a program that in part includes automated collection of data about taxi rides.)

The values in both timestamp columns include the time zone: -4. That's the UTC offset for the Eastern time zone during summer, when daylight saving time is observed. If your PostgreSQL server isn't set to default to Eastern time, I suggest setting your time zone using the following code so your results will match mine:

```
SET TIME ZONE 'US/Eastern';
```

Now let's explore the patterns in these timestamps.

The Busiest Time of Day

One question you might ask of this data is when taxis provide the most rides. Is it morning or evening rush hour, or is there another time when ridership spikes? You can find the answer with a simple aggregation query that uses `date_part()`.

[Listing 12-8](#) contains the query to count rides by hour using the pickup time as the input.

```
SELECT
  1 date_part('hour', tpep_pickup_datetime) AS trip_hour,
  2 count(*)
FROM nyc_yellow_taxi_trips
GROUP BY trip_hour
ORDER BY trip_hour;
```

[Listing 12-8: Counting taxi trips by hour](#)

In the query's first column **1**, `date_part()` extracts the hour from `tpep_pickup_datetime` so we can group the number of rides by hour. Then we aggregate the number of rides in the second column via the `count()` function **2**. The rest of the query follows the standard patterns for grouping and ordering the results, which should return 24 rows, one for each hour of the day:

trip_hour	count
0	8182
1	5003
2	3070
3	2275
4	2229
5	3925
6	10825
7	18287
8	21062
9	18975
10	17367
11	17383
12	18031
13	17998
14	19125
15	18053
16	15069
17	18513
18	22689
19	23190
20	23098
21	24106
22	22554
23	17765

Eyeballing the numbers, it's apparent that on June 1, 2016, New York City taxis had the most passengers between 6 PM and 10 PM, possibly reflecting commutes home plus the plethora of city activities on a summer evening. But to see the overall pattern, it's best to visualize the data. Let's do this next.

Exporting to CSV for Visualization in Excel

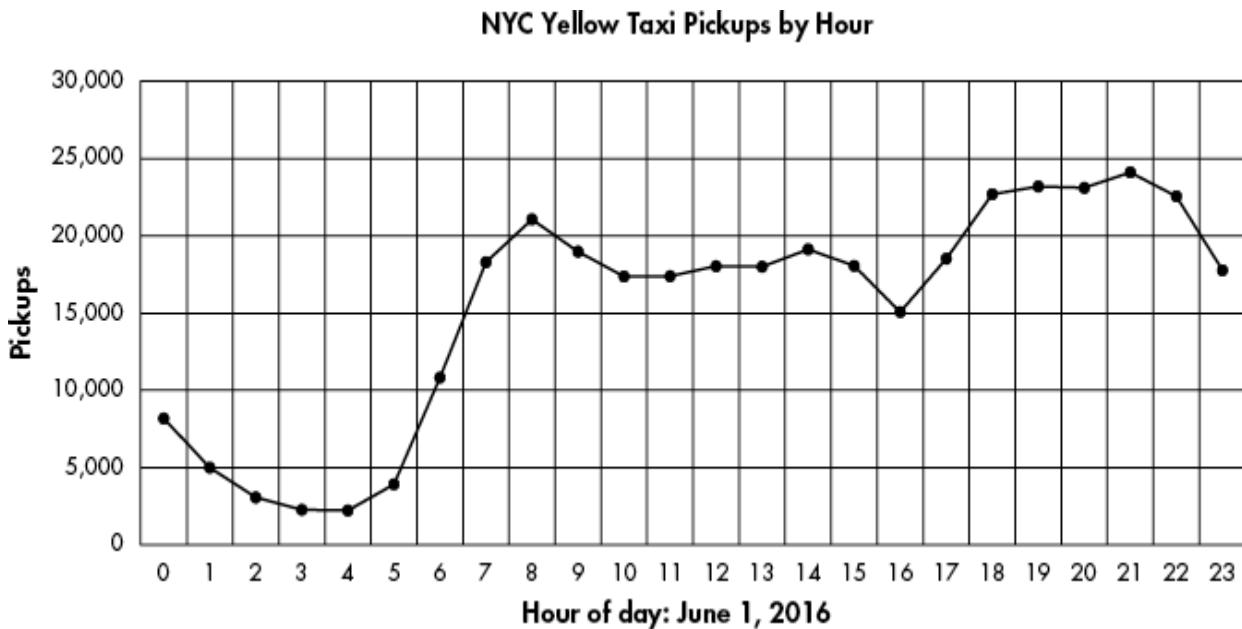
Charting data with a tool such as Microsoft Excel makes it easier to understand patterns, so I often export query results to a CSV file and work up a quick

chart. [Listing 12-9](#) uses the query from the preceding example within a COPY ... TO statement, similar to [Listing 5-9](#) in Chapter 5.

```
COPY
  (SELECT
    date_part('hour', tpep_pickup_datetime) AS trip_hour,
    count(*)
  FROM nyc_yellow_taxi_trips
  GROUP BY trip_hour
  ORDER BY trip_hour
  )
  TO 'C:\YourDirectory\hourly_taxi_pickups.csv'
  WITH (FORMAT CSV, HEADER);
```

[Listing 12-9](#): Exporting taxi pickups per hour to a CSV file

When I load the data into Excel and build a line graph, the day's pattern becomes more obvious and thought-provoking, as shown in [Figure 12-1](#).



[Figure 12-1](#): NYC yellow taxi pickups by hour

Rides bottomed out in the wee hours of the morning before rising sharply between 5 AM and 8 AM. Volume remained relatively steady throughout the day and increased again for evening rush hour after 5 PM. But there was a dip between 3 PM and 4 PM—why?

To answer that question, we would need to dig deeper to analyze data that spanned several days or even several months to see whether our data from June 1, 2016, is typical. We could use the `date_part()` function to compare trip volume on weekdays versus weekends by extracting the day of the week. To be even more ambitious, we could check weather reports and compare trips on rainy days versus sunny days. You can slice a dataset many ways to reach conclusions.

When Do Trips Take the Longest?

Let's investigate another interesting question: at which hour did taxi trips take the longest? One way to find an answer is to calculate the median trip time for each hour. The median is the middle value in an ordered set of values; it's often more accurate than an average for making comparisons because a few very small or very large values in the set won't skew the results as they would with the average.

In Chapter 6, we used the `percentile_cont()` function to find medians. We use it again in [Listing 12-10](#) to calculate median trip times.

```
SELECT
  1 date_part('hour', tpep_pickup_datetime) AS trip_hour,
  2 percentile_cont(.5)
    3 WITHIN GROUP (ORDER BY
                      tpep_dropoff_datetime - tpep_pickup_datetime) AS
      median_trip
  FROM nyc_yellow_taxi_trips
  GROUP BY trip_hour
  ORDER BY trip_hour;
```

[Listing 12-10](#): Calculating median trip time by hour

We're aggregating data by the hour portion of the timestamp column `tpep_pickup_datetime` again, which we extract using `date_part()` 1. For the input to the `percentile_cont()` function 2, we subtract the pickup time from the drop-off time in the `WITHIN GROUP` clause 3. The results show that the 1 PM hour has the highest median trip time of 15 minutes:

date_part	median_trip
-----	-----
0	00:10:04

```
1 00:09:27
2 00:08:59
3 00:09:57
4 00:10:06
5 00:07:37
6 00:07:54
7 00:10:23
8 00:12:28
9 00:13:11
10 00:13:46
11 00:14:20
12 00:14:49
13 00:15:00
14 00:14:35
15 00:14:43
16 00:14:42
17 00:14:15
18 00:13:19
19 00:12:25
20 00:11:46
21 00:11:54
22 00:11:37
23 00:11:14
```

As we would expect, trip times are shortest in the early morning. This makes sense because less traffic early in the day means passengers are more likely to get to their destinations faster.

Now that we've explored ways to extract portions of the timestamp for analysis, let's dig deeper into analysis that involves intervals.

Finding Patterns in Amtrak Data

Amtrak, the nationwide rail service in America, offers several packaged trips across the United States. The All American, for example, is a train that departs from Chicago and stops in New York, New Orleans, Los Angeles, San Francisco, and Denver before returning to Chicago. Using data from the Amtrak website (<https://www.amtrak.com/>), we'll build a table with information for each segment of the trip. The trip spans four time zones, so we'll track the time zone with each arrival and departure. Then we'll calculate the duration of the journey at each segment and figure out the length of the entire trip.

Calculating the Duration of Train Trips

Using [Listing 12-11](#), let's create a table that tracks the six segments of the All American route.

```
CREATE TABLE train_rides (
    trip_id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    segment text NOT NULL,
    departure timestamp NOT NULL, 1
    arrival timestamp NOT NULL
);

INSERT INTO train_rides (segment, departure, arrival) 2
VALUES
    ('Chicago to New York', '2020-11-13 21:30 CST', '2020-11-
14 18:23 EST'),
    ('New York to New Orleans', '2020-11-15 14:15 EST', '2020-
11-16 19:32 CST'),
    ('New Orleans to Los Angeles', '2020-11-17 13:45 CST',
'2020-11-18 9:00 PST'),
    ('Los Angeles to San Francisco', '2020-11-19 10:10 PST',
'2020-11-19 21:24 PST'),
    ('San Francisco to Denver', '2020-11-20 9:10 PST', '2020-
11-21 18:38 MST'),
    ('Denver to Chicago', '2020-11-22 19:10 MST', '2020-11-23
14:50 CST');

SET TIME ZONE 'US/Central'; 3

SELECT * FROM train_rides;
```

[Listing 12-11](#): Creating a table to hold train trip data

First, we use the standard `CREATE TABLE` statement. Note that columns for departure and arrival times are set to `timestamp` **1**. Next, we insert rows that represent the six legs of the trip **2**. Each timestamp input reflects the time zone of the city of departure or arrival. Specifying the city's time zone is the key to getting an accurate calculation of trip duration and accounting for time zone changes. It also accounts for annual changes to and from daylight saving time if they were to occur during the time span you're examining.

Next, we set the session to the Central time zone, the value for Chicago, using the `US/Central` designator **3**. We'll use Central time as our reference when viewing the timestamps so that regardless of your and my machine's default time zones, we'll share the same view of the data.

The final SELECT statement should return the contents of the table like this:

trip_id	segment	arrival	departure
-	-	-	-
06	1	Chicago to New York 2020-11-14 17:23:00-06	2020-11-13 21:30:00-
06	2	New York to New Orleans 2020-11-16 19:32:00-06	2020-11-15 13:15:00-
06	3	New Orleans to Los Angeles 2020-11-18 11:00:00-06	2020-11-17 13:45:00-
06	4	Los Angeles to San Francisco 2020-11-19 23:24:00-06	2020-11-19 12:10:00-
06	5	San Francisco to Denver 2020-11-21 19:38:00-06	2020-11-20 11:10:00-
06	6	Denver to Chicago 2020-11-23 14:50:00-06	2020-11-22 20:10:00-

All timestamps should now carry a UTC offset of -06, reflecting the Central time zone in the United States during November, when standard time is in effect. All time values display in their Central time equivalents.

Now that we've created segments corresponding to each leg of the trip, we'll use [Listing 12-12](#) to calculate the duration of each segment.

```
SELECT segment,
       1 to_char(departure, 'YYYY-MM-DD HH12:MI a.m. TZ') AS departure,
       2 arrival - departure AS segment_duration
  FROM train_rides;
```

[Listing 12-12](#): Calculating the length of each trip segment

This query lists the trip segment, the departure time, and the duration of the segment journey. Before we look at the calculation, notice the additional code around the `departure` column **1**. These are PostgreSQL-specific formatting functions that specify how to format different components of the timestamp. In this case, the `to_char()` function turns the `departure` timestamp column into a string of characters formatted as `YYYY-MM-DD HH12:MI a.m. TZ`. The `YYYY-MM-DD` portion specifies the ISO format for the date, and the `HH12:MI a.m.` portion presents the time in hours and minutes. The `HH12` portion

specifies the use of a 12-hour clock rather than 24-hour military time. The `a.m.` portion specifies that we want to show morning or night times using lowercase characters separated by periods, and the `TZ` portion denotes the time zone.

For a complete list of formatting functions, check out the PostgreSQL documentation at <https://www.postgresql.org/docs/current/functions-formatting.html>.

Last, we subtract `departure` from `arrival` to determine the `segment_duration` **2**. When you run the query, the output should look like this:

segment	departure	
<code>segment_duration</code>		
-----	-----	-
-----	-----	
Chicago to New York 19:53:00	2020-11-13 09:30 p.m. CST	
New York to New Orleans day 06:17:00	2020-11-15 01:15 p.m. CST	1
New Orleans to Los Angeles 21:15:00	2020-11-17 01:45 p.m. CST	
Los Angeles to San Francisco 11:14:00	2020-11-19 12:10 p.m. CST	
San Francisco to Denver day 08:28:00	2020-11-20 11:10 a.m. CST	1
Denver to Chicago 18:40:00	2020-11-22 08:10 p.m. CST	

Subtracting one timestamp from another produces an `interval` data type, which was introduced in Chapter 4. As long as the value is less than 24 hours, PostgreSQL presents the interval in the `HH:MM:SS` format. For values greater than 24 hours, it returns the format `1 day 08:28:00`, as shown in the San Francisco to Denver segment.

In each calculation, PostgreSQL accounts for the changes in time zones so we don't inadvertently add or lose hours when subtracting. If we used a `timestamp` without `time zone` data type, we would end up with an incorrect trip length if a segment spanned multiple time zones.

Calculating Cumulative Trip Time

As it turns out, San Francisco to Denver is the longest leg of the All American train trip. But how long does the entire trip take? To answer this question, we'll

revisit window functions, which you first learned about in “Ranking with `rank()` and `dense_rank()`” in Chapter 11.

Our prior query produced an interval, which we labeled `segment_duration`. The next natural next step would be to write a query to add those values, creating a cumulative interval after each segment. And indeed, we can use `sum()` as a window function, combined with the `OVER` clause used in Chapter 11, to create running totals. But when we do, the resulting values are odd. To see what I mean, run the code in [*Listing 12-13*](#).

```
SELECT segment,
       arrival - departure AS segment_duration,
       sum(arrival - departure) OVER (ORDER BY trip_id) AS
cume_duration
FROM train_rides;
```

[*Listing 12-13*](#): Calculating cumulative intervals using `OVER`

In the third column, we sum the intervals generated when we subtract `departure` from `arrival`. The resulting running total in the `cume_duration` column is accurate but formatted in an unhelpful way:

segment cume_duration	segment_duration	
---	-----	-----
Chicago to New York	19:53:00	19:53:00
New York to New Orleans	1 day 06:17:00	1 day
26:10:00		
New Orleans to Los Angeles	21:15:00	1 day
47:25:00		
Los Angeles to San Francisco	11:14:00	1 day
58:39:00		
San Francisco to Denver	1 day 08:28:00	2 days
67:07:00		
Denver to Chicago	18:40:00	2 days
85:47:00		

PostgreSQL creates one sum for the day portion of the interval and another for the hours and minutes. So, instead of a more understandable cumulative time of 5 days 13:47:00, the database reports 2 days 85:47:00. Both results amount to the same length of time, but 2 days 85:47:00 is harder to

decipher. This is an unfortunate limitation of summing the database intervals using this syntax.

To get around the limitation, we'll wrap the window function calculation for the cumulative duration inside the `justify_interval()` function, shown in [Listing 12-14](#).

```
SELECT segment,
       arrival - departure AS segment_duration,
  1 justify_interval(sum(arrival - departure)
                     OVER (ORDER BY trip_id)) AS
cume_duration
FROM train_rides;
```

[Listing 12-14](#): Using `justify_interval()` to better format cumulative trip duration

The `justify_interval()` function 1 standardizes output of interval calculations so that groups of 24 hours are rolled up to days, and groups of 30 days are rolled up to months. So, instead of returning a cumulative duration of 2 days 85:47:00, as in the previous listing, `justify_interval()` converts 72 of those 85 hours to three days and adds them to the days value. The output is easier to understand:

segment	segment_duration	cume_duration
Chicago to New York	19:53:00	19:53:00
New York to New Orleans	1 day 06:17:00	2 days 02:10:00
New Orleans to Los Angeles	21:15:00	2 days 23:25:00
Los Angeles to San Francisco	11:14:00	3 days 10:39:00
San Francisco to Denver	1 day 08:28:00	4 days 19:07:00
Denver to Chicago	18:40:00	5 days 13:47:00

The final `cume_duration` adds all the segments to return the total trip duration of 5 days 13:47:00. That's a long time to spend on a train, but I'm sure the scenery is well worth the ride.

Wrapping Up

Handling times and dates in SQL databases adds an intriguing dimension to your analysis, letting you answer questions about when an event occurred along with other temporal concerns in your data. With a solid grasp of time and date

formats, time zones, and functions to dissect the components of a timestamp, you can analyze just about any dataset you come across.

Next, we'll look at advanced query techniques that help answer more complex questions.

TRY IT YOURSELF

Try these exercises to test your skills on dates and times:

Using the New York City taxi data, calculate the length of each ride using the pickup and drop-off timestamps. Sort the query results from the longest ride to the shortest. Do you notice anything about the longest or shortest trips that you might want to ask city officials about?

Using the `AT TIME ZONE` keywords, write a query that displays the date and time for London, Johannesburg, Moscow, and Melbourne the moment January 1, 2100, arrives in New York City. Use the code in [Listing 12-5](#) to find time zone names.

As a bonus challenge, use the statistics functions in Chapter 11 to calculate the correlation coefficient and *r*-squared values using trip time and the `total_amount` column in the New York City taxi data, which represents the total amount charged to passengers. Do the same with the `trip_distance` and `total_amount` columns. Limit the query to rides that last three hours or less.

13

ADVANCED QUERY TECHNIQUES



Sometimes data analysis requires advanced SQL techniques that go beyond a table join or basic SELECT query. In this chapter, we'll cover techniques that include writing a query that uses the results of other queries as inputs and reclassifying numerical values into categories before counting them.

For the exercises, I'll introduce a dataset of temperatures recorded in select US cities, and we'll revisit datasets you've created in previous chapters. The code for the exercises is available, along with all the book's resources, at <https://nostarch.com/practical-sql-2nd-edition/>. You'll continue to use the `analysis` database you've already built. Let's get started.

Using Subqueries

A *subquery* is a query nested inside another query. Typically, it performs a calculation or a logical test or generates rows to be passed into the main outer query. Subqueries are part of standard ANSI SQL, and the syntax is not unusual: we just enclose a query in parentheses. For example, we can write a subquery that returns multiple rows and treat those results as a table in the FROM clause of the main outer query. Or we can create a *scalar subquery* that returns a single value and use it as part of an *expression* to filter rows via WHERE, IN, and HAVING clauses. A *correlated subquery* is one that depends on a value or

table name from the outer query to execute. Conversely, an *uncorrelated subquery* has no reference to objects in the main query.

It's easier to understand these concepts by working with data, so let's revisit several datasets from earlier chapters, including the census county-level population estimates table `us_counties_pop_est_2019` and the business patterns table `cbp_naics_72_establishments`.

Filtering with Subqueries in a WHERE Clause

A WHERE clause lets you filter query results based on criteria you provide, using an expression such as `WHERE quantity > 1000`. But this requires that you already know the value to use for comparison. What if you don't? That's one way a subquery comes in handy: it lets you write a query that generates one or more values to use as part of an expression in a WHERE clause.

Generating Values for a Query Expression

Say you wanted to write a query to show which US counties are at or above the 90th percentile, or top 10 percent, for population. Rather than writing two separate queries—one to calculate the 90th percentile and another to find counties with populations at or higher—you can do both at once using a subquery as part of a WHERE clause, as shown in [Listing 13-1](#).

```
SELECT county_name,
       state_name,
       pop_est_2019
  FROM us_counties_pop_est_2019
 WHERE pop_est_2019 >= (
   SELECT percentile_cont(.9) WITHIN GROUP (ORDER BY
   pop_est_2019)
   FROM us_counties_pop_est_2019
 )
 ORDER BY pop_est_2019 DESC;
```

[Listing 13-1:](#) Using a subquery in a WHERE clause

The WHERE clause 1, which filters by the total population column `pop_est_2019`, doesn't include a value as it normally would. Instead, after the `>=` comparison operators, we provide a subquery in parentheses. This subquery

uses the `percentile_cont()` function to generate one value: the 90th percentile cutoff point in the `pop_est_2019` column.

NOTE

Using `percentile_cont()` to filter with a subquery works only if you pass in a single input, as shown. If you pass in an array, as in [Listing 6-12](#) on page 90, `percentile_cont()` returns an array, and the query will fail to evaluate the `>=` against an array type.

This is an example of an uncorrelated subquery. It does not depend on any values in the outer query, and it will be executed just once to generate the requested value. If you run the subquery portion only, by highlighting it in pgAdmin, it will execute, and you should see a result of `213707.3`. But you won't see that number when you run the entire query in [Listing 13-1](#), because the subquery result is passed directly to the outer query's `WHERE` clause.

The entire query should return 315 rows, or about 10 percent of the 3,142 rows in `us_counties_pop_est_2019`.

county_name	state_name	pop_est_2019
Los Angeles County	California	10039107
Cook County	Illinois	5150233
Harris County	Texas	4713325
Maricopa County	Arizona	4485414
San Diego County	California	3338330
--snip--		
Cabarrus County	North Carolina	216453
Yuma County	Arizona	213787

The result includes all counties with a population greater than or equal to `213707.3`, the value the subquery generated.

Using a Subquery to Identify Rows to Delete

We can use the same subquery in a `DELETE` statement to specify what to remove from a table. In [Listing 13-2](#), we make a copy of the census table using the method you learned in Chapter 10 and then delete everything from that backup except the 315 counties in the top 10 percent of population.

```
CREATE TABLE us_counties_2019_top10 AS
SELECT * FROM us_counties_pop_est_2019;

DELETE FROM us_counties_2019_top10
WHERE pop_est_2019 < (
    SELECT percentile_cont(.9) WITHIN GROUP (ORDER BY
pop_est_2019)
    FROM us_counties_2019_top10
) ;
```

[Listing 13-2](#): Using a subquery in a WHERE clause with DELETE

Run the code in [Listing 13-2](#), and then execute `SELECT count(*) FROM us_counties_2019_top10;` to count the remaining rows. The result should be 315 rows, which is the original 3,142 minus the 2,827 below the value identified by the subquery.

Creating Derived Tables with Subqueries

If your subquery returns rows and columns, you can place it in a `FROM` clause to create a new table known as a *derived table* that you can query or join with other tables, just as you would a regular table. It's another example of an uncorrelated subquery.

Let's look at a simple example. In Chapter 6, you learned the difference between average and median. A median often better indicates a dataset's central value because a few very large or small values (or outliers) can skew an average. For that reason, I often compare the two. If they're close, the data more likely falls in a *normal distribution* (the familiar bell curve), and the average is a good representation of the central value. If the average and median are far apart, some outliers might be having an effect or the distribution is skewed, not normal.

Finding the average and median population of US counties as well as the difference between them is a two-step process. We need to calculate the average and the median and then subtract the two. We can do both operations in one fell swoop with a subquery in the `FROM` clause, as shown in [Listing 13-3](#).

```
SELECT round(calcs.average, 0) AS average,
       calcs.median,
       round(calcs.average - calcs.median, 0) AS
median_average_diff
```

```

FROM (
  1 SELECT avg(pop_est_2019) AS average,
         percentile_cont(.5)
              WITHIN GROUP (ORDER BY pop_est_2019)::numeric
        AS median
      FROM us_counties_pop_est_2019
    )
  2 AS calcs;

```

[Listing 13-3](#): Subquery as a derived table in a `FROM` clause

The subquery 1 that produces a derived table is straightforward. We use the `avg()` and `percentile_cont()` functions to find the average and median of the census table's `pop_est_2019` column and name each column with an alias. Then we name the derived table `calcs` 2 so we can reference it in the main query.

In the main query, we subtract the `median` from the `average`, both of which are returned by the subquery. The result is rounded and labeled with the alias `median_average_diff`. Run the query, and the result should be the following:

average	median	median_average_diff
104468	25726	78742

The difference between the median and average, 78,742, is nearly three times the size of the median. That indicates we have some high-population counties inflating the average.

Joining Derived Tables

Joining multiple derived tables lets you perform several preprocessing steps before final calculations in a main query. For example, in Chapter 11, we calculated the rate of tourism-related businesses per 1,000 population in each county. Let's say we want to do the same at the state level. Before we can calculate that rate, we need to know the number of tourism businesses in each state and the population of each state. [Listing 13-4](#) shows how to write subqueries for both tasks and join them to calculate the overall rate.

```

SELECT census.state_name AS st,
       census.pop_est_2018,

```

```

        est.establishment_count,
    1
    round((est.establishment_count/census.pop_est_2018::numeric) *
1000, 1)
                    AS estabs_per_thousand
FROM
(
    2 SELECT st,
            sum(establishments) AS establishment_count
        FROM cbp_naics_72_establishments
        GROUP BY st
)
AS est
JOIN
(
    3 SELECT state_name,
            sum(pop_est_2018) AS pop_est_2018
        FROM us_counties_pop_est_2019
        GROUP BY state_name
)
AS census
  ON est.st = census.state_name
ORDER BY estabs_per_thousand DESC;

```

[Listing 13.4](#): Joining two derived tables

You learned how to calculate rates in Chapter 11, so the math and syntax in the outer query for finding `estabs_per_thousand` 1 should be familiar. We divide the number of establishments by the population and then multiply that quotient by a thousand. For the inputs, we use the values generated from two derived tables.

The first 2 finds the number of establishments in each state using the `sum()` aggregate function. We give this derived table the alias `est` for reference in the main part of the query. The second 3 finds the 2018 estimated population by state by using `sum()` on the `pop_est_2018` column. We alias this derived table as `census`.

Next, we join the derived tables 4 by linking the `st` column in `est` to the `state_name` column in `census`. We then list the results in descending order based on the rate. Here's a sample of the 51 rows showing the highest and lowest rates:

st estabs_per_thousand	pop_est_2018	establishment_count
District of Columbia 3.9	701547	2754
Montana 3.4	1060665	3569
Vermont 3.2	624358	1991
Maine 3.2	1339057	4282
Wyoming 3.1	577601	1808
--snip--		
Arizona 1.9	7158024	13288
Alabama 1.9	4887681	9140
Utah 1.9	3153550	6062
Mississippi 1.9	2981020	5645
Kentucky 1.8	4461153	8251

At the top is Washington, DC, unsurprising given the tourist activity generated by the museums, monuments, and other attractions in the nation's capital. Montana may seem like a surprise in second place, but it's a low-population state with major tourist destinations including Glacier and Yellowstone national parks. Mississippi and Kentucky are among those states with the fewest tourism-related businesses per 1,000 population.

Generating Columns with Subqueries

You can also place a subquery in the column list after `SELECT` to generate a value for that column in the query result. The subquery must generate only a single row. For example, the query in [Listing 13-5](#) selects the geography and population information from `us_counties_pop_est_2019` and then adds an uncorrelated subquery to add the median of all counties to each row in the new column `us_median`.

```
SELECT county_name,
       state_name AS st,
       pop_est_2019,
       (SELECT percentile_cont(.5) WITHIN GROUP (ORDER BY
pop_est_2019)
        FROM us_counties_pop_est_2019) AS us_median
FROM us_counties_pop_est_2019;
```

[Listing 13-5](#): Adding a subquery to a column list

The first rows of the result set should look like this:

	county_name	st
	pop_est_2019	us_median
-----	-----	-----
Autauga County		Alabama
55869	25726	
Baldwin County		Alabama
223234	25726	
Barbour County		Alabama
24686	25726	
Bibb County		Alabama
22394	25726	
Blount County		Alabama
57826	25726	
--snip--		

On its own, that repeating `us_median` value isn't very helpful. It would be more interesting and useful to generate values that indicate how much each county's population deviates from the median value. Let's look at how we can use the same subquery technique to do that. [*Listing 13-6*](#) builds on [*Listing 13-5*](#) by substituting a subquery after `SELECT` that calculates the difference between the population and the median for each county.

```
SELECT county_name,
       state_name AS st,
       pop_est_2019,
       pop_est_2019 - (SELECT percentile_cont(.5) WITHIN GROUP
(ORDER BY pop_est_2019) 1
                    FROM us_counties_pop_est_2019) AS
diff_from_median
FROM us_counties_pop_est_2019
WHERE (pop_est_2019 - (SELECT percentile_cont(.5) WITHIN GROUP
```

```
(ORDER BY pop_est_2019) 2
    FROM us_counties_pop_est_2019))
BETWEEN -1000 AND 1000;
```

[Listing 13-6](#): Using a subquery in a calculation

The subquery 1 is now part of a calculation that subtracts the subquery's result from `pop_est_2019`, the total population, giving the column an alias of `diff_from_median`. To make this query even more useful, we can filter results to show counties whose population is close to the median. To do this, we repeat the calculation with the subquery in the `WHERE` clause 2 and filter results using the `BETWEEN -1000 AND 1000` expression.

The outcome should reveal 78 counties. Here are the first five rows:

county_name diff_from_median	st	pop_est_2019
Cherokee County 470	Alabama	26196
Geneva County 545	Alabama	26271
Cleburne County -807	Arkansas	24919
Johnson County 852	Arkansas	26578
St. Francis County -732	Arkansas	24994
--snip--		

Bear in mind that subqueries can add to overall query execution time. In [*Listing 13-6*](#), I removed the subquery from [*Listing 13-5*](#) that displays the column `us_median` to avoid repeating the subquery a third time. With our data set, the impact is minimal; if we were working with millions of rows, winnowing some unneeded subqueries might provide a significant speed boost.

Understanding Subquery Expressions

You can also use subqueries to filter rows by evaluating whether a condition evaluates as `true` or `false`. For this, we can use *subquery expressions*, which are a

combination of a keyword with a subquery and are generally used in WHERE clauses to filter rows based on the existence of values in another table.

The PostgreSQL documentation at <https://www.postgresql.org/docs/current/functions-subquery.html> lists available subquery expressions, but here we'll examine the syntax for two that tend to be used most often: IN and EXISTS. To prep, run the code in [Listing 13-7](#) to create a small table called `retirees` that we'll query along with the `employees` table you built in Chapter 7. We'll imagine that we've received this data from a vendor listing people who've applied for retirement benefits.

```
CREATE TABLE retirees (
    id int,
    first_name text,
    last_name text
);

INSERT INTO retirees
VALUES (2, 'Janet', 'King'),
       (4, 'Michael', 'Taylor');
```

[Listing 13-7](#): Creating and filling a `retirees` table

Now let's use this table in some subquery expressions.

Generating Values for the IN Operator

The subquery expression `IN (subquery)` works like the IN operator example in Chapter 3 except we employ a subquery to provide the list of values to check against rather than manually entering one. In [Listing 13-8](#), we use an uncorrelated subquery, which will be executed one time, to generate `id` values from the `retirees` table. The values it returns become the list for the IN operator in the WHERE clause. This lets us find employees who are also present in the table of `retirees`.

```
SELECT first_name, last_name
FROM employees
WHERE emp_id IN (
    SELECT id
    FROM retirees)
ORDER BY emp_id;
```

[Listing 13-8](#): Generating values for the `IN` operator

Run the query, and the output shows the two people in `employees` whose `emp_id` have a matching `id` in the `retirees` table:

first_name	last_name
Janet	King
Michael	Taylor

NOTE

Avoid using `NOT IN`. The presence of `NULL` values in a subquery result set will cause a query with a `NOT IN` expression to return no rows. The PostgreSQL wiki recommends using `NOT EXISTS` instead, described in the next section.

Checking Whether Values Exist

The subquery expression `EXISTS (subquery)` returns a value of `true` if the subquery in parentheses returns at least one row. If it returns no rows, `EXISTS` evaluates to `false`.

The `EXISTS` subquery expression in [*Listing 13-9*](#) shows an example of a correlated subquery—it includes an expression in its `WHERE` clause that requires data from the outer query. Also, because the subquery is correlated, it will execute once for each row returned by the outer query, each time checking whether there's an `id` in `retirees` that matches `emp_id` in `employees`. If there is a match, the `EXISTS` expression returns `true`.

```
SELECT first_name, last_name
FROM employees
WHERE EXISTS (
    SELECT id
    FROM retirees
    WHERE id = employees.emp_id);
```

[Listing 13-9](#): Using a correlated subquery with `WHERE EXISTS`

When you run the code, it should return the same result as it did in [*Listing 13-8*](#). Using this approach is particularly helpful if you need to join on more than one

column, which you can't do with the `IN` expression. You also can add the `NOT` keyword with `EXISTS` to perform the opposite function and find rows in the `employees` table with no corresponding record in `retirees`, as in [Listing 13-10](#).

```
SELECT first_name, last_name
FROM employees
WHERE NOT EXISTS (
    SELECT id
    FROM retirees
    WHERE id = employees.emp_id);
```

[Listing 13-10](#): Using a correlated subquery with `WHERE NOT EXISTS`

That should produce these results:

first_name	last_name
Julia	Reyes
Arthur	Pappas

The technique of using `NOT` with `EXISTS` is helpful for finding missing values or assessing whether a dataset is complete.

Using Subqueries with `LATERAL`

Placing the keyword `LATERAL` before subqueries in a `FROM` clause adds several bits of functionality that help simplify otherwise complicated queries.

LATERAL with `FROM`

First, a subquery preceded by `LATERAL` can reference tables and other subqueries that appear before it in the `FROM` clause, which can reduce redundant code by making it easy to reuse calculations.

[Listing 13-11](#) calculates the change in county population from 2018 to 2019 two ways: raw change in numbers and percent change.

```
SELECT county_name,
       state_name,
       pop_est_2018,
       pop_est_2019,
       raw_chg,
       round(pct_chg * 100, 2) AS pct_chg
```

```

FROM us_counties_pop_est_2019,
     1 LATERAL (SELECT pop_est_2019 - pop_est_2018 AS
raw_chg) rc,
     2 LATERAL (SELECT raw_chg / pop_est_2018::numeric AS
pct_chg) pc
ORDER BY pct_chg DESC;

```

[Listing 13-11: Using LATERAL subqueries in the FROM clause](#)

In the `FROM` clause, after naming the `us_counties_pop_est_2019` table, we add the first `LATERAL` subquery **1**. In parentheses, we place a query that subtracts the 2018 population estimate from the 2019 estimate and alias the result as `raw_chg`. Because a `LATERAL` subquery can reference a table listed before it in the `FROM` clause without needing to specify its name, we can omit the `us_counties_pop_est_2019` table from the subquery. Subqueries in `FROM` must have an alias, so we label this one `rc`.

The second `LATERAL` subquery **2** calculates the percent change in population from 2018 to 2019. To find percent change, we must know the raw change. Rather than re-calculate it, we can reference the `raw_chg` value from the previous subquery. That helps make our code shorter and easier to read.

The query results should look like this:

county_name raw_chg	state_name	pop_est_2018	pop_est_2019
pct_chg			
Loving County 21	Texas	148	169
14.19			
McKenzie County 1430	North Dakota	13594	15024
10.52			
Loup County 47	Nebraska	617	664
7.62			
Kaufman County 7875	Texas	128279	136154
6.14			
Williams County 2120	North Dakota	35469	37589
5.98			
--snip--			

LATERAL with JOIN

Combining LATERAL with JOIN creates functionality similar to a *for loop* in a programming language: for each row generated by the query in front of the LATERAL join, a subquery or function after the LATERAL join will be evaluated once.

We'll reuse the teachers table from Chapter 2 and create a new table to record each time a teacher swipes a badge to unlock a lab door. Our task is to find the two most recent times a teacher accessed a lab. [Listing 13-12](#) shows the code.

```
| ALTER TABLE teachers ADD CONSTRAINT id_key PRIMARY KEY (id);  
?  
? CREATE TABLE teachers_lab_access (  
    access_id bigint PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
    access_time timestamp with time zone,  
    lab_name text,  
    teacher_id bigint REFERENCES teachers (id)  
);  
  
?  
? INSERT INTO teachers_lab_access (access_time, lab_name,  
    teacher_id)  
VALUES ('2022-11-30 08:59:00-05', 'Science A', 2),  
      ('2022-12-01 08:58:00-05', 'Chemistry B', 2),  
      ('2022-12-21 09:01:00-05', 'Chemistry A', 2),  
      ('2022-12-02 11:01:00-05', 'Science B', 6),  
      ('2022-12-07 10:02:00-05', 'Science A', 6),  
      ('2022-12-17 16:00:00-05', 'Science B', 6);  
  
SELECT t.first_name, t.last_name, a.access_time, a.lab_name  
FROM teachers t  
LEFT JOIN LATERAL (SELECT *  
    FROM teachers_lab_access  
    WHERE teacher_id = t.id  
    ORDER BY access_time DESC  
    LIMIT 2) a  
    ON true  
    ORDER BY t.id;
```

[Listing 13-12](#): Using a subquery with a LATERAL join

First, we add a primary key 1 to the teachers table using ALTER TABLE (we didn't place a constraint on this table in Chapter 2 because we were just covering the basics about creating tables). Next, we make a simple

`teachers_lab_access` table **2** with columns to record the lab name and access timestamp. The table has a surrogate primary key `access_id` and a foreign key `teacher_id` that references `id` in `teachers`. Finally, we add six rows to the table using an `INSERT` **3** statement.

Now we're ready to query the data. In our `SELECT` statement, we join `teachers` to a subquery using `LEFT JOIN`. We add the `LATERAL` **4** keyword, which means for each row returned from `teachers`, the subquery will execute, returning the two most recent labs accessed by that particular teacher and the times they were accessed. Using `LEFT JOIN` will return all rows from `teachers` regardless of whether the subquery finds a matching teacher in `teachers_lab_access`.

In the `WHERE` **5** clause, the subquery references the outer query using the foreign key of `teacher_lab_access`. This `LATERAL` join syntax requires that the subquery have an alias **6**, which here is `a`, and the value `true` in the `ON` portion **7** of the `JOIN` clause. In this case, `true` lets us create the join without naming specific columns to join upon.

Run the query, and the results should look like this:

first_name	last_name	access_time	lab_name
Janet	Smith		
Lee	Reynolds	2022-12-21 09:01:00-05	Chemistry A
Lee	Reynolds	2022-12-01 08:58:00-05	Chemistry B
Samuel	Cole		
Samantha	Bush		
Betty	Diaz		
Kathleen	Roush	2022-12-17 16:00:00-05	Science B
Kathleen	Roush	2022-12-07 10:02:00-05	Science A

The two teachers with IDs in the access table have their two most recent lab access times shown. Teachers who didn't access a lab display `NULL` values; if we want to remove those from the results, we could substitute `INNER JOIN` (or just `JOIN`) for `LEFT JOIN`.

Next, let's explore another syntax for working with subqueries.

Using Common Table Expressions

The *common table expression (CTE)*, a relatively recent addition to standard SQL, allows you to use one or more `SELECT` queries to predefine temporary tables

that you can reference as often as needed in your main query. CTEs are informally called `WITH` queries because you define them using a `WITH ... AS` statement. The following examples show some advantages of using them, including cleaner code and less redundancy.

[Listing 13-13](#) shows a simple CTE based on our census estimates data. The code determines how many counties in each state have 100,000 people or more. Let's walk through the example.

```
| WITH large_counties (county_name, state_name, pop_est_2019)
|   AS (
|     2 SELECT county_name, state_name, pop_est_2019
|           FROM us_counties_pop_est_2019
|           WHERE pop_est_2019 >= 100000
|     )
|   3 SELECT state_name, count(*)
|       FROM large_counties
|       GROUP BY state_name
|       ORDER BY count(*) DESC;
```

[Listing 13-13](#): Using a simple CTE to count large counties

The `WITH ... AS` statement **1** defines the temporary table `large_counties`. After `WITH`, we name the table and list its column names in parentheses. Unlike column definitions in a `CREATE TABLE` statement, we don't need to provide data types, because the temporary table inherits those from the subquery **2**, which is enclosed in parentheses after `AS`. The subquery must return the same number of columns as defined in the temporary table, but the column names don't need to match. The column list is optional if you're not renaming columns; I've included it here so you can see the syntax.

The main query **3** counts and groups the rows in `large_counties` by `state_name` and then orders by the count in descending order. The top six rows of the results should look like this:

state_name	count
Texas	40
Florida	36
California	35
Pennsylvania	31
New York	28

Texas, Florida, and California are among the states that had the most counties with a 2019 population of 100,000 or more.

[Listing 13-14](#) uses a CTE to rewrite the join of derived tables in [Listing 13-4](#) (finding the rate of tourism-related businesses per 1,000 population in each state) into a more readable format.

WITH

```
1 counties (st, pop_est_2018) AS
  (SELECT state_name, sum(pop_est_2018)
   FROM us_counties_pop_est_2019
   GROUP BY state_name),

2 establishments (st, establishment_count) AS
  (SELECT st, sum(establishments) AS establishment_count
   FROM cbp_naics_72_establishments
   GROUP BY st)

SELECT counties.st,
       pop_est_2018,
       establishment_count,
       round((establishments.establishment_count /
              counties.pop_est_2018)::numeric(10,1)) * 1000, 1)
          AS estabs_per_thousand
3 FROM counties JOIN establishments
  ON counties.st = establishments.st
 ORDER BY estabs_per_thousand DESC;
```

[Listing 13-14:](#) Using CTEs in a table join

Following the `WITH` keyword, we define two tables using subqueries. The first subquery, `counties 1`, returns the 2018 population of each state. The second, `establishments 2`, returns the number of tourism-related businesses per state. With those tables defined, we join them `3` on the `st` column in each table and calculate the rate per thousand. The results are identical to the joined derived tables in [Listing 13-4](#), but [Listing 13-14](#) is easier to comprehend.

As another example, you can use a CTE to simplify queries that have redundant code. For example, in [Listing 13-6](#), we used a subquery with the

`percentile_cont()` function in two locations to find median county population. In [Listing 13-15](#), we can write that subquery just once as a CTE.

```
| WITH us_median AS
|   (SELECT percentile_cont(.5)
|    WITHIN GROUP (ORDER BY pop_est_2019) AS us_median_pop
|    FROM us_counties_pop_est_2019)
|
|   SELECT county_name,
|         state_name AS st,
|         pop_est_2019,
|         2 us_median_pop,
|         3 pop_est_2019 - us_median_pop AS diff_from_median
|   4 FROM us_counties_pop_est_2019 CROSS JOIN us_median
|   5 WHERE (pop_est_2019 - us_median_pop)
|         BETWEEN -1000 AND 1000;
```

[Listing 13-15](#): Using CTEs to minimize redundant code

After the `WITH` keyword, we define `us_median` **1** as the result of the same subquery used in [Listing 13-6](#), which finds the median population using `percentile_cont()`. Then we reference the `us_median_pop` column on its own **2**, as part of a calculated column **3**, and in a `WHERE` clause **5**. To make the value available to every row in the `us_counties_pop_est_2019` table during `SELECT`, we use the `CROSS JOIN` **4** you learned in Chapter 7.

This query provides identical results to those in [Listing 13-6](#), but we had to write the subquery that finds the median only once. Another bonus is that you can more easily revise the query. For example, to find counties whose population is close to the 90th percentile, you need to substitute `.9` for `.5` as input to `percentile_cont()` in only one place.

Readable code, less redundancy, and easier modifications are often-cited reasons for using CTEs. Another, beyond the scope of this book, is the ability to add a `RECURSIVE` keyword that lets a CTE loop through query results within the CTE itself—a task useful when dealing with data organized in a hierarchy. An example is a company’s personnel listing, where you might want to find all the people who report to a particular executive. The recursive CTE will start with the executive and then loop down through rows finding her direct reports and then the people who report to those people. You can learn more about

recursive query syntax via the PostgreSQL documentation at <https://www.postgresql.org/docs/current/queries-with.html>.

Performing Cross Tabulations

Cross tabulations provide a simple way to summarize and compare variables by displaying them in a table layout, or matrix. Rows in the matrix represent one variable, columns represent another variable, and each cell where a row and column intersect holds a value, such as a count or percentage.

You'll often see cross tabulations, also called *pivot tables* or *crosstabs*, used to report summaries of survey results or to compare pairs of variables. A frequent example happens during elections when candidates' votes are tallied by geography:

candidate	ward 1	ward 2	ward 3
Collins	602	1,799	2,112
Banks	599	1,398	1,616
Rutherford	911	902	1,114

In this case, the candidates' names are one variable, the wards (or city districts) are another variable, and the cells at the intersection of the two hold the vote totals for that candidate in that ward. Let's look at how to generate cross tabulations.

Installing the crosstab() Function

Standard ANSI SQL doesn't have a crosstab function, but PostgreSQL does as part of a *module* you can install easily. Modules are PostgreSQL extras that aren't part of the core application; they include functions related to security, text search, and more. You can find a list of PostgreSQL modules at <https://www.postgresql.org/docs/current/contrib.html>.

PostgreSQL's `crosstab()` function is part of the `tablefunc` module. To install `tablefunc`, execute this command in pgAdmin:

```
CREATE EXTENSION tablefunc;
```

PostgreSQL should return the message CREATE EXTENSION. (If you’re working with another database management system, check its documentation for a similar functionality. For example, Microsoft SQL Server has the PIVOT command.)

Next, we’ll create a basic crosstab so you can learn the syntax, and then we’ll handle a more complex case.

Tabulating Survey Results

Let’s say your company needs a fun employee activity so you coordinate an ice cream social at each of your three offices. The trouble is that people are particular about ice cream flavors. To choose flavors people will like in each office, you decide to conduct a survey.

The CSV file *ice_cream_survey.csv* contains 200 responses to your survey. You can download this file, along with all the book’s resources, at <https://nostarch.com/practical-sql-2nd-edition/>. Each row includes a `response_id`, `office`, and `flavor`. You’ll need to count how many people chose each flavor at each office and share the results in a readable way.

In your `analysis` database, use the code in [Listing 13-16](#) to create a table and load the data. Make sure you change the file path to the location on your computer where you saved the CSV file.

```
CREATE TABLE ice_cream_survey (
    response_id integer PRIMARY KEY,
    office text,
    flavor text
);

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

[Listing 13-16:](#) Creating and filling the *ice_cream_survey* table

If you want to inspect the data, run the following to view the first five rows:

```
SELECT *
FROM ice_cream_survey
ORDER BY response_id
LIMIT 5;
```

The data should look like this:

response_id	office	flavor
1	Uptown	Chocolate
2	Midtown	Chocolate
3	Downtown	Strawberry
4	Uptown	Chocolate
5	Midtown	Chocolate

It looks like chocolate is in the lead! But let's confirm this choice by using the code in [Listing 13-17](#) to generate a crosstab.

```
SELECT *
| FROM crosstab('SELECT 2 office,
|                 3 flavor,
|                 4 count(*)
|             FROM ice_cream_survey
|             GROUP BY office, flavor
|             ORDER BY office',
|
| 5 'SELECT flavor
|     FROM ice_cream_survey
|     GROUP BY flavor
|     ORDER BY flavor')
|
\$ AS (office text,
       chocolate bigint,
       strawberry bigint,
       vanilla bigint);
```

[Listing 13-17](#): Generating the ice cream survey crosstab

The query begins with a `SELECT *` statement that selects everything from the contents of the `crosstab()` function **1**. We supply two queries as parameters to the `crosstab()` function; note that because these queries are parameters, we place them inside single quotes. The first query generates the data for the crosstab and has three required columns. The first column, `office 2`, supplies the row names for the crosstab. The second column, `flavor 3`, supplies the category (or column) name to be associated with the value provided in the third column. Those values will display in each cell where a row and a column

intersect in the table. In this case, we want the intersecting cells to show a `count()` 4 of each flavor selected at each office. This first query on its own creates a simple aggregated list.

The second query parameter 5 produces the category names for the columns. The `crosstab()` function requires that the second subquery returns only one column, so we use `SELECT` to retrieve `flavor` and `GROUP BY` to return that column's unique values.

Then we specify the names and data types of the crosstab's output columns following the `AS` keyword 6. The list must match the row and column names in the order the queries generate them. For example, because the second query that supplies the category columns orders the flavors alphabetically, the output column list must as well.

When we run the code, our data displays in a clean, readable crosstab:

office	chocolate	strawberry	vanilla
Downtown	23	32	19
Midtown	41		23
Uptown	22	17	23

It's easy to see at a glance that the Midtown office favors chocolate but has no interest in strawberry, which is represented by a `NULL` value showing that strawberry received no votes. But strawberry is the top choice Downtown, and the Uptown office is more evenly split among the three flavors.

Tabulating City Temperature Readings

Let's create another crosstab, but this time we'll use real data. The `temperature_readings.csv` file, also available with all the book's resources at <https://nostarch.com/practical-sql-2nd-edition/>, contains a year's worth of daily temperature readings from three observation stations around the United States: Chicago, Seattle, and Waikiki, a neighborhood on the south shore of the city of Honolulu. The data come from the US National Oceanic and Atmospheric Administration (NOAA) at <https://www.ncdc.noaa.gov/cdo-web/datatools/findstation/>.

Each row in the CSV file contains four values: the station name, the date, and the day's maximum and minimum temperatures. All temperatures are in Fahrenheit. For each month in each city, we want to compare climates using the

median high temperature. [Listing 13-18](#) has the code to create the temperature_readings table and import the CSV file.

```
CREATE TABLE temperature_readings (
    station_name text,
    observation_date date,
    max_temp integer,
    min_temp integer,
    CONSTRAINT temp_key PRIMARY KEY (station_name,
    observation_date)
);

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

[Listing 13-18:](#) Creating and filling a temperature_readings table

The table contains the four columns from the CSV file; we add a natural primary key using the station name and observation date. A quick count should return 1,077 rows. Now, let's see what cross tabulating the data does using [Listing 13-19](#).

```
SELECT *
FROM crosstab('SELECT
    1 station_name,
    2 date_part(''month'', observation_date),
    3 percentile_cont(.5)
        WITHIN GROUP (ORDER BY max_temp)
    FROM temperature_readings
    GROUP BY station_name,
            date_part(''month'', observation_date)
    ORDER BY station_name',
    'SELECT month
    FROM 4 generate_series(1,12) month')

AS (station text,
    jan numeric(3,0),
    feb numeric(3,0),
    mar numeric(3,0),
    apr numeric(3,0),
    may numeric(3,0),
    jun numeric(3,0),
```

```

    jul numeric(3,0),
    aug numeric(3,0),
    sep numeric(3,0),
    oct numeric(3,0),
    nov numeric(3,0),
    dec numeric(3,0)
);

```

[Listing 13-19](#): Generating the temperature readings crosstab

The crosstab structure is the same as in [Listing 13-18](#). The first subquery inside `crosstab()` generates the data for the crosstab, finding the median maximum temperature for each month. It supplies three required columns. The first, `station_name` **1**, names the rows. The second column uses the `date_part()` function **2** from Chapter 12 to extract the month from `observation_date`, which provides the crosstab columns. Then we use `percentile_cont(.5)` **3** to find the 50th percentile, or the median, of the `max_temp`. We group by station name and month so we have a median `max_temp` for each month at each station.

As in [Listing 13-18](#), the second subquery produces the set of category names for the columns. I'm using a function called `generate_series()` **4** in a manner noted in the official PostgreSQL documentation to create a list of numbers from 1 to 12 that match the month numbers `date_part()` extracts from `observation_date`.

Following AS, we provide the names and data types for the crosstab's output columns. Each is a `numeric` type, matching the output of the `percentile` function. The following output is practically poetry:

station						jan	feb	mar	apr	may	jun
	jul	aug	sep	oct	nov	dec					
	---	---	---	---	---	---	---	---	---	---	---
CHICAGO	NORTHERLY	ISLAND	IL	US		34	36	46	50	66	77
81	80	77	65	57	35						
SEATTLE	BOEING	FIELD	WA	US		50	54	56	64	66	71
76	77	69	62	55	42						
WAIKIKI	717.2	HI	US			83	84	84	86	87	87
88	87	87	86	84	82						

We've transformed a raw set of daily readings into a compact table showing the median maximum temperature each month for each station. At a glance, we can see that the temperature in Waikiki is consistently balmy, whereas Chicago's median high temperatures vary from just above freezing to downright pleasant. Seattle falls between the two.

Crosstabs do take time to set up, but viewing datasets in a matrix often makes comparisons easier than viewing the same data in a vertical list. Keep in mind that the `crosstab()` function is resource-intensive, so tread carefully when querying sets that have millions or billions of rows.

Reclassifying Values with CASE

The ANSI Standard SQL `CASE` statement is a *conditional expression*, meaning it lets you add some “if this, then . . .” logic to a query. You can use `CASE` in multiple ways, but for data analysis, it’s handy for reclassifying values into categories. You can create categories based on ranges in your data and classify values according to those categories.

The `CASE` syntax follows this pattern:

```
| CASE WHEN condition THEN result
|   2 WHEN another_condition THEN result
|   3 ELSE result
| END
```

We give the `CASE` keyword **1** and then provide at least one `WHEN condition THEN result` clause, where *condition* is any expression the database can evaluate as `true` or `false`, such as `county = 'Dutchess County'` or `date > '1995-08-09'`. If the condition is `true`, the `CASE` statement returns the *result* and stops checking any further conditions. The *result* can be any valid data type. If the condition is `false`, the database moves on to evaluate the next condition.

To evaluate more conditions, we can add optional `WHEN . . . THEN` clauses **2**. We can also provide an optional `ELSE` clause **3** to return a result in case no condition evaluates as `true`. Without an `ELSE` clause, the statement would return a `NULL` when no conditions are `true`. The statement finishes with an `END` keyword **4**.

[Listing 13-20](#) shows how to use the CASE statement to reclassify the temperature readings into descriptive groups (named according to my own bias against cold weather).

```
SELECT max_temp,
       CASE WHEN max_temp >= 90 THEN 'Hot'
            WHEN max_temp >= 70 AND max_temp < 90 THEN 'Warm'
            WHEN max_temp >= 50 AND max_temp < 70 THEN
              'Pleasant'
            WHEN max_temp >= 33 AND max_temp < 50 THEN 'Cold'
            WHEN max_temp >= 20 AND max_temp < 33 THEN
              'Frigid'
            WHEN max_temp < 20 THEN 'Inhumane'
            ELSE 'No reading'
       END AS temperature_group
  FROM temperature_readings
 ORDER BY station_name, observation_date;
```

[Listing 13-20](#): Reclassifying temperature data with CASE

We create six ranges for the `max_temp` column in `temperature_readings`, which we define using comparison operators. The CASE statement evaluates each value to find whether any of the six expressions are true. If so, the statement outputs the appropriate text. Note that the ranges account for all possible values in the column, leaving no gaps. If none of the statements is true, then the `ELSE` clause assigns the value to the category `No reading`.

Run the code; the first five rows of output should look like this:

max_temp	temperature_group
31	Frigid
34	Cold
32	Frigid
32	Frigid
34	Cold
--snip--	

Now that we've collapsed the dataset into six categories, let's use those categories to compare climate among the three cities in the table.

Using CASE in a Common Table Expression

The operation we performed with CASE on the temperature data in the previous section is a good example of a preprocessing step you could use in a CTE. Now that we've grouped the temperatures in categories, let's count the groups by city in a CTE to see how many days of the year fall into each temperature category.

[Listing 13-21](#) shows the code for reclassifying the daily maximum temperatures recast to generate a temps_collapsed CTE and then use it for an analysis.

```
| WITH temps_collapsed (station_name, max_temperature_group) AS
  (SELECT station_name,
          CASE WHEN max_temp >= 90 THEN 'Hot'
                WHEN max_temp >= 70 AND max_temp < 90 THEN
                  'Warm'
                WHEN max_temp >= 50 AND max_temp < 70 THEN
                  'Pleasant'
                WHEN max_temp >= 33 AND max_temp < 50 THEN
                  'Cold'
                WHEN max_temp >= 20 AND max_temp < 33 THEN
                  'Frigid'
                WHEN max_temp < 20 THEN 'Inhumane'
                ELSE 'No reading'
              END
    FROM temperature_readings)
?
| SELECT station_name, max_temperature_group, count(*)
  FROM temps_collapsed
 GROUP BY station_name, max_temperature_group
 ORDER BY station_name, count(*) DESC;
```

[Listing 13-21: Using CASE in a CTE](#)

This code reclassifies the temperatures and then counts and groups by station name to find general climate classifications of each city. The WITH keyword defines the CTE of temps_collapsed **1**, which has two columns: station_name and max_temperature_group. We then run a SELECT query on the CTE **2**, performing straightforward count (*) and GROUP BY operations on both columns. The results should look like this:

station_name	max_temperature_group	
count		
---	-----	---
--		
CHICAGO NORTHERLY ISLAND IL US	Warm	

133		
CHICAGO NORTHERLY ISLAND IL US	Cold	
92		
CHICAGO NORTHERLY ISLAND IL US	Pleasant	
91		
CHICAGO NORTHERLY ISLAND IL US	Frigid	
30		
CHICAGO NORTHERLY ISLAND IL US	Inhumane	
8		
CHICAGO NORTHERLY ISLAND IL US	Hot	
8		
SEATTLE BOEING FIELD WA US	Pleasant	
198		
SEATTLE BOEING FIELD WA US	Warm	
98		
SEATTLE BOEING FIELD WA US	Cold	
50		
SEATTLE BOEING FIELD WA US	Hot	
3		
WAIKIKI 717.2 HI US	Warm	
361		
WAIKIKI 717.2 HI US	Hot	
5		

Using this classification scheme, the amazingly consistent Waikiki weather, with Warm maximum temperatures 361 days of the year, confirms its appeal as a vacation destination. From a temperature standpoint, Seattle looks good too, with nearly 300 days of Pleasant or Warm high temps (although this belies Seattle's legendary rainfall). Chicago, with 30 days of Frigid max temps and 8 days Inhumane, probably isn't for me.

Wrapping Up

In this chapter, you learned to make queries work harder for you. You can now add subqueries in multiple locations to provide finer control over filtering or preprocessing data before analyzing it in a main query. You also can visualize data in a matrix using cross tabulations and reclassify data into groups; both techniques give you more ways to find and tell stories using your data. Great work!

Throughout the next chapters, we'll dive into SQL techniques that are more specific to PostgreSQL. We'll begin by working with and searching text and strings.

TRY IT YOURSELF

Perform the following two tasks to help you become more familiar with the concepts introduced in the chapter:

Revise the code in [Listing 13-21](#) to dig deeper into the nuances of Waikiki's high temperatures. Limit the `temp_collapse` table to the Waikiki maximum daily temperature observations. Then use the `WHEN` clauses in the `CASE` statement to reclassify the temperatures into seven groups that would result in the following text output:

```
'90 or more'  
'88-89'  
'86-87'  
'84-85'  
'82-83'  
'80-81'  
'79 or less'
```

In which of those groups does Waikiki's daily maximum temperature fall most often?

Revise the ice cream survey crosstab in [Listing 13-17](#) to flip the table. In other words, make `flavor` the rows and `office` the columns. Which elements of the query do you need to change? Are the counts different?

14

MINING TEXT TO FIND MEANINGFUL DATA



Next, you'll learn how to use SQL to transform, search, and analyze text. You'll start with simple text wrangling using string formatting and pattern matching before moving on to more advanced analysis. We'll use two data-sets: a small collection of crime reports from a sheriff's department near Washington, DC, and a set of speeches delivered by US presidents.

Text offers plenty of possibilities for analysis. You can extract meaning from *unstructured data*—paragraphs of text in speeches, reports, press releases, and other documents—by transforming it into *structured data*, in rows and columns in a table. Or you can use advanced text analysis features, such as PostgreSQL's full-text search. With these techniques, ordinary text can reveal facts or trends that might otherwise remain hidden.

Formatting Text Using String Functions

PostgreSQL has more than 50 built-in string functions that handle routine but necessary tasks, such as capitalizing letters, combining strings, and removing unwanted spaces. Some are part of the ANSI SQL standard, and others are specific to PostgreSQL. You'll find a complete list of string functions at <https://www.postgresql.org/docs/current/functions-string.html>, but in this section we'll examine several you might use often.

You can try each of these in a simple query that places a function after `SELECT`, like this: `SELECT upper('hello');`. Examples of each function plus code for all the listings in this chapter are available at <https://nostarch.com/practical-sql-2nd-edition/>.

Case Formatting

The capitalization functions format the text's case. The `upper(string)` function capitalizes all alphabetical characters of a string passed to it. Nonalphabetic characters, such as numbers, remain unchanged. For example, `upper('Neal7')` returns `NEAL7`. The `lower(string)` function lowercases all alphabetical characters while keeping nonalphabetic characters unchanged. For example, `lower('Randy')` returns `randy`.

The `initcap(string)` function capitalizes the first letter of each word. For example, `initcap('at the end of the day')` returns `At The End Of The Day`. This function can be handy for formatting titles of books or movies, but because it doesn't recognize acronyms, it's not always the perfect solution. For example, `initcap('Practical SQL')` returns `Practical Sq1`, because it doesn't recognize SQL as an acronym.

The `upper()` and `lower()` functions are ANSI SQL standard commands, but `initcap()` is PostgreSQL-specific. These three functions give you enough options to rework a column of text into the case you prefer. Note that capitalization does not work with all locales or languages.

Character Information

Several functions return data about the string and are helpful on their own or combined with other functions. The `char_length(string)` function returns the number of characters in a string, including any spaces. For example, `char_length(' Pat ')` returns a value of 5, because the three letters in `Pat` and the spaces on either end total five characters. You can also use the non-ANSI SQL function `length(string)` to count strings, which has a variant that lets you count the length of binary strings.

NOTE

The `length()` function can return a different value than `char_length()` when used with multibyte encodings, such as character sets covering the Chinese, Japanese, or Korean languages.

The `position(substring in string)` function returns the location of the substring characters in the string. For example, `position(' , ' in 'Tan, Bella')` returns 4, because the comma and space characters (`,`) specified in the substring passed as the first parameter starting at the fourth index position in the main string `Tan, Bella`.

Both `char_length()` and `position()` are in the ANSI SQL standard.

Removing Characters

The `trim(characters from string)` function removes characters from the beginning and end of a string. To declare one or more characters to remove, add them to the function followed by the keyword `from` and the string you want to change. Options to remove leading characters (at the front of the string), trailing characters (at the end of the string), or both make this function super flexible.

For example, `trim('s' from 'socks')` removes s characters at the beginning and end, returning `ock`. To remove only the s at the end of the string, add the `trailing` keyword before the character to trim: `trim(trailing 's' from 'socks')` returns `sock`.

If you don't specify any characters to remove, `trim()` removes spaces at either end of the string by default. For example, `trim(' Pat ')` returns `Pat` without the leading or trailing spaces. To confirm the length of the trimmed string, we can nest `trim()` inside `char_length()` like this:

```
SELECT char_length(trim(' Pat '));
```

This query should return 3, the number of letters in `Pat`, which is the result of `trim(' Pat ')`.

The `ltrim(string, characters)` and `rtrim(string, characters)` functions are PostgreSQL-specific variations of the `trim()` function. They remove characters from the left or right ends of a string. For example, `rtrim('socks', 's')` returns `sock` by removing only the `s` on the right end of the string.

Extracting and Replacing Characters

The `left(string, number)` and `right(string, number)` functions, both ANSI SQL standard, extract and return selected characters from a string. For example, to get just the 703 area code from the phone number 703-555-1212, use `left('703-555-1212', 3)` to specify that you want the first three characters of the string starting from the left. Likewise, `right('703-555-1212', 8)` returns eight characters from the right: 555-1212.

To substitute characters in a string, use the `replace(string, from, to)` function. To change `bat` to `cat`, for example, you would use `replace('bat', 'b', 'c')` to specify that you want to replace the `b` in `bat` with a `c`.

Now that you know basic functions for manipulating strings, let's look at how to match more complex patterns in text and turn those patterns into data we can analyze.

Matching Text Patterns with Regular Expressions

Regular expressions (or *regex*) are a type of notational language that describes text patterns. If you have a string with a noticeable pattern (say, four digits followed by a hyphen and then two more digits), you can write a regular expression that matches the pattern. You can then use the notation in a `WHERE` clause to filter rows by the pattern or use regular expression functions to extract and wrangle text that contains the same pattern.

Regular expressions can seem inscrutable to beginning programmers; they take practice to comprehend because they use single-character symbols that aren't intuitive. Getting an expression to match a pattern can involve trial and error, and each programming language has subtle differences in the way it handles regular expressions. Still, learning regular expressions is a good investment because you gain superpower-like abilities to search text using many programming languages, text editors, and other applications.

In this section, I'll provide enough regular expression basics to work through the exercises. To learn more, I recommend interactive online code testers, such as <https://www.regexr.com/> or <https://www.regexpal.com/>, which have notation references.

Regular Expression Notation

Matching letters and numbers using regular expression notation is straightforward because letters and numbers (and certain symbols) are literals that indicate the same characters. For example, `A1` matches the first two characters in `Alicia`.

For more complex patterns, you'll use combinations of the regular expression elements in [Table 14-1](#).

Table 14-1: Regular Expression Notation Basics

Expression	Description
.	A dot is a wildcard that finds any character except a newline.
[FGz]	Any character in the square brackets. Here, F, G, or z.
[a-z]	A range of characters. Here, lowercase a to z.
[^a-z]	The caret negates the match. Here, not lowercase a to z.
\w	Any word character or underscore. Same as [A-Za-z0-9_].
\d	Any digit.
\s	A space.
\t	Tab character.
\n	Newline character.
\r	Carriage return character.
^	Match at the start of a string.
\$	Match at the end of a string.
?	Get the preceding match zero or one time.
*	Get the preceding match zero or more times.
+	Get the preceding match one or more times.
{m}	Get the preceding match exactly m times.
{m, n}	Get the preceding match between m and n times.
a b	The pipe denotes alternation. Find either a or b .
()	Create and report a capture group or set precedence.
(?:)	Negate the reporting of a capture group.

Using these regular expressions, you can match various characters and indicate how many times and where to match them. For example, placing characters inside square brackets ([]) lets you match any single character or a range. So, [FGz] matches a single F, G, or z, whereas [A-Za-z] will match any uppercase or lowercase letter.

The backslash (\) precedes a designator for special characters, such as a tab (\t), digit (\d), or newline (\n), which is a line ending character in text files.

There are several ways to indicate how many times to match a character. Placing a number inside curly brackets indicates you want to match it that many times. For example, \d{4} matches four digits in a row, and \d{1, 4} matches one to four digits.

The ?, *, and + characters provide a useful shorthand notation for the number of matches you want. The plus sign (+) after a character indicates to match it one or more times, for example. So, the expression a+ would find the aa characters in the string aardvark.

Additionally, parentheses indicate a *capture group*, which you can use to identify a portion of the entire matched expression. For example, if you were hunting for an *HH:MM:SS* time format in text and wanted to report only the hour, you could use an expression such as (\d{2}):(\d{2}):(\d{2}). This looks for two digits (\d{2}) of the hour followed by a colon,

another two digits for the minutes and a colon, and then the two-digit seconds. By placing the first `\d{2}` inside parentheses, you can extract only those two digits, even though the entire expression matches the full time.

[Table 14-2](#) shows examples of combining regular expressions to capture different portions of the sentence “The game starts at 7 p.m. on May 2, 2024.”

Table 14-2: Regular Expression Matching Examples

Expression	What it matches	Result
<code>.+</code>	Any character one or more times	The game starts at 7 p.m. on May 2, 2024.
<code>\d{1,2} (? :a.m. p.m.)</code>	One or two digits followed by a space and <code>a.m.</code> or <code>p.m.</code> in a noncapture group	7 p.m.
<code>^\w+</code>	One or more word characters at the start	The
<code>\w+.\$</code>	One or more word characters followed by any character at the end	2024.
<code>May June</code>	Either of the words <code>May</code> or <code>June</code>	May
<code>\d{4}</code>	Four digits	2024
<code>May \d, \d{4}</code>	<code>May</code> followed by a space, digit, comma, space, and four digits	May 2, 2024

These results show the usefulness of regular expressions for matching the parts of the string that interest us. For example, to find the time, we use the expression `\d{1,2} (? :a.m.|p.m.)` to look for either one or two digits because the time could be a single or double digit followed by a space. Then we look for either `a.m.` or `p.m.`; the pipe symbol separating the terms indicates the either-or condition, and placing them in parentheses separates the logic from the rest of the expression. We need the `:` symbol to indicate that we don’t want to treat the terms inside the parentheses as a capture group, which would report `a.m.` or `p.m.` only. The `:` ensures that the full match will be returned.

You can use any of these regular expressions by placing the text and regular expression inside the `substring(string from pattern)` function to return the matched text. For example, to find the four-digit year, use the following query:

```
SELECT substring('The game starts at 7 p.m. on May 2, 2024.' from '\d{4}');
```

This query should return 2024, because we specified that the pattern should look for four digits in a row, and 2024 is the only portion of this string that matches these criteria. You can check out sample `substring()` queries for all the examples in [Table 14-2](#) in the book’s code resources at <https://nostarch.com/practical-sql-2nd-edition/>.

Using Regular Expressions with WHERE

You’ve filtered queries using `LIKE` and `ILIKE` in `WHERE` clauses. In this section, you’ll learn to use regular expressions in `WHERE` clauses so you can perform more complex matches.

We use a tilde (~) to make a case-sensitive match on a regular expression and a tilde-asterisk (~*) to perform a case-insensitive match. You can negate either expression by adding an exclamation point in front. For example, !~* indicates to *not* match a regular expression that is case-insensitive. [Listing 14-1](#) shows how this works using the 2019 US Census estimates us_counties_pop_est_2019 table from previous exercises.

```
SELECT county_name
  FROM us_counties_pop_est_2019
 WHERE county_name ~* '(lade|lare)'
 ORDER BY county_name;

SELECT county_name
  FROM us_counties_pop_est_2019
 WHERE county_name ~* 'ash' AND county_name !~ 'Wash'
 ORDER BY county_name;
```

[Listing 14-1](#): Using regular expressions in a WHERE clause

The first WHERE clause **1** uses the tilde-asterisk (~*) to perform a case-insensitive match on the regular expression (lade|lare) to find any county names that contain either the letters lade or lare. The results should show eight rows:

county_name

Bladen County
Clare County
Clarendon County
Glades County
Langlade County
Philadelphia County
Talladega County
Tulare County

As you can see, each county name includes the letters lade or lare.

The second WHERE clause **2** uses the tilde-asterisk (~*) as well as a negated tilde (!~) to find county names containing the letters ash but excluding those that include wash. This query should return the following:

county_name

Ashe County
Ashland County
Ashland County
Ashley County
Ashtabula County
Nash County
Wabash County
Wabash County
Wabasha County

All nine counties in this output have names that contain the letters ash, but none have wash.

These are fairly simple examples, but you can do more complex matches using regular expressions that you wouldn't be able to perform with the wildcards available with just `LIKE` and `ILIKE`.

Regular Expression Functions to Replace or Split Text

[Listing 14-2](#) shows three regular expression functions that replace and split text.

```
| SELECT regexp_replace('05/12/2024', '\d{4}', '2023');  
|  
2 SELECT regexp_split_to_table('Four,score,and,seven,years,ago', ',' );  
|  
3 SELECT regexp_split_to_array('Phil Mike Tony Steve', ' ' );
```

Listing 14-2: Regular expression functions to replace and split text

The `regexp_replace(string, pattern, replacement text)` function lets you substitute a matched pattern with replacement text. In the example at 1, we're searching the date string 05/12/2024 for any set of four digits in a row using `\d{4}`. When found, we replace them with the replacement text 2023. The result of that query is 05/12/2023 returned as text.

The `regexp_split_to_table(string, pattern)` function splits delimited text into rows.

[Listing 14-2](#) uses this function to split the string 'Four,score,and,seven,years,ago' on commas 2, resulting in a set of rows that has one word in each row:

```
regexp_split_to_table  
-----  
Four  
score  
and  
seven  
years  
ago
```

Keep this function in mind as you tackle the “Try It Yourself” exercises at the end of the chapter.

The `regexp_split_to_array(string, pattern)` function splits delimited text into an array. The example splits the string Phil Mike Tony Steve on spaces 3, returning a text array that should look like this in pgAdmin:

```
regexp_split_to_array  
-----  
{Phil,Mike,Tony,Steve}
```

The `text[]` notation in pgAdmin's column header along with curly brackets around the results confirms that this is indeed an array type, which provides another means of analysis. For example, you could then use a function such as `array_length()` to count the number of words, as shown in [Listing 14-3](#).

```
SELECT array_length(regexp_split_to_array('Phil Mike Tony Steve', ' '), 1);
```

[Listing 14-3](#): Finding an array length

The array that `regexp_split_to_array()` produces is one-dimensional; that is, the result contains one list of names. Arrays can have additional dimensions—for example, a two-dimensional array can represent a matrix with rows and columns. Thus, here we pass 1 as a second argument 1 to `array_length()`, indicating we want the length of the first (and only) dimension of the array. The query should return 4 because the array has four elements. You can read more about `array_length()` and other array functions at <https://www.postgresql.org/docs/current/functions-array.html>.

If you can identify a pattern in the text, you can use a combination of regular expression symbols to locate it. This technique is particularly useful when you have repeating patterns in text that you want to turn into a set of data to analyze. Let's practice how to use regular expression functions using a real-world example.

Turning Text to Data with Regular Expression Functions

A sheriff's department in one of the Washington, DC, suburbs publishes daily reports that detail the date, time, location, and description of incidents the department investigates. These reports would be great to analyze, except they post the information in Microsoft Word documents saved as PDF files, which is not the friendliest format for importing into a database.

If I copy and paste incidents from the PDF into a text editor, the result is blocks of text that look something like [Listing 14-4](#).

```
| 4/16/17-4/17/17
| 2100-0900 hrs.
| 46000 Block Ashmere Sq.
| Sterling
| Larceny: 6 The victim reported that a
| bicycle was stolen from their opened
| garage door during the overnight hours.
| C0170006614

04/10/17
1605 hrs.
21800 block Newlin Mill Rd.
Middleburg
Larceny: A license plate was reported
stolen from a vehicle.
SO170006250
```

[Listing 14-4](#): Crime reports text

Each block of text includes dates 1, times 2, a street address 3, city or town 4, the type of crime 5, and a description of the incident 6. The last piece of information is a code 7 that might be a

unique ID for the incident, although we'd have to check with the sheriff's department to be sure. There are slight inconsistencies. For example, the first block of text has two dates (4/16/17-4/17/17) and two times (2100-0900 hrs.), meaning the exact time of the incident is unknown and likely occurred within that time span. The second block has one date and time.

If you compile these reports regularly, you can expect to find some good insights that could answer important questions: Where do crimes tend to occur? Which crime types occur most frequently? Do they happen more often on weekends or weekdays? Before you can start answering these questions, you'll need to extract the text into table columns using regular expressions.

NOTE

Extracting elements from text is labor-intensive, so it's a good idea to ask the owner of the data whether the text was produced from a database. If it was and you can obtain a structured export such as a CSV file from that database, you'll save considerable time.

Creating a Table for Crime Reports

I've collected five of the crime incidents into a file named *crime_reports.csv* that you can download via the link to the book's resources at <https://nostarch.com/practical-sql-2nd-edition/>. Download the file and save it on your computer. Then use the code in [Listing 14-5](#) to build a table that has a column for each data element you can parse from the text using a regular expression.

```
CREATE TABLE crime_reports (
    crime_id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    case_number text,
    date_1 timestamp,
    date_2 timestamp,
    street text,
    city text,
    crime_type text,
    description text,
    original_text text NOT NULL
);

COPY crime_reports (original_text)
FROM 'C:\YourDirectory\crime_reports.csv'
WITH (FORMAT CSV, HEADER OFF, QUOTE ''');
```

[Listing 14-5](#): Creating and loading the *crime_reports* table

Run the CREATE TABLE statement in [Listing 14-5](#) and then use COPY to load the text into the column `original_text`. The rest of the columns will be NULL until we fill them.

When you run `SELECT original_text FROM crime_reports;` in pgAdmin, the results grid should display five rows and the first several words of each report. When you double-click any cell, pgAdmin shows all the text in that row, as shown in [Figure 14-1](#).

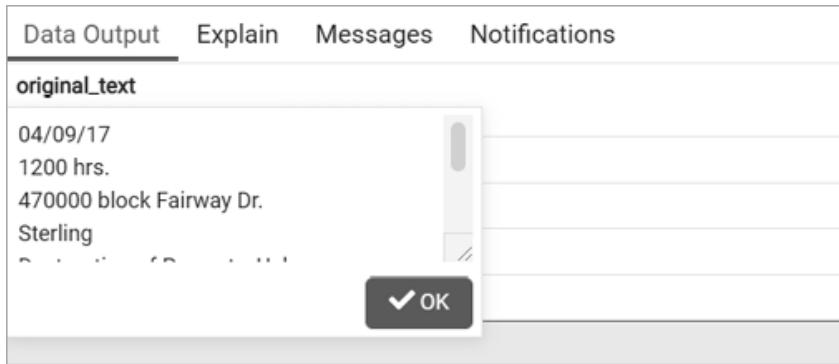


Figure 14-1: Displaying additional text in the pgAdmin results grid

Now that you've loaded the text you'll be parsing, let's explore this data using PostgreSQL regular expression functions.

Matching Crime Report Date Patterns

The first piece of data we want to extract from `original_text` is the date or dates of the crime. Most reports have one date, although one has two. The reports also have associated times, and we'll combine the extracted date and time into a timestamp. We'll fill `date_1` with each report's first (or only) date and time. If a second date or second time exists, we'll add it to `date_2`.

We'll use the `regexp_match(string, pattern)` function, which is similar to `substring()` with a few exceptions. One is that it returns each match as text in an array. Also, if there are no matches, it returns `NULL`. As you might recall from Chapter 6, you use an array to pass a list of values into the `percentile_cont()` function to calculate quartiles. I'll show you how to work with results that come back as an array when we parse the crime reports.

NOTE

The `regexp_match()` function was introduced in PostgreSQL 10 and is not available in earlier versions.

To start, let's use `regexp_match()` to find dates in each of the five incidents. The general pattern to match is `MM/DD/YY`, although there may be one or two digits for both the month and date. Here's a regular expression that matches the pattern:

```
\d{1,2}\/\d{1,2}\/\d{2}
```

In this expression, the first `\d{1,2}` indicates the month. The numbers inside the curly brackets specify that you want at least one digit and at most two digits. Next, you want to look for a forward slash (/), but because a forward slash can have special meaning in regular expressions, you must *escape* that character by placing a backslash (\) in front of it, like this `\/`. Escaping a character in this context simply means we want to treat it as a literal rather than letting it take on special meaning. So, the combination of the backslash and forward slash (`\/`) indicates you want a forward slash.

Another `\d{1,2}` follows for a single- or double-digit day of the month. The expression ends with a second escaped forward slash and `\d{2}` to indicate the two-digit year. Let's pass the expression `\d{1,2}\/\d{1,2}\/\d{2}` to `regexp_match()`, as shown in [Listing 14-6](#).

```
SELECT crime_id,
       regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}')
  FROM crime_reports
 ORDER BY crime_id;
```

[Listing 14-6](#): Using `regexp_match()` to find the first date

Run that code in pgAdmin, and the results should look like this:

crime_id	regexp_match
1	{4/16/17}
2	{4/8/17}
3	{4/4/17}
4	{04/10/17}
5	{04/09/17}

Note that each row shows the first date listed for the incident, because `regexp_match()` returns the first match it finds by default. Also note that each date is enclosed in curly brackets. That's PostgreSQL indicating that `regexp_match()` returns each result as an array type, or list of elements. Later, in the "Extracting Text from the `regexp_match()` Result" section, I'll show you how to access elements in the array. You also can read more about arrays in PostgreSQL at <https://www.postgresql.org/docs/current/arrays.html>.

Matching the Second Date When Present

We've successfully matched the first date from each report. But recall that one of the five incidents has a second date. To find and display all the dates in the text, you must use the related `regexp_matches()` function and pass in an option in the form of the flag `g`, as shown in [Listing 14-7](#).

```
SELECT crime_id,
       regexp_matches(original_text, '\d{1,2}\/\d{1,2}\/\d{2}', 'g'1)
  FROM crime_reports
 ORDER BY crime_id;
```

[Listing 14-7](#): Using the `regexp_matches()` function with the `g` flag

The `regexp_matches()` function, when supplied the `g` flag `1`, differs from `regexp_match()` by returning each match the expression finds as a row in the results rather than returning just the first match.

Run the code again with this revision; you should now see two dates for the incident that has a `crime_id` of 1, like this:

crime_id	regexp_matches
1	{4/16/17}
1	{4/17/17}
2	{4/8/17}
3	{4/4/17}
4	{04/10/17}
5	{04/09/17}

Any time a crime report has a second date, we want to load it and the associated time into the date_2 column. Although adding the g flag shows us all the dates, to extract just the second date in a report, we can use the pattern we always see when two dates exist. In [Listing 14-4](#), the first block of text showed the two dates separated by a hyphen, like this:

4/16/17-4/17/17

This means you can switch back to `regexp_match()` and write a regular expression to look for a hyphen followed by a date, as shown in [Listing 14-8](#).

```
SELECT crime_id,
       regexp_match(original_text, '-\d{1,2}\/\d{1,2}\/\d{2}')
  FROM crime_reports
 ORDER BY crime_id;
```

[Listing 14-8](#): Using `regexp_match()` to find the second date

Although this query finds the second date in the first item (and returns a NULL for the rest), there's an unintended consequence: it displays the hyphen along with it.

crime_id	regexp_match
1	{-4/17/17}
2	
3	
4	
5	

You don't want to include the hyphen, because it's an invalid format for the `timestamp` data type. Fortunately, you can specify the exact part of the regular expression you want to return by placing parentheses around it to create a capture group, like this:

- (\d{1,2}\/\d{1,2}\/\d{2})

This notation returns only the part of the regular expression you want. Run the modified query in [Listing 14-9](#) to report only the data in parentheses.

```
SELECT crime_id,
       regexp_match(original_text, '-(\d{1,2}\/\d{1,2}\/\d{2})')
  FROM crime_reports
 ORDER BY crime_id;
```

[Listing 14-9](#): Using a capture group to return only the date

The query in [Listing 14-9](#) should return just the second date without the leading hyphen, as shown here:

crime_id	regexp_match
1	{4/17/17}
2	
3	
4	
5	

The process you've just completed is typical. You start with text to analyze and then write and refine the regular expression until it finds the data you want. So far, we've created regular expressions to match the first date and a second date, if it exists. Now, let's use regular expressions to extract additional data elements.

Matching Additional Crime Report Elements

In this section, we'll capture times, addresses, crime type, description, and case number from the crime reports. Here are the expressions for capturing this information:

First hour `\/\d{2}\n(\d{4})`

The first hour, which is the hour the crime was committed or the start of the time range, always follows the date in each crime report, like this:

```
4/16/17-4/17/17
2100-0900 hrs.
```

To find the first hour, we start with an escaped forward slash and `\d{2}`, which represents the two-digit year preceding the first date (17). The `\n` character indicates the newline because the hour always starts on a new line, and `\d{4}` represents the four-digit hour (2100). Because we just want to return the four digits, we put `\d{4}` inside parentheses as a capture group.

Second hour `\/\d{2}\n\d{4}-(\d{4})`

If the second hour exists, it will follow a hyphen, so we add a hyphen and another `\d{4}` to the expression we just created for the first hour. Again, the second `\d{4}` goes inside a capture group, because 0900 is the only hour we want to return.

Street hrs. `\n(\d+ .+(?:Sq.|Plz.|Dr.|Ter.|Rd.))`

In this data, the street always follows the time's `hrs.` designation and a newline (`\n`), like this:

```
04/10/17
1605 hrs.
21800 block Newlin Mill Rd.
```

The street address always starts with some number that varies in length and ends with an abbreviated suffix of some kind. To describe this pattern, we use `\d+` to match any digit that appears one or more times. Then we specify a space and look for any character one or more times using the dot wildcard and plus sign (`.+`) notation. The expression ends with a series of terms separated by the alternation pipe symbol that looks like this:

`(?:Sq.|Plz.|Dr.|Ter.|Rd.).` The terms are inside parentheses, so the expression will match one or another of those terms. When we group terms like this, if we don't want the parentheses to act as a capture group, we need to add `?:` to negate that effect.

NOTE

In a large dataset, it's likely roadway names would end with suffixes beyond the five in our regular expression. After making an initial pass at extracting the street, you can run a query to check for unmatched rows to find additional suffixes to match.

City `(?:Sq.|Plz.|Dr.|Ter.|Rd.)\n(\w+ \w+|\w+)\n`

Because the city always follows the street suffix, we reuse the terms separated by the alternation symbol we just created for the street. We follow that with a newline (`\n`) and then use a capture group to look for two words or one word `(\w+ \w+|\w+)` before a final newline, because a town or city name can be more than a single word.

Crime type `\n(?:\w+ \w+|\w+)\n(.*):`

The type of crime always precedes a colon (the only time a colon is used in each report) and might consist of one or more words, like this:

```
--snip--  
Middleburg  
Larceny: A license plate was reported  
stolen from a vehicle.  
SO170006250  
--snip--
```

To create an expression that matches this pattern, we follow a newline with a nonreporting capture group that looks for the one- or two-word city. Then we add another newline and match any character that occurs zero or more times before a colon using `(.*):`.

Description `:\s(.+)(?:C0|SO)`

The crime description always comes between the colon after the crime type and the case number. The expression starts with the colon, a space character (`\s`), and then a capture group to find any character that appears one or more times using the `.+` notation. The nonreporting capture group `(?:C0|SO)` tells the program to stop looking when it encounters either `C0` or `SO`, the two character pairs that start each case number (a `C` followed by a zero, and an `S` followed by a capital `O`). We have to do this because the description might have one or more line breaks.

Case number (? :c0 | so) [0-9] +

The case number starts with either c0 or so, followed by a set of digits. To match this pattern, the expression looks for either c0 or so in a nonreporting capture group followed by any digit from 0 to 9 that occurs one or more times using the [0-9] range notation.

Now let's pass some of these regular expressions to `regexp_match()` to see them in action.

[Listing 14-10](#) shows a sample `regexp_match()` query that retrieves the case number, first date, crime type, and city.

```
SELECT
    regexp_match(original_text, '(?:C0|SO)[0-9]+') AS case_number,
    regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}') AS date_1,
    regexp_match(original_text, '\n(?:\w+\ \w+|\w+)\n(.*):') AS crime_type,
    regexp_match(original_text, '(?:Sq.|Plz.|Dr.|Ter.|Rd.)\n(\w+
\w+|\w+)\n') AS city
FROM crime_reports
ORDER BY crime_id;
```

[Listing 14-10:](#) Matching case number, date, crime type, and city

Run the code, and the results should look like this:

case_number	date_1	crime_type	city
{C0170006614}	{4/16/17}	{Larceny}	{Sterling}
{C0170006162}	{4/8/17}	{"Destruction of Property"}	{Sterling}
{C0170006079}	{4/4/17}	{Larceny}	{Sterling}
{SO170006250}	{04/10/17}	{Larceny}	{Middleburg}
{SO170006211}	{04/09/17}	{"Destruction of Property"}	{Sterling}

After all that wrangling, we've transformed the text into a structure that is more suitable for analysis. Of course, you would have to include many more incidents to count the frequency of crime type by city or by the number of crimes per month to identify any trends.

To load each parsed element into the table's columns, we'll create an `UPDATE` query. But before you can insert the text into a column, you'll need to learn how to extract the text from the array that `regexp_match()` returns.

Extracting Text from the `regexp_match()` Result

In "Matching Crime Report Date Patterns," I mentioned that `regexp_match()` returns data in an array type containing text. Two clues reveal that these are array types. The first is that the data type designation in the column header shows `text[]` instead of `text`. The second is that each result is surrounded by curly brackets. [Figure 14-2](#) shows how pgAdmin displays the results of the query in [Listing 14-10](#).

	case_number	date_1	crime_type	city
	text[]	text[]	text[]	text[]
1	{C0170006614}	{4/16/17}	{Larceny}	{Sterling}
2	{C0170006162}	{4/8/17}	{"Destruction of Property"}	{Sterling}
3	{C0170006079}	{4/4/17}	{Larceny}	{Sterling}
4	{SO170006250}	{04/10/17}	{Larceny}	{Middleburg}
5	{SO170006211}	{04/09/17}	{"Destruction of Property"}	{Sterling}

Figure 14-2: Array values in the pgAdmin results grid

The `crime_reports` columns we want to update are not array types, so rather than passing in the array values returned by `regexp_match()`, we need to extract the values from the array first. We do this by using array notation, as shown in [Listing 14-11](#).

```

SELECT
    crime_id,
    1 (regexp_match(original_text, '(?:C0|SO) [0-9]+')) [1] 2
        AS case_number
FROM crime_reports
ORDER BY crime_id;

```

Listing 14-11: Retrieving a value from within an array

First, we wrap the `regexp_match()` function 1 in parentheses. Then, at the end, we provide a value of 1, which represents the first element in the array, enclosed in square brackets 2. The query should produce the following results:

crime_id	case_number
1	C0170006614
2	C0170006162
3	C0170006079
4	SO170006250
5	SO170006211

Now the data type designation in the pgAdmin column header should show `text` instead of `text[]`, and the values are no longer enclosed in curly brackets. We can now insert these values into `crime_reports` using an `UPDATE` query.

Updating the `crime_reports` Table with Extracted Data

To start updating columns in `crime_reports`, [Listing 14-12](#) combines the extracted first date and time into a single `timestamp` value for the column `date_1`.

```

UPDATE crime_reports
SET date_1 =

```

```

(
 2 (regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}') )[1]
 3 || ' ' ||
 4 (regexp_match(original_text, '\/\d{2}\n(\d{4})')) [1]
 5 ||' US/Eastern'
) ::timestampz
RETURNING crime_id, date_1, original_text;

```

[Listing 14-12:](#) Updating the `crime_reports` `date_1` column

Because the `date_1` column is of type `timestamp`, we must provide an input in that data type. To do that, we'll use the PostgreSQL double-pipe (`||`) concatenation operator to combine the extracted date and time in a format that's acceptable for `timestamp` with time zone input. In the `SET` clause **1**, we start with the regex pattern that matches the first date **2**. Next, we concatenate the date with a space using two single quotes **3** and repeat the concatenation operator. This step combines the date with a space before connecting it to the regex pattern that matches the time **4**. Then we include the time zone for the Washington, DC, area by concatenating that at the end of the string **5** using the `US/Eastern` designation. Concatenating these elements creates a string in the pattern of `MM/DD/YY HH:MM TIMEZONE`, which is acceptable as a `timestamp` input. We cast the string to a `timestamp` with time zone data type **6** using the PostgreSQL double-colon shorthand and the `timestampz` abbreviation.

When you run the `UPDATE`, the `RETURNING` clause will display the columns we specify from the updated rows, including the now-filled `date_1` column alongside a portion of the `original_text` column, like this:

crime_id	date_1	original_text
-	-	-
1	2017-04-16 21:00:00-04	4/16/17-4/17/17 2100-0900 hrs. 46000 Block Ashmere Sq. Sterling Larceny: The victim reported that a bicycle was stolen from their opened garage door during the overnight hours. C0170006614
2	2017-04-08 16:00:00-04	4/8/17 1600 hrs. 46000 Block Potomac Run Plz. Sterling Destruction of Property: The victim reported that their vehicle was spray painted and the trim was ripped off while it was parked at this location. C0170006162
<i>--snip--</i>		

At a glance, you can see that `date_1` accurately captures the first date and time that appears in the original text and puts it into a format that we can analyze—quantifying, for example, which

times of day crimes most often occur. Note that if you're not in the Eastern time zone, the timestamps will instead reflect your pgAdmin client's time zone. Also, in pgAdmin, you may need to double-click a cell in the `original_text` column to see the full text.

Using CASE to Handle Special Instances

You could write an UPDATE statement for each remaining data element, but combining those statements into one would be more efficient. [Listing 14-13](#) updates all the `crime_reports` columns using a single statement while handling inconsistent values in the data.

```
UPDATE crime_reports
SET date_11 =
  (
    (regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}') [1]
     || ' '
     || (regexp_match(original_text, '\/\d{2}\n(\d{4})') [1]
         || 'US/Eastern'
        )::timestampz,
    date_22 =
      CASE3
        WHEN4 (SELECT regexp_match(original_text, '-'
          (\d{1,2}\/\d{1,2}\/\d{2})) IS NULL5)
          AND (SELECT regexp_match(original_text, '\/\d{2}\n\d{4}-
          (\d{4})') IS NOT NULL6)
        THEN7
          ((regexp_match(original_text, '\d{1,2}\/\d{1,2}\/\d{2}') [1]
           || ' '
           || (regexp_match(original_text, '\/\d{2}\n\d{4}-
           (\d{4})') [1]
               || 'US/Eastern'
              )::timestampz
            )
        WHEN8 (SELECT regexp_match(original_text, '-'
          (\d{1,2}\/\d{1,2}\/\d{2})) IS NOT NULL)
          AND (SELECT regexp_match(original_text, '\/\d{2}\n\d{4}-
          (\d{4})') IS NOT NULL)
        THEN
          ((regexp_match(original_text, '-(\d{1,2}\/\d{1,2}\/\d{1,2})) [1]
           || ' '
           || (regexp_match(original_text, '\/\d{2}\n\d{4}-
           (\d{4})') [1]
               || 'US/Eastern'
              )::timestampz
            )
        END,
        street = (regexp_match(original_text, 'hrs.\n(\d+ .+
        ?:Sq.|Plz.|Dr.|Ter.|Rd.)')) [1],
        city = (regexp_match(original_text,
          '(:Sq.|Plz.|Dr.|Ter.|Rd.)\n(\w+ \w+|\w+)\n')) [1],
        crime_type = (regexp_match(original_text, '\n(?:\w+ \w+|\w+)\n(.*)')) [1],
        description = (regexp_match(original_text, ':s(.+) (?:(C0|SO)')) [1],
        case_number = (regexp_match(original_text, '(?:C0|SO) [0-9]+')) [1];
  );
```

[Listing 14-13](#): Updating all crime_reports columns

This UPDATE statement might look intimidating, but it's not if we break it down by column. First, we use the same code from [Listing 14-9](#) to update the date_1 column **1**. But to update date_2 **2**, we need to account for the inconsistent presence of a second date and time. In our limited dataset, there are three possibilities:

A second hour exists but not a second date. This occurs when a report covers a range of hours on one date.

A second date and second hour exist. This occurs when a report covers more than one date.

Neither a second date nor a second hour exists.

To insert the correct value in date_2 for each scenario, we use a CASE statement to test for each possibility. After the CASE keyword **3**, we use a series of WHEN ... THEN statements to check for the first two conditions and provide the value to insert; if neither condition exists, the CASE statement will by default return a NULL.

The first WHEN statement **4** checks whether regexp_match() returns a NULL **5** for the second date and a value for the second hour (using IS NOT NULL **6**). If that condition evaluates as true, the THEN statement **7** concatenates the first date with the second hour to create a timestamp for the update.

The second WHEN statement **8** checks that regexp_match() returns a value for the second hour and second date. If true, the THEN statement concatenates the second date with the second hour to create a timestamp.

If neither of the two WHEN statements returns true, the CASE statement will return a NULL because there is only a first date and first time.

NOTE

The WHEN statements handle the possibilities that exist in our small sample dataset. If you are working with more data, you might need to handle additional variations, such as a second date but not a second time.

When we run the full query in [Listing 14-13](#), PostgreSQL should report UPDATE 5. Success! Now that we've updated all the columns with the appropriate data while accounting for elements that have additional data, we can examine all the columns of the table and find the parsed elements from original_text. [Listing 14-14](#) queries four of the columns.

```
SELECT date_1,
       street,
       city,
       crime_type
  FROM crime_reports
 ORDER BY crime_id;
```

[Listing 14-14](#): Viewing selected crime data

The results of the query should show a nicely organized set of data that looks something like this:

date_1	street	city
crime_type		
2017-04-16 21:00:00-04	46000 Block Ashmere Sq.	Sterling
Larceny		
2017-04-08 16:00:00-04	46000 Block Potomac Run Plz.	Sterling
Destruction of ...		
2017-04-04 14:00:00-04	24000 Block Hawthorn Thicket Ter.	Sterling
Larceny		
2017-04-10 16:05:00-04	21800 block Newlin Mill Rd.	Middleburg
Larceny		
2017-04-09 12:00:00-04	470000 block Fairway Dr.	Sterling
Destruction of ...		

You've successfully transformed raw text into a table that can answer questions and reveal storylines about crime in this area.

The Value of the Process

Writing regular expressions and coding a query to update a table can take time, but there is value to identifying and collecting data this way. In fact, some of the best datasets you'll encounter are those you build yourself. Everyone can download the same datasets, but the ones you build are yours alone. You get to be first person to find and tell the story behind the data.

Also, after you set up your database and queries, you can use them again and again. In this example, you could collect crime reports every day (either by hand or by automating downloads using a programming language such as Python) for an ongoing dataset that you can mine continually for trends.

In the next section, we'll continue our exploration of text by implementing a search engine using PostgreSQL.

Full-Text Search in PostgreSQL

PostgreSQL comes with a powerful full-text search engine that adds capabilities for searching large amounts of text, similar to online search tools and technology that powers search on research databases, such as Factiva. Let's walk through a simple example of setting up a table for text search and associated search functions.

For this example, I assembled 79 speeches by US presidents since World War II. Consisting mostly of State of the Union addresses, these public texts are available through the Internet Archive at <https://archive.org/> and the University of California's American Presidency Project at <https://www.presidency.ucsb.edu/>. You can find the data in the *president_speeches.csv* file along with the book's resources at <https://nostarch.com/practical-sql-2nd-edition/>.

Let's start with the data types unique to full-text search.

Text Search Data Types

PostgreSQL's implementation of text search includes two data types. The `tsvector` data type represents the text to be searched and to be stored in a normalized form. The `tsquery` data type represents the search query terms and operators. Let's look at the details of both.

Storing Text as Lexemes with `tsvector`

The `tsvector` data type reduces text to a sorted list of *lexemes*, which are linguistic units in a given language. It's helpful to think of lexemes as word roots without the variations created by suffixes. For example, a `tsvector` type column would store the words *washes*, *washed*, and *washing* as the lexeme *wash* while noting each word's position in the original text. Converting text to `tsvector` also removes small *stop words* that usually don't play a role in search, such as *the* or *it*.

To see how this data type works, let's convert a string to `tsvector` format. [Listing 14-15](#) uses the PostgreSQL search function `to_tsvector()`, which normalizes the text "I am walking across the sitting room to sit with you" to lexemes using the `english` language search configuration.

```
SELECT to_tsvector('english', 'I am walking across the sitting room to sit with you.');
```

[Listing 14-15](#): Converting text to `tsvector` data

Execute the code, and it should return the following output in the `tsvector` data type:

```
'across':4 'room':7 'sit':6,9 'walk':3
```

The `to_tsvector()` function reduces the number of words from eleven to four, eliminating words such as *I*, *am*, and *the*, which are not helpful search terms. The function removes suffixes, changing *walking* to *walk* and *sitting* to *sit*. It orders the words alphabetically, and the number following each colon indicates its position in the original string, taking stop words into account. Note that *sit* is recognized as being in two positions, one for *sitting* and one for *sit*.

NOTE

To see additional search language configurations installed with PostgreSQL, you can run the query

```
SELECT cfgname FROM pg_ts_config;
```

Creating the Search Terms with `tsquery`

The `tsquery` data type represents the full-text search query, again optimized as lexemes. It also provides operators for controlling the search. Examples of operators include the ampersand (&) for AND, the pipe symbol (|) for OR, and the exclamation point (!) for NOT. The <-> followed by operator lets you search for adjacent words or words a certain distance apart.

[Listing 14-16](#) shows how the `to_tsquery()` function converts search terms to the `tsquery` data type.

```
SELECT to_tsquery('english', 'walking & sitting');
```

[Listing 14-16](#): Converting search terms to tsquery data

After running the code, you should see that the resulting `tsquery` data type has normalized the terms into lexemes, which match the format of the data to search:

```
'walk' & 'sit'
```

Now you can use terms stored as `tsquery` to search text optimized as `tsvector`.

Using the @@ Match Operator for Searching

With the text and search terms converted to the full-text search data types, you can use the double at sign (@@) match operator to check whether a query matches text. The first query in [Listing 14-17](#) uses `to_tsquery()` to evaluate whether the text contains both `walking` and `sitting`, which we combine with the & operator. It returns a Boolean value of `true` because the lexemes of both `walking` and `sitting` are present in the text converted by `to_tsvector()`.

```
SELECT to_tsvector('english', 'I am walking across the sitting room') @@ to_tsquery('english', 'walking & sitting');
```

```
SELECT to_tsvector('english', 'I am walking across the sitting room') @@ to_tsquery('english', 'walking & running');
```

[Listing 14-17](#): Querying a `tsvector` type with a `tsquery`

However, the second query returns `false` because both `walking` and `running` are not present in the text. Now let's build a table for searching the speeches.

Creating a Table for Full-Text Search

The code in [Listing 14-18](#) creates and fills `president_speeches` with a column for the original text as well as a column of type `tsvector`. After the import, we'll convert the speech text to the `tsvector` data type. Note that to accommodate how I set up the CSV file, the `WITH` clause in `COPY` has a different set of parameters than what we've generally used. It's pipe-delimited and uses an ampersand for quoting.

```
CREATE TABLE president_speeches (
    president text NOT NULL,
    title text NOT NULL,
    speech_date date NOT NULL,
    speech_text text NOT NULL,
    search_speech_text tsvector,
    CONSTRAINT speech_key PRIMARY KEY (president, speech_date)
);

COPY president_speeches (president, title, speech_date, speech_text)
FROM 'C:\YourDirectory\president_speeches.csv'
WITH (FORMAT CSV, DELIMITER '|', HEADER OFF, QUOTE '@');
```

[Listing 14-18](#): Creating and filling the president_speeches table

After executing the query, run `SELECT * FROM president_speeches;` to see the data. In pgAdmin, double-click any cell to see extra words not visible in the results grid. You should see a sizable amount of text in each row of the `speech_text` column.

Next, we use the `UPDATE` query in [Listing 14-19](#) to copy the contents of `speech_text` to the `tsvector` column `search_speech_text` and transform it to that data type at the same time:

```
UPDATE president_speeches
| SET search_speech_text = to_tsvector('english', speech_text);
```

[Listing 14-19](#): Converting speeches to tsvector in the search_speech_text column

The `SET` clause 1 fills `search_speech_text` with the output of `to_tsvector()`. The first argument in the function specifies the language for parsing the lexemes. We’re using `english` here, but you can substitute `spanish`, `german`, `french`, and other languages (some languages may require you to find and install additional dictionaries). Using `simple` for the language will remove stop words but not reduce words to lexemes. The second argument is the name of the input column. Run the code to fill the column.

Finally, we want to index the `search_speech_text` column to speed up searches. You learned about indexing in Chapter 8, which focused on PostgreSQL’s default index type, B-tree. For full-text search, the PostgreSQL documentation recommends using the *generalized inverted index (GIN)*. A GIN index, according to the documentation, contains “an index entry for each word (lexeme), with a compressed list of matching locations.” See <https://www.postgresql.org/docs/current/textsearch-indexes.html> for details.

You can add a GIN index using `CREATE INDEX` in [Listing 14-20](#).

```
CREATE INDEX search_idx ON president_speeches USING
gin(search_speech_text);
```

[Listing 14-20](#): Creating a GIN index for text search

Now you’re ready to use search functions.

NOTE

Another way to set up a column for search is to create an index on a text column using the `to_tsvector()` function. See <https://www.postgresql.org/docs/current/textsearch-tables.html> for details.

Searching Speech Text

Nearly 80 years’ worth of presidential speeches is fertile ground for exploring history. For example, the query in [Listing 14-21](#) lists the speeches in which the president discussed Vietnam.

```
SELECT president, speech_date
FROM president_speeches
| WHERE search_speech_text @@ to_tsquery('english', 'Vietnam')
ORDER BY speech_date;
```

[Listing 14-21](#): Finding speeches containing the word Vietnam

In the `WHERE` clause, the query uses the double at sign (`@@`) match operator **1** between the `search_speech_text` column (of data type `tsvector`) and the query term *Vietnam*, which `to_tsquery()` transforms into `tsquery` data. The results should list 19 speeches, showing that the first mention of Vietnam came up in a 1961 special message to Congress by John F. Kennedy and became a recurring topic starting in 1966 as America's involvement in the Vietnam War escalated.

president	speech_date
John F. Kennedy	1961-05-25
Lyndon B. Johnson	1966-01-12
Lyndon B. Johnson	1967-01-10
Lyndon B. Johnson	1968-01-17
Lyndon B. Johnson	1969-01-14
Richard M. Nixon	1970-01-22
Richard M. Nixon	1972-01-20
Richard M. Nixon	1973-02-02
Gerald R. Ford	1975-01-15
--snip--	

Before we try more searches, let's add a method for showing the location of our search term in the text.

Showing Search Result Locations

To see where our search terms appear in text, we can use the `ts_headline()` function. It displays one or more highlighted search terms surrounded by adjacent words with options to format the display, the number of words to show around the matched search term, and how many matched results to show from each row of text. [*Listing 14-22*](#) highlights how to display a search for a specific instance of the word *tax* using `ts_headline()`.

```
SELECT president,
       speech_date,
1 ts_headline(speech_text, to_tsquery('english', 'tax'),
2   'StartSel = <,
      StopSel = >,
      MinWords=5,
      MaxWords=7,
      MaxFragments=1')
FROM president_speeches
WHERE search_speech_text @@ to_tsquery('english', 'tax')
ORDER BY speech_date;
```

[Listing 14-22](#): Displaying search results with `ts_headline()`

To declare `ts_headline()` 1, we pass the original `speech_text` column rather than the `tsvector` column we used in the search function as the first argument. Then, as the second argument, we pass a `to_tsquery()` function that specifies the word to highlight. We follow this with a third argument that lists optional formatting parameters 2 separated by commas. Here, we specify characters that will identify the start and end of the matched search term or terms (`startSel` and `stopSel`). We also set the minimum and maximum number of total words to display, including the matched terms (`MinWords` and `MaxWords`), plus the maximum number of fragments (or instances of a match) to show using `MaxFragments`. These settings are optional, and you can adjust them according to your needs.

The results of this query should show at most seven words per speech, highlighting words in which `tax` is the root:

president	speech_date	ts_headline
Harry S. Truman bond campaigns	1946-01-21	price controls, increased <taxes>, savings
Harry S. Truman present	1947-01-06	excise <tax> rates which, under the
Harry S. Truman	1948-01-07	increased-after <taxes>-by more than
Harry S. Truman bring	1949-01-05	Congress enact new <tax> legislation to
Harry S. Truman Congress	1950-01-04	considered <tax> reduction of the 80th
Harry S. Truman	1951-01-08	major increase in <taxes> to meet
Harry S. Truman	1952-01-09	This means high <taxes> over the next
Dwight D. Eisenhower	1953-02-02	reduction of the <tax> burden;
Dwight D. Eisenhower	1954-01-07	brought under control. <Taxes> have begun
Dwight D. Eisenhower encouraged increased	1955-01-06	prices and materials. <Tax> revisions
--snip--		

Now, we can quickly see the context of the term we searched. You might also use this function to provide flexible display options for a search feature on a web application. And notice that we didn't just find exact matches. The search engine identified `tax` along with `taxes`, `Tax`, and `Taxes`—words with `tax` as the root and regardless of case.

Let's continue trying forms of searches.

Using Multiple Search Terms

As another example, we could look for speeches in which a president mentioned the word `transportation` but didn't discuss `roads`. We might want to do this to find speeches that focused on broader policy rather than a specific roads program. To do this, we use the syntax in [Listing 14-23](#).

```
SELECT president,
       speech_date,
  1 ts_headline(speech_text,
                to_tsquery('english', 'transportation & !roads'),
                'StartSel = <,
```

```

        StopSel = >,
        MinWords=5,
        MaxWords=7,
        MaxFragments=1')
FROM president_speeches
? WHERE search演讲_text @@ to_tsquery('english', 'transportation & !roads')
ORDER BY speech_date;

```

[Listing 14-23](#): Finding speeches with the word transportation but not roads

Again, we use `ts_headline()` 1 to highlight the terms our search finds. In the `to_tsquery()` function in the `WHERE` clause 2, we pass `transportation` and `roads`, combining them with the ampersand (`&`) operator. We use the exclamation point (!) in front of `roads` to indicate that we want speeches that do not contain this word. This query should find 15 speeches that fit the criteria. Here are the first four rows:

president	speech_date	ts_headline
Harry S. Truman	1947-01-06	such industries as <transportation>, coal, oil, steel
Harry S. Truman	1949-01-05	field of <transportation>.
John F. Kennedy	1961-01-30	Obtaining additional air <transport> mobility--and obtaining
Lyndon B. Johnson	1964-01-08	reformed our tangled <transportation> and transit policies
--snip--		

Notice that the highlighted words in the `ts_headline` column include `transportation` and `transport`. Again, `to_tsquery()` converted `transportation` to the lexeme `transport` for the search term. This database behavior is extremely useful in helping to find relevant related words.

Searching for Adjacent Words

Finally, we'll use the distance (`<->`) operator, which consists of a hyphen between the less-than and greater-than signs, to find adjacent words. Alternatively, you can place a number between the signs to find terms that many words apart. For example, [Listing 14-24](#) searches for any speeches that include the word *military* immediately followed by *defense*.

```

SELECT president,
       speech_date,
       ts_headline(speech_text,
                   to_tsquery('english', 'military <-> defense'),
                   'StartSel = <',
                   StopSel = >,
                   MinWords=5,
                   MaxWords=7,
                   MaxFragments=1')
FROM president_speeches
WHERE search演讲_text @@ to_tsquery('english', 'military <-> defense')

```

```
    to_tsquery('english', 'military <-> defense')
ORDER BY speech_date;
```

Listing 14-24: Finding speeches where defense follows military

This query should find five speeches, and because `to_tsquery()` converts the search terms to lexemes, the words identified in the speeches should include plurals, such as *military defenses*. The following shows the speeches that have the adjacent terms:

president	speech_date	ts_headline
Dwight D. Eisenhower designed	1956-01-05	system our <military> <defenses> are
Dwight D. Eisenhower but likewise	1958-01-09	direct <military> <defense> efforts,
Dwight D. Eisenhower of national life	1959-01-09	survival--the <military> <defense>
Richard M. Nixon <military> <defenses>	1972-01-20	<defense> spending. Strong
Jimmy Carter are strong	1979-01-23	secure. Our <military> <defenses>

If you changed the query terms to `military <2> defense`, the database would return matches where the terms are exactly two words apart, as in the phrase “our military and defense commitments.”

Ranking Query Matches by Relevance

You can also rank search results by relevance using two of PostgreSQL’s full-text search functions. These functions are helpful when you’re trying to understand which piece of text, or speech in this case, is most relevant to your particular search terms.

One function, `ts_rank()`, generates a rank value (returned as a variable-precision `real` data type) based on how often the lexemes you’re searching for appear in the text. The other function, `ts_rank_cd()`, considers how close the lexemes searched are to each other. Both functions can take optional arguments to consider document length and other factors. The rank value they generate is an arbitrary decimal that’s useful for sorting but doesn’t have any inherent meaning. For example, a value of 0.375 generated during one query isn’t directly comparable to the same value generated during a different query.

As an example, *Listing 14-25* uses `ts_rank()` to rank speeches containing all the words *war*, *security*, *threat*, and *enemy*.

```
SELECT president,
       speech_date,
1  ts_rank(search_speech_text,
          to_tsquery('english', 'war & security & threat & enemy'))
     AS score
FROM president_speeches
? WHERE search_speech_text @@ to_tsquery('english', 'war & security & threat & enemy')
```

```
ORDER BY score DESC  
LIMIT 5;
```

[Listing 14-25](#): Scoring relevance with `ts_rank()`

In this query, the `ts_rank()` function 1 takes two arguments: the `search_speech_text` column and the output of a `to_tsquery()` function containing the search terms. The output of the function receives the alias `score`. In the `WHERE` clause 2 we filter the results to only those speeches that contain the search terms specified. Then we order the results in `score` in descending order and return just five of the highest-ranking speeches. The results should be as follows:

president	speech_date	score
William J. Clinton	1997-02-04	0.35810584
George W. Bush	2004-01-20	0.29587495
George W. Bush	2003-01-28	0.28381455
Harry S. Truman	1946-01-21	0.25752166
William J. Clinton	2000-01-27	0.22214262

Bill Clinton's 1997 State of the Union message contains the words *war*, *security*, *threat*, and *enemy* more often than the other speeches, as he discussed the Cold War and other topics. However, it also happens to be one of the longer speeches in the table (which you can determine by using `char_length()`, as you learned earlier in the chapter). The lengths of speeches influences these rankings because `ts_rank()` factors in the number of matching terms in a given text. Two speeches by George W. Bush, delivered in the years before and after the start of the Iraq War, rank next.

It would be ideal to compare frequencies between speeches of identical lengths to get a more accurate ranking, but this isn't always possible. However, we can factor in the length of each speech by adding a normalization code as a third parameter of the `ts_rank()` function, as shown in [Listing 14-26](#).

```
SELECT president,  
       speech_date,  
       ts_rank(search_speech_text,  
              to_tsquery('english', 'war & security & threat & enemy'),  
21)::numeric  
              AS score  
FROM president_speeches  
WHERE search_speech_text @@  
      to_tsquery('english', 'war & security & threat & enemy')  
ORDER BY score DESC  
LIMIT 5;
```

[Listing 14-26](#): Normalizing `ts_rank()` by speech length

Adding the optional code 2 1 instructs the function to divide the `score` by the length of the data in the `search_speech_text` column. This quotient then represents a score normalized by the document length, giving an apples-to-apples comparison among the speeches. The

PostgreSQL documentation at <https://www.postgresql.org/docs/current/textsearch-controls.html> lists all the options available for text search, including using the document length and dividing by the number of unique words.

After running the code in [Listing 14-26](#), the rankings should change:

president	speech_date	score
George W. Bush	2004-01-20	0.0001028060
William J. Clinton	1997-02-04	0.0000982188
George W. Bush	2003-01-28	0.0000957216
Jimmy Carter	1979-01-23	0.0000898701
Lyndon B. Johnson	1968-01-17	0.0000728288

In contrast to the ranking results in [Listing 14-25](#), George W. Bush's 2004 speech now tops the rankings, and Truman's 1946 message falls out of the top five. This might be a more meaningful ranking than the first sample output, because we normalized it by length. But three of the five top-ranked speeches are the same between the two sets, and you can be reasonably certain that each of these three is worthy of closer examination to understand more about presidential speeches that include wartime terminology.

Wrapping Up

Far from being boring, text offers abundant opportunities for data analysis. In this chapter, you've learned techniques for turning ordinary text into data you can extract, quantify, search, and rank. In your work or studies, keep an eye out for routine reports that have facts buried inside chunks of text. You can use regular expressions to dig them out, turn them into structured data, and analyze them to find trends. You can also use search functions to analyze the text.

In the next chapter, you'll learn how PostgreSQL can help you analyze geographic information.

TRY IT YOURSELF

Use your new text-wrangling skills to tackle these tasks:

The style guide of a publishing company you're writing for wants you to avoid commas before suffixes in names. But there are several names like Alvarez, Jr. and Williams, Sr. in your database. Which functions can you use to remove the comma? Would a regular expression function help? How would you capture just the suffixes to place them into a separate column?

Using any one of the presidents' speeches, count the number of unique words that are five characters or more. (Hint: You can use `regexp_split_to_table()` in a subquery to create a table of words to count.) Bonus: Remove commas and periods at the end of each word.

Rewrite the query in [Listing 14-25](#) using the `ts_rank_cd()` function instead of `ts_rank()`. According to the PostgreSQL documentation, `ts_rank_cd()` computes cover density, which takes into account how close the lexeme search terms are to each other. Does using the `ts_rank_cd()` function significantly change the results?

15

ANALYZING SPATIAL DATA WITH POSTGIS



We now turn to *spatial data*, defined as information about the location, shape, and attributes of objects—points, lines, or polygons, for example—within a geographical space. In this chapter, you’ll learn how to construct and query spatial data using SQL, and you’ll be introduced to the PostGIS extension for PostgreSQL that enables support for spatial data types and functions.

Spatial data has become a critical piece of our world’s data ecosystem. A phone app can find nearby coffee shops because it queries a spatial database, asking it to return a list of shops within a certain distance of your location. Governments use spatial data to track the footprints of residential and business parcels; epidemiologists use it to visualize the spread of diseases.

For our exercises, we’ll analyze the location of farmers’ markets across the United States as well as roads and waterways in Santa Fe, New Mexico. You’ll learn how to construct and query spatial data types and incorporate map projections and grid systems. You’ll receive tools to glean information from spatial data, similar to how you’ve analyzed numbers and text.

We’ll start by setting up PostGIS. All code and data for the exercises are available with the book’s resources at <https://nostarch.com/practical-sql-2nd-edition/>.

Enabling PostGIS and Creating a Spatial Database

PostGIS is a free, open source project created by the Canadian geospatial company Refractions Research and maintained by an international team of developers under the Open Source Geospatial Foundation (OSGeo). The GIS portion of its name refers to *geographic information system*, defined as a system that allows for storing, editing, analyzing, and displaying spatial data. You'll find documentation and updates at <https://postgis.net/>.

If you installed PostgreSQL following the steps for Windows, macOS, or the Ubuntu flavor of Linux in Chapter 1, PostGIS should be on your machine. If you installed PostgreSQL some other way on Windows or macOS or if you're on another Linux distribution, follow the installation instructions at <https://postgis.net/install/>.

To enable PostGIS on your analysis database, open pgAdmin's Query Tool and run the statement in [Listing 15-1](#).

```
CREATE EXTENSION postgis;
```

[Listing 15-1](#): Loading the PostGIS extension

You'll see the message `CREATE EXTENSION`, advising that your database has been updated to include spatial data types and analysis functions. Run `SELECT postgis_full_version()`; to display the version number of PostGIS along with the versions of its installed components. The version won't match your installed PostgreSQL version, and that's okay.

Understanding the Building Blocks of Spatial Data

Before you learn to query spatial data, let's look at how it's described in GIS and related data formats. This is important background, but if you want to dive straight into queries, you can skip to "Understanding PostGIS Data Types" later in the chapter and return here afterward.

A point on a grid is the smallest building block of spatial data. The grid might be marked with x- and y-axes, or longitude and latitude if we're using a map. A

grid could be flat with two dimensions, or it could describe a three-dimensional space such as a cube. In some data formats, such as the JavaScript-based *GeoJSON*, a point may have attributes in addition to its location. We could describe a grocery store with a point containing its longitude and latitude as well as attributes for the store's name and hours of operation.

Understanding Two-Dimensional Geometries

The Open Geospatial Consortium (OGC) and International Organization for Standardization (ISO) have created a *simple features access* model that describes standards for building and querying two- and three-dimensional shapes, sometimes referred to as *geometries*. PostGIS supports the standard.

The following are the more common features, starting with points and building in complexity:

Point

A single location in a two- or three-dimensional plane. On maps, a Point is usually a dot marking a longitude and latitude.

LineString

Two or more Points, each connected by straight lines. A LineString can represent features such as a road, biking trail, or stream.

Polygon

A two-dimensional shape with three or more straight sides, each constructed from a LineString. On maps, Polygons represent objects such as nations, states, buildings, and bodies of water. A Polygon can have one or more interior Polygons that act as holes inside the larger Polygon.

MultiPoint

A set of Points. A single MultiPoint object could represent multiple locations of a retailer with each store's latitude and longitude.

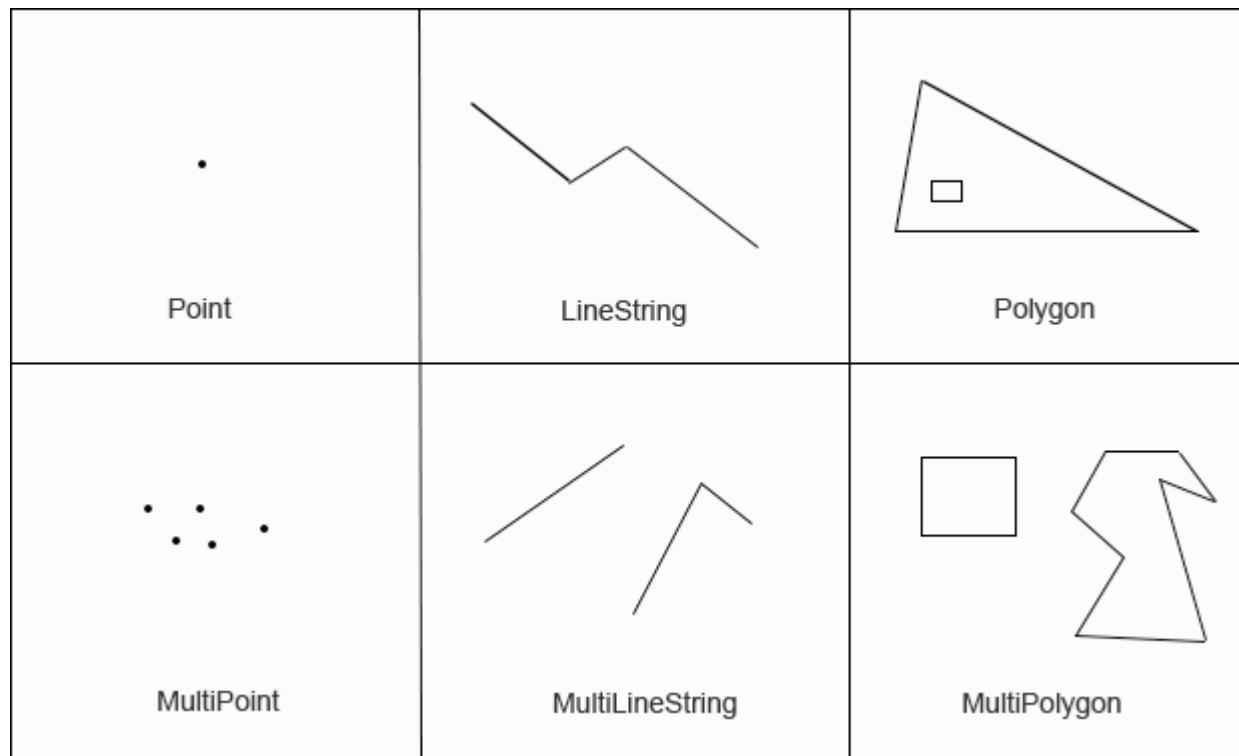
MultiLineString

A set of LineStrings. An example is a road that has several noncontinuous segments.

MultiPolygon

A set of Polygons. A parcel of land that's divided into parts by a road could be grouped in one MultiPolygon object instead of separate polygons.

[Figure 15-1](#) shows an example of each feature. PostGIS enables functions to build, edit, and analyze these objects. These functions take a variety of inputs depending on their purpose, including latitude and longitude, specialized text and binary formats, and simple features. Some functions also take an optional *spatial reference system identifier (SRID)* that specifies the grid on which to place the objects.



[Figure 15-1: Visual examples of geometries](#)

I'll explain the SRID shortly, but first, let's look at examples of an input used by PostGIS functions called *well-known text (WKT)*—a text-based format that represents a geometry.

Well-Known Text Formats

The OGC standard's WKT format specifies a geometry type and its coordinates inside one or more sets of parentheses. The number of coordinates and parentheses varies depending on the type of geometry. [Table 15-1](#) shows examples of frequently used geometry types and their WKT formats. Longitude/latitude pairs are shown for the coordinates, but you might encounter grid systems that use other measures.

NOTE

WKT accepts coordinates in the order of (longitude, latitude), which is backward from Google Maps and some other software. Tom MacWright, formerly of the Mapbox software company, notes at <https://macwright.com/lonlat/> that neither order is “right” and catalogs the “frustrating inconsistency” in which mapping-related code handles the order of coordinates.

Table 15-1: Well-Known Text Formats for Geometries

G eo m etr y	Format	Notes
Point	POINT (-74.9 42.7)	A coordinate pair marking a point at -74.9 longitude and 42.7 latitude.
LineString	LINESTRING (-74.9 42.7, -75.1 42.7)	A straight line with endpoints marked by two coordinate pairs.
Polygon	POLYGON ((-74.9 42.7, -75.1 42.7, -75.1 42.6, -74.9 42.7))	A triangle outlined by three different pairs of coordinates. Although listed twice, the first and last pair are the same coordinates where we close the shape.
Multipoint	MULTIPOINT (-74.9 42.7, -75.1 42.7)	Two Points, one for each pair of coordinates.
MultilineString	MULTILINESTRING ((-76.27 43.1, -76.06 43.08), (-76.2 43.3, -76.2 43.4, -76.4 43.1))	Two LineStrings. The first has two points; the second has three.
Multipolygon	MULTIPOLYGON ((((-74.92 42.7, -75.06 42.71, -75.07 42.64, -74.92 42.7), (-75.0 42.66, -75.0 42.64, -74.98 42.64, -74.98 42.66, -75.0 42.66))))	Two Polygons. The first is a triangle, and the second is a rectangle.

These examples create simple shapes, as you'll see when we construct them using PostGIS later in the chapter. In practice, complex geometries will

comprise thousands of coordinates.

Projections and Coordinate Systems

Representing Earth's spherical surface on a two-dimensional map is not easy. Imagine peeling the outer layer of Earth from the globe and trying to spread it on a table while keeping all pieces of the continents and oceans connected. Inevitably, you'd have to stretch some parts of the map. That's what happens when cartographers create a map *projection* with its own *projected coordinate system*. A projection is simply a flattened representation of the globe with its own two-dimensional coordinate system.

Some projections represent the entire world; others are specific to regions or purposes. The *Mercator projection* has properties useful for navigation; Google Maps and other online maps use a variant of called *Web Mercator*. The math behind its transformation distorts land areas close to the North and South Poles, making them appear much larger than reality. The US Census Bureau uses the *Albers projection*, which minimizes distortion and is the one you see on TV in the United States as votes are tallied on election night.

Projections are derived from *geographic coordinate systems*, which define the grid of latitude, longitude, and height of any point on the globe along with factors including Earth's shape. Whenever you obtain geographic data, it's critical to know the coordinate systems it references so you provide the correct information when writing queries. Often, user documentation will name the coordinate system. Next, let's look at how to specify the coordinate system in PostGIS.

Spatial Reference System Identifier

When using PostGIS (and many GIS applications), you specify the coordinate system via its unique SRID. When you enabled the PostGIS extension at the beginning of this chapter, the process created the table `spatial_ref_sys`, which contains SRIDs as its primary key. The table also contains the column `srtext`, which includes a WKT representation of the spatial reference system plus other metadata.

In this chapter, we'll frequently use SRID 4326, the ID for the geographic coordinate system WGS 84. That's the most recent World Geodetic System (WGS) standard used by GPS, and you'll encounter it often in spatial data. You

can see the WKT representation for WGS 84 by running the code in [Listing 15-2](#) that looks for its SRID, 4326:

```
SELECT srtext
FROM spatial_ref_sys
WHERE srid = 4326;
```

[Listing 15-2](#): Retrieving the WKT for SRID 4326

Run the query and you should get the following result, indented for readability:

```
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84", 6378137, 298.257223563,
            AUTHORITY["EPSG", "7030"]], ,
        AUTHORITY["EPSG", "6326"]], ,
    PRIMEM["Greenwich", 0,
        AUTHORITY["EPSG", "8901"]], ,
    UNIT["degree", 0.0174532925199433,
        AUTHORITY["EPSG", "9122"]], ,
    AUTHORITY["EPSG", "4326"]]
```

You don't need to use this information for any of this chapter's exercises, but it's helpful to know some of the variables and how they define the projection. The `GEOGCS` keyword provides the geographic coordinate system in use. Keyword `PRIMEM` specifies the location of the *prime meridian*, or longitude 0. To see definitions of all the variables, check the reference at <https://docs.geotools.org/stable/javadoc/org/opengis/referencing/doc-files/WKT.html>.

Conversely, if you ever need to find the SRID associated with a coordinate system, you can query the `srtext` column in `spatial_ref_sys` to find it.

Understanding PostGIS Data Types

Installing PostGIS adds several data types to your database. We'll use two: `geography` and `geometry`. Both types can store spatial data, such as the points, lines, polygons, and SRIDs you just learned about, but they have important distinctions:

geography A data type based on a sphere, using the round-Earth coordinate system (longitude and latitude). All calculations occur on the globe, taking its curvature into account. This makes the math complex and limits the number of

functions available to work with the geography type. But because Earth's curvature is factored in, calculations for distance are more precise; you should use the geography data type when handling data that spans large areas. The results from calculations on the geography type will be expressed in meters.

geometry A data type based on a plane, using the Euclidean coordinate system. Calculations occur on straight lines as opposed to along the curvature of a sphere, making calculations for geographical distance less precise than with the geography data type; the results of calculations are expressed in units of whichever coordinate system you've designated.

The PostGIS documentation at

https://postgis.net/docs/using_postgis_dbmanagement.html offers guidance on when to use one or the other type. In short, if you're working strictly with longitude/latitude data or if your data covers a large area, such as a continent or the globe, use the geography type, even though it limits the functions you can use. If your data covers a smaller area, the geometry type provides more functions and better performance. You can also convert one type to the other using CAST.

With the background you have now, we can start working with spatial objects.

Creating Spatial Objects with PostGIS Functions

PostGIS has more than three dozen constructor functions that build spatial objects using WKT or coordinates. You can find a list at https://postgis.net/docs/reference.html#Geometry_Constructors, but the following sections explain several that you'll use in the exercises. Most PostGIS functions begin with the letters *ST*, which is an ISO naming standard that means *spatial type*.

Creating a Geometry Type from Well-Known Text

The `ST_GeomFromText(WKT, SRID)` function creates a geometry data type from an input of a WKT string and an optional SRID. [Listing 15-3](#) shows simple SELECT statements that generate geometry data types for each of the simple features described in [Table 15-1](#).

```
SELECT ST_GeomFromText('POINT(-74.9233606 42.699992)',  
24326);
```

```

SELECT ST_GeomFromText('LINESTRING(-74.9 42.7, -75.1 42.7)', 4326);

SELECT ST_GeomFromText('POLYGON((-74.9 42.7, -75.1 42.7,
                               -75.1 42.6, -74.9 42.7))', 4326);

SELECT ST_GeomFromText('MULTIPOINT (-74.9 42.7, -75.1 42.7)', 4326);

SELECT ST_GeomFromText('MULTILINESTRING((-76.27 43.1, -76.06
43.08),
                               (-76.2 43.3, -76.2
43.4,
                               -76.4 43.1))', 4326);

SELECT ST_GeomFromText('MULTIPOLYGON3((
                               (-74.92 42.7, -75.06
42.71,
                               -75.07 42.64, -74.92
42.7)4,
                               (-75.0 42.66, -75.0
42.64,
                               -74.98 42.64, -74.98
42.66,
                               -75.0 42.66)))', 4326);

```

[Listing 15-3](#): Using `ST_GeomFromText()` to create spatial objects

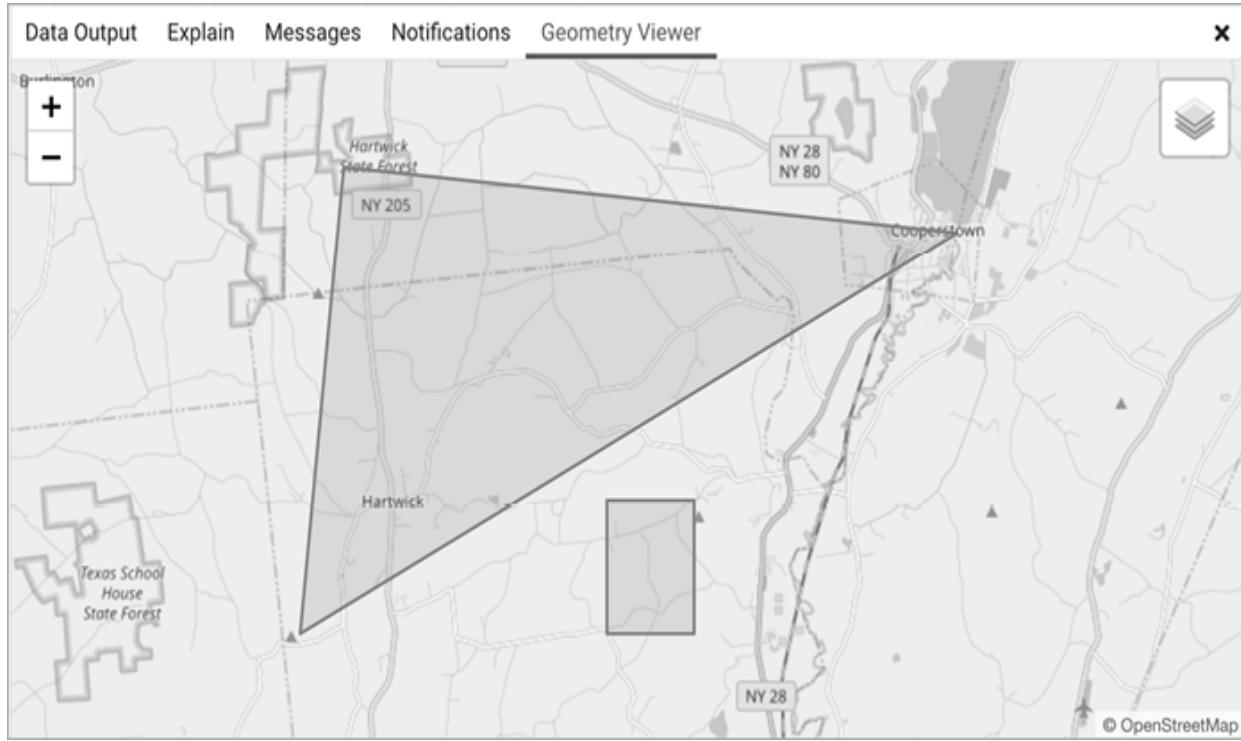
For each example, we give a WKT string as the first input and the SRID 4326 as the second. In the first example, we create a Point by inserting the WKT POINT string **1** as the first argument to `ST_GeomFromText()` with the SRID **2** as the optional second argument. We use the same format in the rest of the examples. Note that we don't have to indent the coordinates. I do so here only to make the coordinate pairs more readable.

Be sure to mind the number of parentheses that segregate objects, particularly in complex structures such as the MultiPolygon. For example, we need to use two opening parentheses **3** and enclose each polygon's coordinates within another set of parentheses **4**.

If you run each statement separately in pgAdmin, you can view both its data output and visual representation. Upon execution, each statement should return a single column of the `geometry` data type displayed as a string of characters that looks something like this truncated example:

```
0101000020E61000008EDA0E5718BB52C017BB7D5699594540 ...
```

The string is of the format *extended well-known binary (EWKB)*, which you typically won't need to interpret directly. Instead, you'll use columns of geometry (or geography) data as inputs to other functions. To see the visual representation, click the eye icon in the pgAdmin result column header. That should open a Geometry Viewer pane in pgAdmin that displays the geometry atop a map that uses OpenStreetMap as the base layer. For example, the `MULTIPOLYGON` example in [Listing 15-3](#) should look like [Figure 15-2](#), with a triangle and a rectangle.



[Figure 15-2: Viewing geometries in pgAdmin](#)

Try viewing each example in [Listing 15-3](#) to get to know the differences between objects.

Creating a Geography Type from Well-Known Text

To create a geography data type, you can use `ST_GeogFromText (WKT)` to convert a WKT or `ST_GeogFromText (EWKT)` to convert a PostGIS-specific variation called *extended WKT* that includes the SRID. [Listing 15-4](#) shows how to pass in the SRID as part of the extended WKT string to create a MultiPoint geography object with three points.

```
SELECT
ST_GeogFromText('SRID=4326;MULTIPOINT (-74.9 42.7, -75.1 42.7,
-74.924 42.6)')
```

[Listing 15-4](#): Using `ST_GeogFromText ()` to create spatial objects

Again, you can view the Points on a map by clicking the eye icon in the geography column in the pgAdmin results grid.

Along with the all-purpose `ST_GeomFromText ()` and `ST_GeogFromText ()` functions, PostGIS includes several that are specific to creating certain spatial objects. I'll cover those briefly next.

Using Point Functions

The `ST_PointFromText ()` and `ST_MakePoint ()` functions will turn a WKT POINT or a collection of coordinates, respectively, into a geometry data type. Points mark coordinates, such as longitude and latitude, which you would use to identify locations or use as building blocks of other objects, such as LineStrings.

[Listing 15-5](#) shows how these functions work.

```
SELECT 1ST_PointFromText('POINT (-74.9233606 42.699992)', 4326);

SELECT 2ST_MakePoint(-74.9233606, 42.699992);
SELECT 3ST_SetSRID(ST_MakePoint(-74.9233606, 42.699992), 4326);
```

[Listing 15-5](#): Functions specific to making Points

The `ST_PointFromText (WKT, SRID)` 1 function creates a point geometry type from a WKT POINT and an optional SRID as the second input. The

PostGIS docs note that the function includes validation of coordinates that makes it slower than the `ST_GeomFromText()` function.

The `ST_MakePoint(x, y, z, m)` **2** function creates a point geometry type on a two-, three-, and four-dimensional grid. The first two parameters, `x` and `y` in the example, represent longitude and latitude coordinates. You can use the optional `z` to represent altitude and `m` to represent a measure. That would allow you, for example, to mark a water fountain on a bike trail at a certain altitude and certain distance from the start of the trail. The `ST_MakePoint()` function is faster than `ST_GeomFromText()` and `ST_PointFromText()`, but if you want to specify an SRID, you'll need to designate one by wrapping it inside the `ST_SetSRID()` **3** function.

Using LineString Functions

Now let's examine some functions we use specifically for creating LineString geometry data types. [Listing 15-6](#) shows how they work.

```
SELECT 1ST_LineFromText('LINESTRING (-105.90 35.67, -105.91  
35.67)', 4326);  
SELECT 2ST_MakeLine(ST_MakePoint(-74.9, 42.7),  
ST_MakePoint(-74.1, 42.4));
```

[Listing 15-6](#): Functions specific to making LineStrings

The `ST_LineFromText(WKT, SRID)` **1** function creates a LineString from a WKT `LINESTRING` and an optional SRID as its second input. Like `ST_PointFromText()` earlier, this function includes validation of coordinates that makes it slower than `ST_GeomFromText()`.

The `ST_MakeLine(geom, geom)` **2** function creates a LineString from inputs that must be of the `geometry` data type. In [Listing 15-6](#), the example uses two `ST_MakePoint()` functions as inputs to create the start and endpoint of the line. You can also pass in an `ARRAY` object with multiple points, perhaps generated by a subquery, to generate a more complex line.

Using Polygon Functions

Let's look at three Polygon functions: `ST_PolygonFromText()`, `ST_MakePolygon()`, and `ST_MPolyFromText()`. All create geometry data

types. [Listing 15-7](#) shows how you can create Polygons with each.

```
SELECT 1ST_PolygonFromText('POLYGON((-74.9 42.7, -75.1 42.7,
                                         -75.1 42.6, -74.9
                                         42.7))', 4326);

SELECT 2ST_MakePolygon(
    ST_GeomFromText('LINESTRING(-74.92 42.7, -75.06
                      42.71,
                                         -75.07 42.64, -74.92
                                         42.7)', 4326));

SELECT 3ST_MPolyFromText('MULTIPOLYGON((
                                         (-74.92 42.7, -75.06
                                         42.71,
                                         -75.07 42.64, -74.92
                                         42.7),
                                         (-75.0 42.66, -75.0
                                         42.64,
                                         -74.98 42.64, -74.98
                                         42.66,
                                         -75.0 42.66)
                                         ) )', 4326);
```

[Listing 15-7](#): Functions specific to making Polygons

The `ST_PolygonFromText(WKT, SRID)` **1** function creates a Polygon from a WKT `POLYGON` and an optional SRID. As with the similarly named functions for creating points and lines, it includes a validation step that makes it slower than `ST_GeomFromText()`.

The `ST_MakePolygon(linestring)` **2** function creates a Polygon from a LineString that must open and close with the same coordinates, ensuring the object is closed. This example uses `ST_GeomFromText()` to create the LineString geometry using a WKT `LINESTRING`.

The `ST_MPolyFromText(WKT, SRID)` **3** function creates a MultiPolygon from a WKT and an optional SRID.

Now you have the building blocks to analyze spatial data. Next, we'll use them to explore a set of data.

Analyzing Farmers' Markets Data

The National Farmers' Market Directory from the US Department of Agriculture catalogs the location and offerings of more than 8,600 “markets that feature two or more farm vendors selling agricultural products directly to customers at a common, recurrent physical location,” according to the update page linked from the main directory site at <https://www.ams.usda.gov/local-food-directories/farmersmarkets/>. Attending these markets is a fun weekend activity, so let’s use SQL spatial queries to find the closest markets.

The *farmers_markets.csv* file contains a portion of the USDA data on each market, and it’s available along with the book’s resources at <https://nostarch.com/practical-sql-2nd-edition/>. Save the file to your computer and run the code in [Listing 15-8](#) to create and load a *farmers_markets* table.

```
CREATE TABLE farmers_markets (
    fmid bigint PRIMARY KEY,
    market_name text NOT NULL,
    street text,
    city text,
    county text,
    st text NOT NULL,
    zip text,
    longitude numeric(10, 7),
    latitude numeric(10, 7),
    organic text NOT NULL
);

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

[Listing 15-8](#): Creating and loading the *farmers_markets* table

The table contains routine address data plus the `longitude` and `latitude` for most markets. Twenty-nine of the markets were missing those values when I downloaded the file from the USDA. An `organic` column indicates whether the market offers organic products; a hyphen (-) in that column indicates an unknown value. After you import the data, count the rows using `SELECT count(*) FROM farmers_markets;`. If everything imported correctly, you should have 8,681 rows.

Creating and Filling a Geography Column

To perform spatial queries on the markets' longitude and latitude, we need to convert those coordinates into a single column with a spatial data type. Because we're working with locations spanning the entire United States and an accurate measurement of a large spherical distance is important, we'll use the geography type. After creating the column, we can update it using Points derived from the coordinates and then apply an index to speed up queries. [Listing 15-9](#) contains the statements for doing these tasks.

```
ALTER TABLE farmers_markets ADD COLUMN geog_point
geography(POINT, 4326); 1

UPDATE farmers_markets
SET geog_point =
  2 ST_SetSRID(
    3
    ST_MakePoint(longitude,latitude)4::geography, 4326
  );
  4

CREATE INDEX market_pts_idx ON farmers_markets USING GIST
(geog_point); 5

SELECT longitude,
       latitude,
       geog_point,
  6 ST_AsEWKT(geog_point)
FROM farmers_markets
WHERE longitude IS NOT NULL
LIMIT 5;
```

[Listing 15-9:](#) *Creating and indexing a geography column*

The `ALTER TABLE` statement 1 you learned in Chapter 10 with the `ADD COLUMN` option creates a column of the `geography` type called `geog_point` that will hold points and reference the WGS 84 coordinate system, which we denote using SRID 4326.

Next, we run a standard `UPDATE` statement to fill the `geog_point` column. Nested inside an `ST_SetSRID()` 2 function, the `ST_MakePoint()` 3 function takes as input the `longitude` and `latitude` columns from the table. The

output, which is the `geometry` type by default, must be cast to `geography` to match the `geog_point` column type. To do this, we add the PostgreSQL-specific double-colon syntax (`::`) **4** to the output of `ST_MakePoint()`.

Adding a Spatial Index

Before you start analysis, it's wise to add an index to the new column to speed up queries. In Chapter 8, you learned about PostgreSQL's default index, the B-tree. A B-tree index is useful for data that you can order and search using equality and range operators, but it's less useful for spatial objects. The reason is that you cannot easily sort GIS data along one axis. For example, the application has no way to determine which of these coordinate pairs is greatest: (0,0), (0,1), or (1,0).

Instead, the makers of PostGIS include support for an index designed for spatial data called *R-tree*. In an R-tree index, each spatial item is represented in the index as a rectangle that surrounds its boundaries, and the index itself is a hierarchy of rectangles. (Find a good overview at <https://postgis.net/workshops/postgis-intro/indexing.html>.)

We add a spatial index to the `geog_point` column by including the keywords `USING GIST` in the `CREATE INDEX` statement **5** in [Listing 15-9](#). GIST refers to a generalized search tree (GiST), an interface to facilitate incorporating specialized indexes to the database. PostgreSQL core team member Bruce Momjian describes GiST as “a general indexing framework designed to allow indexing of complex data types.”

With the index in place, we use the `SELECT` statement to view the geography data to show the newly encoded `geog_points` column. To view the extended WKT version of `geog_point`, we wrap it in a `ST_AsEWKT()` function **6** to show the extended well-known text coordinates and SRID. The results should look similar to this, with `geog_point` truncated for brevity:

longitude	latitude	geog_point	st_asewkt
-105.5890000	47.4154000	01010000...	SRID=4326;POINT (-105.589 47.4154)
-98.9530000	40.4998000	01010000...	SRID=4326;POINT (-98.953 40.4998)
-119.4280000	35.7610000	01010000...	SRID=4326;POINT (-119.428 35.761)

```
-92.3063000 42.1718000 01010000... SRID=4326;POINT (-92.3063  
42.1718)  
-70.6868160 44.1129600 01010000...  
SRID=4326;POINT (-70.686816 44.11296) )
```

Now we're ready to perform calculations on the points.

Finding Geographies Within a Given Distance

Several years ago, while reporting a story on farming in Iowa, I visited the massive Downtown Farmers' Market in Des Moines. With hundreds of vendors, the market spanned several city blocks in the Iowa capital. Farming is big business there, and even though the downtown market is huge, it's not the only one in the area. Let's use PostGIS to find more farmers' markets near downtown Des Moines.

The PostGIS function `ST_DWithin()` returns a Boolean value of `true` if one spatial object is within a specified distance of another object. If you're working with the `geography` data type, as we are here, you need to use meters as the distance unit. If you're using the `geometry` type, use the distance unit specified by the SRID.

NOTE

PostGIS distance measurements are on a straight line for geometry data, and on a sphere for geography data. Be careful not to confuse either with driving distance along roads, which is usually farther from point to point. To perform calculations related to driving distances, check out the extension pgRouting at <https://pgRouting.org/>.

[Listing 15-10](#) uses the `ST_DWithin()` function to filter `farmers_markets` to show markets within 10 kilometers of the Downtown Farmers' Market in Des Moines.

```
SELECT market_name,  
       city,  
       st,  
       geog_point  
FROM farmers_markets  
WHERE ST_DWithin(1 geog_point,  
                2 ST_GeogFromText('POINT(-93.6204386
```

```
41.5853202)' ,  
    3 10000)  
ORDER BY market_name;
```

[Listing 15-10](#): Using ST_DWithin() to locate farmers' markets within 10 km of a point

The first input for `ST_DWithin()` is `geog_point 1`, which holds the location of each row's market in the `geography` data type. The second input is the `ST_GeogFromText()` function **2** that returns a Point geography from WKT. The coordinates `-93.6204386` and `41.5853202` represent the longitude and latitude of the Downtown Farmers' Market. The final input is `10000 3`, which is the number of meters in 10 kilometers. The database calculates the distance between each market in the table and the downtown market. If a market is within 10 kilometers, it is included in the results.

We're using Points here, but this function works with any `geography` or `geometry` type. If you're working with objects such as polygons, you can use the related `ST_DFullyWithin()` function to find objects that are completely within a specified distance.

Run the query; it should return nine rows (I've omitted the `geog_point` column for brevity):

market_name	city
st	
-----	-----
Beaverdale Farmers Market	Des Moines
Iowa	
Capitol Hill Farmers Market	Des Moines
Iowa	
Downtown Farmers' Market - Des Moines	Des Moines
Iowa	
Drake Neighborhood Farmers Market	Des Moines
Iowa	
Eastside Farmers Market	Des Moines
Iowa	
Highland Park Farmers Market	Des Moines
Iowa	
Historic Valley Junction Farmers Market	West Des Moines
Iowa	
LSI Global Greens Farmers' Market	Des Moines

Iowa

Valley Junction Farmers Market

Iowa

West Des Moines

One of these nine markets is the Downtown Farmers' Market in Des Moines, which makes sense because its location is at the point used for comparison. The rest are other markets in Des Moines or in nearby West Des Moines.

To see these points on a map, in pgAdmin's results grid, click the eye icon in the `geog_point` column header. The geography viewer should display a map as shown in [Figure 15-3](#).



[Figure 15-3: Farmers' markets near downtown Des Moines, Iowa](#)

This operation should be familiar: it's a standard feature on many online maps and product apps that let you locate stores or points of interest near you.

Although this list of nearby markets is helpful, it would be even better to know the exact distance of markets from downtown. We'll use another function to report that.

Finding the Distance Between Geographies

The `ST_Distance()` function returns the minimum distance between two geometries, providing meters for geographies and SRID units for geometries. For example, [Listing 15-11](#) finds the distance in miles from Yankee Stadium in New York City's Bronx borough to Citi Field in Queens, home of the New York Mets.

```
SELECT ST_Distance(
    ST_GeogFromText('POINT(-73.9283685
40.8296466)'),
    ST_GeogFromText('POINT(-73.8480153
40.7570917)')
) / 1609.344 AS mets_to_yanks;
```

[Listing 15-11](#): Using `ST_Distance()` to calculate the miles between Yankee Stadium and Citi Field

To convert the distance units from meters to miles, we divide the result of `ST_Distance()` by 1609.344 (the number of meters in a mile). The result is about 6.5 miles.

```
mets_to_yanks
-----
6.543861827875209
```

Let's apply this technique to the farmers' market data using the code in [Listing 15-12](#). We'll again find all farmers' markets within 10 kilometers of the Downtown Farmers' Market in Des Moines and show the distance in miles.

```
SELECT market_name,
       city,
       1 round(
           (ST_Distance(geog_point,
                        ST_GeogFromText('POINT(-93.6204386
41.5853202)'))
            ) / 1609.344) 2::numeric, 2
       ) AS miles_from_dt
FROM farmers_markets
WHERE 3 ST_DWithin(geog_point,
                    ST_GeogFromText('POINT(-93.6204386
41.5853202)'), 10000)
ORDER BY miles_from_dt ASC;
```

[***Listing 15-12***](#): Using `ST_Distance()` for each row in `farmers_markets`

The query is similar to [***Listing 15-10***](#), which used `ST_DWithin()` to find markets 10 kilometers or closer to downtown, but adds the `ST_Distance()` function as a column to calculate and display the distance from downtown. I've wrapped the function inside `round(1)` to trim the output.

We provide `ST_Distance()` with the same two inputs we gave `ST_DWithin()` in [***Listing 15-10***](#): `geog_point` and the `ST_GeogFromText()` function. The `ST_Distance()` function then calculates the distance between the points specified by both inputs, returning the result in meters. To convert to miles, we divide by 1609.344 **2**, the approximate number of meters in a mile. Then, to provide the `round()` function with the correct input data type, we cast the column result to type `numeric`.

The WHERE clause **3** uses the same `ST_DWithin()` function and inputs as in [***Listing 15-10***](#). You should see the following results, ordered by distance in ascending order:

market_name	city
miles_from_dt	
Downtown Farmers' Market - Des Moines	Des Moines
0.00	
Capitol Hill Farmers Market	Des Moines
1.15	
Drake Neighborhood Farmers Market	Des Moines
1.70	
LSI Global Greens Farmers' Market	Des Moines
2.30	
Highland Park Farmers Market	Des Moines
2.93	
Eastside Farmers Market	Des Moines
3.40	
Beaverdale Farmers Market	Des Moines
3.74	
Historic Valley Junction Farmers Market	West Des Moines
4.68	
Valley Junction Farmers Market	West Des Moines
4.70	

Again, you see this type of result often when you're searching online for a store or address. You might also find the technique helpful for other analysis

scenarios, such as finding all the schools within a certain distance of a known source of pollution or all the homes within five miles of an airport.

Finding the Nearest Geographies

Sometimes it's helpful to have the database simply return the spatial objects that are in closest proximity to another object without specifying some arbitrary distance in which to search. For example, we may want to find the closest farmers' market regardless of whether it's 10 kilometers away or 100. To do that, we can instruct PostGIS to implement a *K-nearest neighbors* search algorithm by using the `<->` distance operator in the `ORDER BY` clause of a query. Nearest neighbors algorithms solve a range of classification problems by identifying similar items—text recognition is an example. In this case, PostGIS will identify some number of spatial objects, represented by k , nearest to an object we specify.

For example, let's say we're planning to visit the vacation spot of Bar Harbor, Maine, and want to find the three farmer's markets closest to town. We can use the code in [Listing 15-13](#).

```
SELECT market_name,
       city,
       st,
       round(
           (ST_Distance(geog_point,
                         ST_GeogFromText('POINT(-68.2041607
44.3876414)'))
            / 1609.344)::numeric, 2
        ) AS miles_from_bh
  FROM farmers_markets
 ORDER BY geog_point <->1 ST_GeogFromText('POINT(-68.2041607
44.3876414)')
 LIMIT 3;
```

[Listing 15-13](#): Using the `<->` distance operator for a nearest neighbors search

The query is similar to [Listing 15-12](#), but instead of using a `WHERE` clause with `ST_DWithin()`, we provide an `ORDER BY` clause that contains the `<-> 1` distance operator. To the left of the operator, we place the `geog_point` column; to the right we supply the WKT for the Point locating downtown Bar Harbor inside `ST_GeogFromText()`. In effect, this syntax says, “Order the results by the distance from the geography to the Point.”

Adding `LIMIT 3` restricts the results to the three closest markets (the three nearest neighbors):

market_name	city	st
miles_from_bh		
Bar Harbor Eden Farmers' Market	Bar Harbor	Maine
0.32		
Northeast Harbor Farmers' Market	Northeast Harbor	Maine
7.65		
Southwest Harbor Farmers' Market	Southwest Harbor	Maine
9.56		

You can, of course, change the number in the `LIMIT` clause to return more or fewer results. Using `LIMIT 1`, for example, will return only the closest market.

So far, you've learned how to work with spatial objects constructed from WKT. Next, I'll show you a common data format used in GIS called the *shapefile* and how to bring it into PostGIS for analysis.

Working with Census Shapefiles

A *shapefile* is a GIS data file format developed by Esri, a US company known for its ArcGIS mapping visualization and analysis platform. Shapefiles are a standard file format for GIS platforms—such as ArcGIS and the open source QGIS—and are used by governments, corporations, nonprofits, and technical organizations to display, analyze, and distribute data with geographic features.

Shapefiles hold information describing the shape of a feature (such as a county, a road, or a lake) plus a database with each feature's attributes. Those attributes might include their name and other demographic descriptors. A single shapefile can contain only one type of shape, such as polygons or points, and when you load a shapefile into a GIS platform that supports visualization, you can view the shapes and query their attributes. PostgreSQL, with the PostGIS extension, lets you query the spatial data in the shapefile, which we'll do in “Exploring the Census 2019 Counties Shapefile” and “Performing Spatial Joins” later in the chapter.

First, let's examine the structure and contents of shapefiles.

Understanding the Contents of a Shapefile

A shapefile comprises a collection of files with different extensions, each with a different purpose. Often, when you download a shapefile, it comes in a compressed archive, such as `.zip`. You'll need to unzip it to access the individual files.

Per ArcGIS documentation, these are the most common extensions you'll encounter:

- .shp** Main file that stores the feature geometry.
- .shx** Index file that stores the index of the feature geometry.
- .dbf** Database table (in dBASE format) that stores the attribute information of features.
- .xml** XML-format file that stores metadata about the shapefile.
- .prj** Projection file that stores the coordinate system information. You can open this file with a text editor to view the geographic coordinate system and projection.

According to the documentation, files with the first three extensions include necessary data required for working with a shapefile. The other file types are optional. You can load a shapefile into PostGIS to access its spatial objects and the attributes for each. Let's do that next and explore some additional analysis functions.

I've included several shapefiles with the resources for this chapter at <https://nostarch.com/practical-sql-2nd-edition/>. We'll start with TIGER/Line Shapefiles from the US Census that contain the boundaries for each county or county equivalent, such as parish or borough, as of 2019. You can learn more about this series of shapefiles at <https://www.census.gov/geographies/mapping-files/time-series/geo/tiger-line-file.html>.

NOTE

Many organizations provide data in shapefile format. Start with your national or local government agencies or check the Wikipedia entry ‘List of GIS data sources.’

Save *tl_2019_us_county.zip* from the book’s resources for this chapter to your computer and unzip it; the archive should contain files including those with the extensions I listed earlier.

Loading Shapefiles

If you’re using Windows, the PostGIS suite includes a Shapefile Import/Export Manager with a simple *graphical user interface (GUI)*. In recent years, builds of that GUI have become harder to find on macOS and Linux distributions, so for those operating systems we’ll instead use the command line application `shp2pgsql`.

We’ll start with the Windows GUI. If you’re on macOS or Linux, skip ahead to “Importing Shapefiles Using `shp2pgsql`.”

Windows Shapefile Importer/Exporter

On Windows, if you followed the installation steps in Chapter 1, you should find the Shapefile Import/Export Manager by selecting **Start▶PostGIS Bundle *x.y* for PostgreSQL x64 *x.y*▶PostGIS Bundle *x.y* for PostgreSQL x64 *x.y* Shapefile and DBF Loader Exporter**.

Whatever you see in place of *x.y* should match your PostgreSQL and PostGIS versions. Click to launch the application.

To establish a connection between the app and your `analysis` database, follow these steps:

Click **View connection details**.

In the dialog that opens, enter **postgres** for the username, and enter a password if you added one for the server during initial setup.

Ensure that Server Host has `localhost` and `5432` by default. Leave those as is unless you’re connecting to a different server or port.

Enter **analysis** for the database name. [*Figure 15-4*](#) shows a screenshot of what the connection should look like.

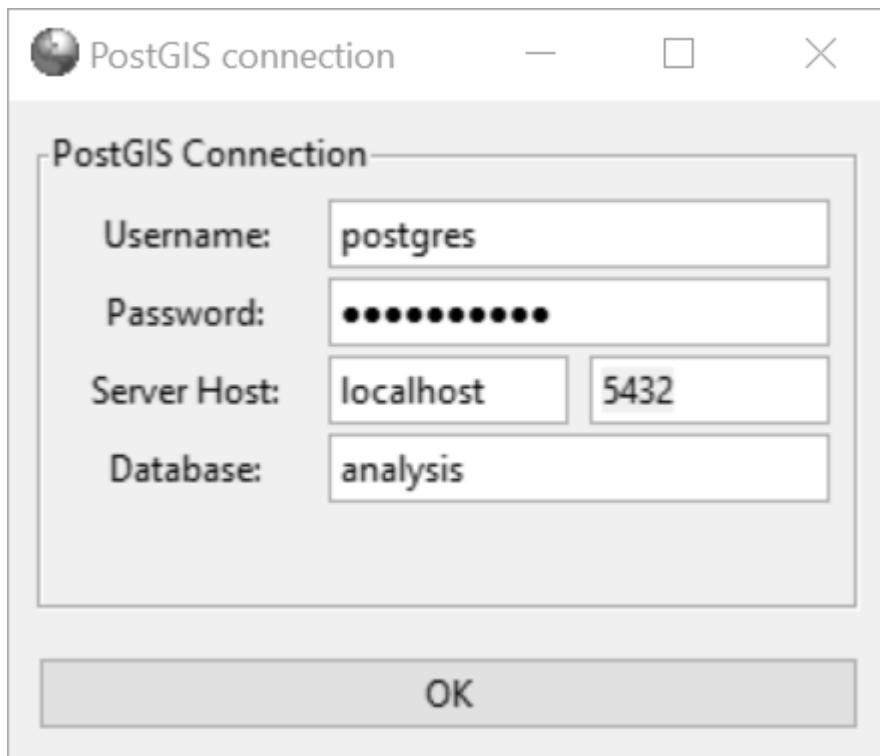


Figure 15-4: Establishing the PostGIS connection in the shapefile loader

Click **OK**. You should see the message `Connection Succeeded` in the log window. Now that you've successfully established the PostGIS connection, you can load your shapefile.

Under Options, change DBF file character encoding to **Latin1**—we do this because the shapefile attributes include county names with characters that require this encoding. Keep the default checked boxes, including the one to create an index on the spatial column. Click **OK**.

Click **Add File** and select *tl_2019_us_county.shp* from the location you saved it. Click **Open**. The file should appear in the Shapefile list in the loader, as shown in [Figure 15-5](#).

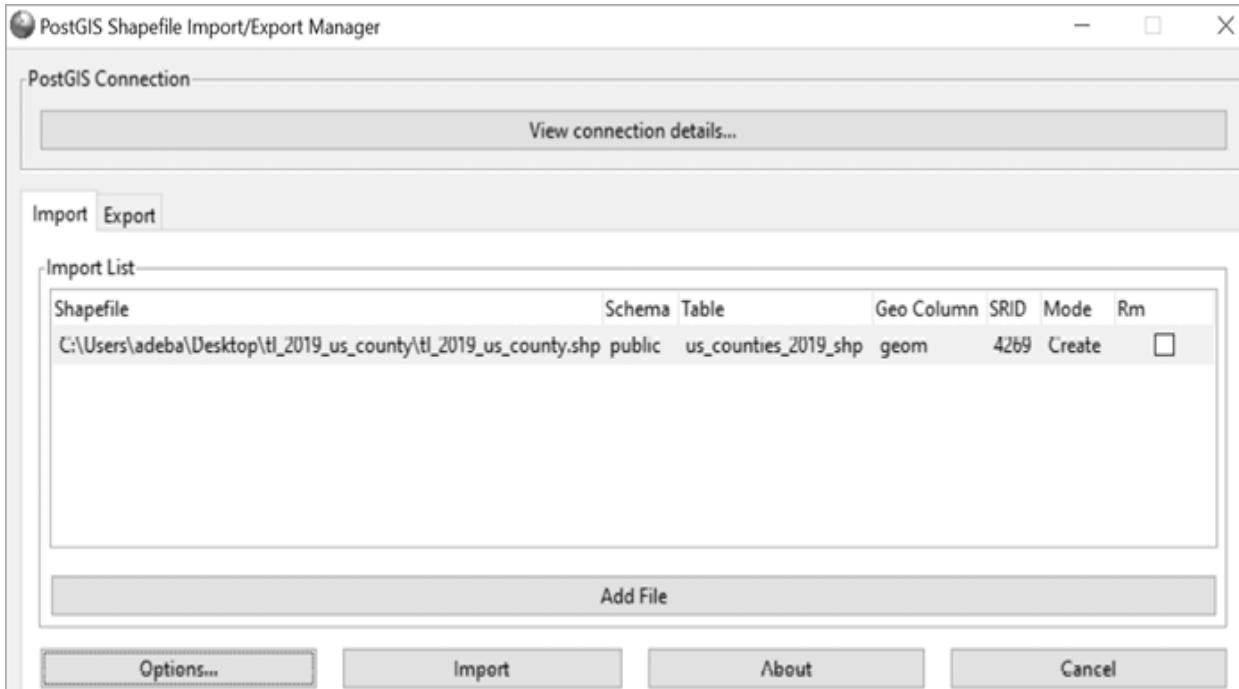


Figure 15-5: Specifying upload details in the shapefile loader

In the Table column, double-click to select the table name. Replace it with **us_counties_2019_shp**. Press ENTER to accept the value.

In the SRID column, double-click and enter **4269**. That's the ID for the North American Datum 1983 coordinate system, which is often used by US federal agencies including the US Census Bureau. Again, press ENTER to accept the value.

Click **Import**.

In the log window, you should see a message that ends with the following message:

```
Shapefile type: Polygon
PostGIS type: MULTIPOLYGON[2]
Shapefile import completed.
```

Switch to pgAdmin, and in the object browser, expand the analysis node and continue expanding by selecting **Schemas▶public▶Tables**. Refresh your tables by right-clicking **Tables** and selecting **Refresh** from the pop-up menu. You should see **us_counties_2019_shp** listed. Congrats! You've loaded your shapefile into a table. As part of the import, the shapefile loader also indexed

the geom column. You can move ahead to the section “Exploring the Census 2019 Counties Shapefile.”

Importing Shapefiles using shp2pgsql

The Shapefile Import/Export Manager isn’t available on all PostGIS distributions for macOS and Linux. For that reason, I’ll show you how to import shapefiles using the PostGIS command line tool `shp2pgsql`, which lets you accomplish the same thing using a single text command.

On macOS and Linux, you execute command line tools in the Terminal application. If you’re not familiar with working on the command line, you may want to pause here and read Chapter 18, “Using PostgreSQL from the Command Line,” to get set up. Otherwise, on macOS launch Terminal from your Applications folder (under Utilities); on Linux, open your distribution’s terminal.

At the command line, you use the following syntax to import a shapefile into a new table; the italicized code here are placeholders:

```
shp2pgsql -I -s SRID -W encoding shapefile_name table_name |  
psql -d database -U user
```

A lot’s happening here. Let’s look at each argument following the command:

`-I` adds an index on the new table’s geometry column using GiST.

`-s` lets you specify an SRID for the geometric data.

`-w` lets you specify file encoding, if necessary.

`shapefile_name` is the name (including full path) of the file ending with the `.shp` extension.

`table_name` indicates the new table you want the shapefile imported to.

Following these arguments, you place a pipe symbol (`|`) to direct the output of `shp2pgsql` to `psql`, the PostgreSQL command line utility. That’s followed by arguments for naming the database and user. For example, to load the `tl_2019_us_county.shp` shapefile from the book’s resources into a `us_counties_2019_shp` table in the `analysis` database, in your terminal you would move to the directory containing the shapefile and run the following command (all on one line):

```
shp2pgsql -I -s 4269 -W LATIN1 tl_2019_us_county.shp  
us_counties_2019_shp | psql -d analysis -U postgres
```

The server should respond with a number of SQL `INSERT` statements before creating the index and returning you to the command line. It might take some time to construct the entire set of arguments the first time around. But after you've done one, subsequent imports should take less time because you can simply substitute file and table names into the syntax you already wrote.

Load your shapefile, and then you'll be ready to explore the data with queries.

Exploring the Census 2019 Counties Shapefile

Your new `us_counties_2019_shp` table contains columns including each county's name as well as the *Federal Information Processing Standards (FIPS)* codes uniquely assigned to each state and county. The `geom` column contains the spatial data for each county's boundary. To start, let's check what kind of spatial object `geom` contains using the `ST_AsText()` function. Use the code in [Listing 15-14](#) to show the WKT representation of the first `geom` value in the table.

```
SELECT ST_AsText(geom)  
FROM us_counties_2019_shp  
ORDER BY gid  
LIMIT 1;
```

[Listing 15-14:](#) *Checking the `geom` column's WKT representation*

The result is a MultiPolygon with hundreds of coordinate pairs. Here's a portion of the output:

```
MULTIPOLYGON (((-97.019516 42.004097, -97.019519  
42.004933, -97.019527 42.007501, -97.019529 42.009755, -97.019529  
42.009776, -97.019529 42.009939, -97.019529 42.010163, -97.019538  
42.013931, -97.01955 42.014546, -97.01955 42.014565, -97.019551  
42.014608, -97.019551 42.014632, -97.01958 42.016158, -97.019622  
42.018384, -97.019629 42.018545, -97.01963 42.019475, -97.01963  
42.019553, -97.019644 42.020927, --snip-- )))
```

Each coordinate pair marks a Point on the boundary of the county, and remember that a `MULTIPOLYGON` object can contain a set of polygons. In the case of US counties, that will enable storage of counties whose boundaries

contain more than one distinct, separated area. Now, you're ready to analyze the data.

Finding the Largest Counties in Square Miles

Which county can claim the title of largest in area? To find the answer, [Listing 15-15](#) uses the `ST_Area()` function, which returns the area of a Polygon or MultiPolygon object. If you're working with a `geography` data type, `ST_Area()` returns the result in square meters. With a `geometry` data type—as used with this shapefile—the function returns the area in SRID units. Typically, those units are not useful for practical analysis, so we'll cast the `geometry` type to `geography` to obtain square meters. It's an intensive calculation, so expect extra time for this query to complete.

```
SELECT name,
       statefp AS st,
       round(
           ( ST_Area(1geom::geography) / 2589988.110336
        ) ::numeric, 2
       ) AS 3square_miles
  FROM us_counties_2019_shp
 ORDER BY square_miles 4DESC
 LIMIT 5;
```

[Listing 15-15](#): Finding the largest counties by area using `ST_Area()`

The `geom` column is data type `geometry`, so to find the area in square meters, we cast the `geom` column to a `geography` data type using the double-colon syntax **1**. Then, to get square miles, we divide the area by 2589988.110336, which is the number of square meters in a square mile **2**. To make the result easier to read, I've wrapped it in a `round()` function and named the resulting column `square_miles` **3**. Finally, we list the results in descending order from the largest area to the smallest **4** and use `LIMIT 5` to show the first five results, which should look like this:

name	st	square_miles
Yukon-Koyukuk	02	147871.00
North Slope	02	94827.92
Bethel	02	45559.08

Northwest Arctic	02	40619.78
Valdez-Cordova	02	40305.54

Congratulations to Alaska, where the boroughs (the name for county equivalents up there) are big. The five largest are all in Alaska, denoted by the state FIPS code 02. Yukon-Koyukuk, located in the heart of Alaska, is more than 147,800 square miles. (Keep that information in mind for the “Try It Yourself” exercise at the end of the chapter.)

Note that the shapefile doesn’t include a state name, just its FIPS code. Because the spatial data resides in a table, in the next section we’ll join to another census table to obtain the state name.

Finding a County by Longitude and Latitude

If you’ve ever wondered how spammy online ads seem to know where you live (“This one trick helped a Boston man fix his old shoes!”), it’s thanks to *geolocation services* that use various means, such as your phone’s GPS, to find your longitude and latitude. Given your coordinates, a spatial query can then determine which geography (a city or town, for example) that point falls into.

You can replicate this technique using your census shapefile and the `ST_Within()` function, which returns `true` if one geometry is inside another on the coordinate grid. [Listing 15-16](#) shows an example using the longitude and latitude of downtown Hollywood, California.

```
SELECT sh.name,
       c.state_name
  FROM us_counties_2019_shp sh JOIN us_counties_pop_est_2019 c
    ON sh.statefp = c.state_fips AND sh.countyfp =
   c.county_fips
 WHERE 1ST_Within(
      'SRID=4269;POINT(-118.3419063 34.0977076)' ::geometry,
      geom
     );
```

[Listing 15-16](#): Using `ST_Within()` to find the county belonging to a pair of coordinates

The `ST_Within()` function 1 inside the `WHERE` clause requires two geometry inputs and evaluates whether the first is inside the second. For the function to work properly, both `geometry` inputs must have the same SRID. In this example, the first input is an extended WKT representation of a Point that

includes the SRID 4269 (same as the census data), which is cast as a geometry type. The `ST_Within()` function doesn't accept a separate SRID input, so to set it for the supplied WKT, you must prefix it to the string like this:

`'SRID=4269;POINT (-118.3419063 34.0977076)'`. The second input is the `geom` column from the table.

Run the query; you should see the following result:

name	state_name
Los Angeles	California

It shows that the Point you supplied is within Los Angeles county in California. We also see how this technique can gain value (or raise privacy concerns) by relating a Point to data about its surrounding area—as we did here by joining to county population estimates. Suddenly, we can tell a lot about someone based on data describing where they spend time.

Try supplying other longitude and latitude pairs to see which US county they fall in. If you provide coordinates outside the United States, the query should return no results because the shapefile contains only US areas.

Examining Demographics Within a Distance

A fundamental metric for planners trying to locate a new school, business, or other community amenity is the number of people who live within a certain distance of it. Will there be enough people nearby to make construction worthwhile? To find the answer, we can use spatial and demographics data to estimate the population contained in the geographies within a certain distance of the planned location.

Say we're considering building a restaurant in downtown Lincoln, Nebraska, and we want to understand how many people live within 50 miles of the potential location. The code in [Listing 15-17](#) uses the `ST_DWithin()` function to find counties that have any portion of their boundary within 50 miles of downtown Lincoln and sum their estimated 2019 population.

```
SELECT sum(c.pop_est_2019) AS pop_est_2019
FROM us_counties_2019_shp sh JOIN us_counties_pop_est_2019 c
    ON sh.statefp = c.state_fips AND sh.countyfp =
c.county_fips
WHERE ST_DWithin(sh.geom::geography,
```

```
ST_GeogFromText('SRID=4269;POINT(-96.699656  
40.811567)',  
80467);
```

[Listing 15-17](#): Using `ST_DWithin()` to count people near Lincoln, Nebraska

In [Listing 15-10](#), we used `ST_DWithin()` to find farmers' markets close to Des Moines, Iowa. Here, we apply the same technique. We pass three arguments to `ST_DWithin()`: the census shapefile's `geom` column cast to the geography type; a point representing downtown Lincoln; and the distance of 50 miles using its equivalent in meters, 80,467.

The query should return a sum of 1,470,295, using the data from the joined census estimates table's `pop_est_2019` column.

Say we want to list the county names and visualize their borders in pgAdmin; we can modify our query, as in [Listing 15-18](#).

```
SELECT sh.name,  
       c.state_name,  
       c.pop_est_2019,  
       ST_Transform(sh.geom, 4326) AS geom  
FROM us_counties_2019_shp sh JOIN us_counties_pop_est_2019 c  
    ON sh.statefp = c.state_fips AND sh.countyfp =  
c.county_fips  
WHERE ST_DWithin(geom::geography,  
                 ST_GeogFromText('SRID=4269;POINT(-96.699656  
40.811567)'),  
                 80467);
```

[Listing 15-18](#): Displaying counties near Lincoln, Nebraska

This query should return 25 rows with the county name and its population. If you click the eye icon in the header of the `geom` column, you should see the counties displayed on a map in pgAdmin's Geometry Viewer, as in [Figure 15-6](#).

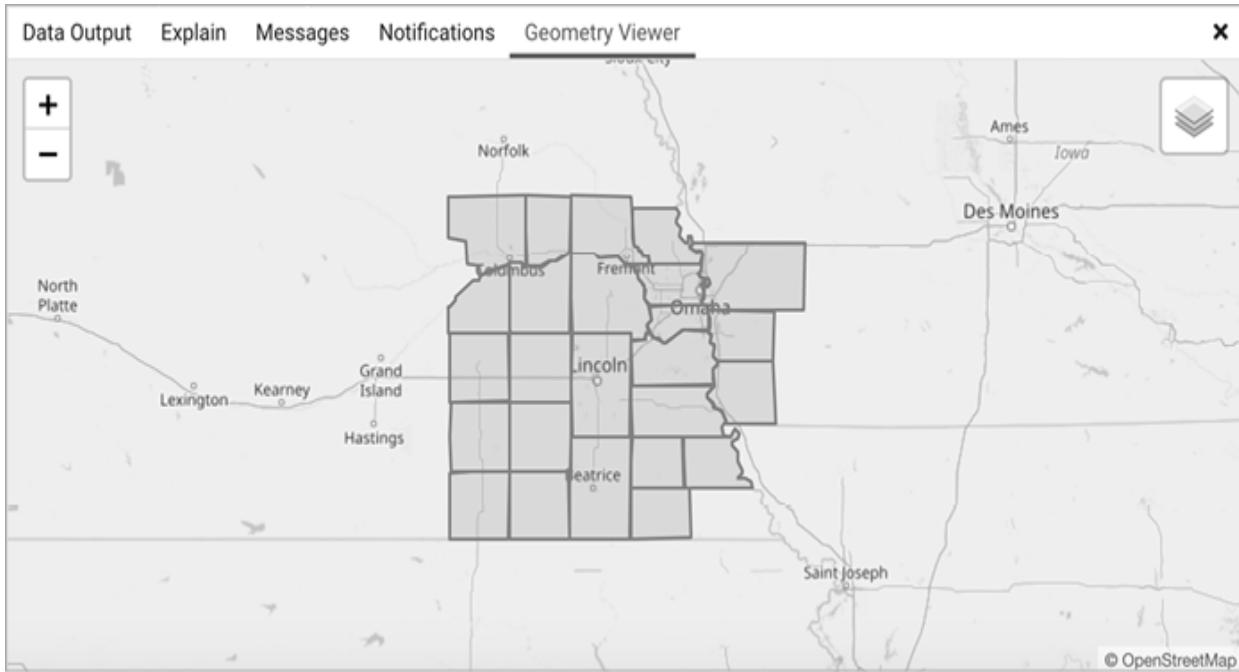


Figure 15-6: Counties that have a portion of their boundaries within 50 miles of Lincoln

These queries show counties that have any portion of their boundaries within 50 miles of Lincoln. Because counties tend to be large in area, they're a bit crude for determining an exact number of people within the distance of the point. For a more precise count, we could use smaller census geographies such as tracts or block groups, both of which are subcomponents of counties.

Finally, note that the pgAdmin Geometry Viewer's base map is the free OpenStreetMap, which uses the WGS 84 coordinate system. Our census shapefiles use a different coordinate system: North American Datum 83. For our data to display properly against the base map, we must use the `ST_Transform()` function [1](#) to convert the census geometry to the SRID of 4326. If we omit that function, the geographies will display on a blank canvas in the viewer because the coordinate systems don't match.

Performing Spatial Joins

Joining tables with spatial data opens up interesting opportunities for analysis. For example, you could join a table of coffee shops (which includes their longitude and latitude) to the counties table to find out how many shops exist

in each county based on their location. In this section, we'll explore spatial joins with a detailed look at roads and waterways using census data.

Exploring Roads and Waterways Data

Much of the year, the Santa Fe River, which cuts through the New Mexico state capital, is a dry riverbed better described as an *intermittent stream*. According to the Santa Fe city website, the river is susceptible to flash flooding and was named the nation's most endangered river in 2007. If you were an urban planner, it would help to know where the river intersects roadways so you could plan for emergency response when it floods.

You can find these locations using another set of US Census TIGER/Line shapefiles that has details on roads and waterways in Santa Fe County. These shapefiles are also included with the book's resources. Download and unzip *tl_2019_35049_linearwater.zip* and *tl_2019_35049_roads.zip*, and then import both using the same steps from earlier in the chapter. Name the water table *santafe_linearwater_2019* and the roads table *santafe_roads_2019*.

Next, refresh your database and run a quick `SELECT * FROM` query on both tables to view the data. You should have 11,655 rows in the roads table and 1,148 in the linear water table.

As with the counties shapefile, both tables have an indexed `geom` column of type `geometry`. It's helpful to check the type of spatial object in the column so you know the type of spatial feature you're querying. You can do that using the `ST_AsText()` function or `ST_GeometryType()`, as shown in [Listing 15-19](#).

```
SELECT ST_GeometryType(geom)
FROM santafe_linearwater_2019
LIMIT 1;
```

```
SELECT ST_GeometryType(geom)
FROM santafe_roads_2019
LIMIT 1;
```

[Listing 15-19](#): Using `ST_GeometryType()` to determine geometry

Both queries should return one row with the same value: `ST_MultiLineString`. That tell us that waterways and roads are stored as `MultiLineString` objects, a set of `LineStrings` that can be noncontinuous.

Joining the Census Roads and Water Tables

To find all the roads in Santa Fe that intersect the Santa Fe River, we'll join the roads and waterway tables with a query that tells us where the objects touch. We'll do this using the `ST_Intersects()` function, which returns a Boolean true if two spatial objects contact each other. Inputs can be either geometry or geography types. [Listing 15-20](#) joins the tables.

```
SELECT water.fullname AS waterway, 1
      roads.rtyp,
      roads.fullname AS road
  FROM santafe_linearwater_2019 water JOIN santafe_roads_2019
  roads 2
  3 ON ST_Intersects(water.geom, roads.geom)
 WHERE water.fullname = 4'Santa Fe Riv'
       AND roads.fullname IS NOT NULL
 ORDER BY roads.fullname;
```

[Listing 15-20](#): Spatial join with `ST_Intersects()` to find roads crossing the Santa Fe River

The SELECT column list 1 includes the `fullname` column from the `santafe_linearwater_2019` table, which gets `water` as its alias in the `FROM` 2 clause. The column list includes the `rtyp` code, which represents the route type, and `fullname` columns from the `santafe_roads_2019` table, aliased as `roads`.

In the `ON` portion 3 of the `JOIN` construct, we use the `ST_Intersects()` function with the `geom` columns from both tables as inputs. Here, the expression evaluates as true if the geometries intersect. We use `fullname` to filter the results to show only those that have the full string 'Santa Fe Riv' 4, which is how the Santa Fe River is listed in the water table. We also eliminate instances where road names are `NULL`. The query should return 37 rows; here are the first five:

waterway	rtyp	road
Santa Fe Riv	M	Baca Ranch Ln
Santa Fe Riv	M	Baca Ranch Ln
Santa Fe Riv	M	Caja del Oro Grant Rd
Santa Fe Riv	M	Caja del Oro Grant Rd

Each road in the results intersects with a portion of the Santa Fe River. The route type code for each of the first results is M, which indicates that the road name shown is its *common* name as opposed to a county or state recognized name, for example. Other road names in the complete results carry route types of C, S, or U (for unknown). The full route type code list is available at <https://www.census.gov/library/reference/code-lists/route-type-codes.html>.

Finding the Location Where Objects Intersect

We successfully identified roads that intersect the Santa Fe River. That's good, but it would really help to know the precise location of each intersection. We can modify the query to include the `ST_Intersection()` function, which returns the location of the place where objects touch. I've added it as a column in [Listing 15-21](#).

```
SELECT water.fullname AS waterway,
       roads.rttyp,
       roads.fullname AS road,
       1 ST_AsText(ST_Intersection(2water.geom, roads.geom))
  FROM santafe_linearwater_2019 water JOIN santafe_roads_2019
                                         roads
     ON ST_Intersects(water.geom, roads.geom)
 WHERE water.fullname = 'Santa Fe Riv'
   AND roads.fullname IS NOT NULL
 ORDER BY roads.fullname;
```

[Listing 15-21](#): Using `ST_Intersection()` to show where roads cross the river

The function returns a geometry object, so to view its WKT representation, we must wrap it in `ST_AsText()` 1. The `ST_Intersection()` function takes two inputs: the `geom` columns 2 from both the `water` and `roads` tables. Run the query, and the results should now include the exact coordinate location, or locations, where the river crosses the roads (I've rounded the Point coordinates for brevity).

waterway	rttyp	road	st_astext
-----	-----	-----	-----

```
Santa Fe Riv      M      Baca Ranch Ln      POINT (-106.049802  
35.642638)  
Santa Fe Riv      M      Baca Ranch Ln      POINT (-106.049743  
35.643126)  
Santa Fe Riv      M      Caja del Oro Grant Rd POINT (-106.024674  
35.657624)  
Santa Fe Riv      M      Caja del Oro Grant Rd POINT (-106.024692  
35.657644)  
Santa Fe Riv      M      Cam Carlos Rael     POINT (-105.986934  
35.672342)  
--snip--
```

Much better than poring over a map with a pencil, and this might prompt more ideas for analyzing spatial data. For example, if you have a shapefile of building footprints, you could find buildings near the river and in danger of flooding during heavy rains. Governments and private organizations regularly use these techniques as part of their planning process.

Wrapping Up

Mapping is a powerful analysis tool, and the techniques you learned in this chapter give you a strong start toward exploring more with PostGIS. You may indeed want to visualize this data, and that's entirely possible with a GIS application such as Esri's ArcGIS (<https://www.esri.com/>) or the free open source QGIS (<https://qgis.org/>). Both can use a PostGIS-enabled PostgreSQL database as a data source, allowing you to visualize shapefile data in your tables or the results of queries.

You've now added working with geographic data to your analysis skills. Next, we'll explore another widely used data type called JavaScript Object Notation (JSON) and how PostgreSQL enables storing and querying it.

TRY IT YOURSELF

Use the spatial data you've imported in this chapter to try additional analysis:

Earlier, you found which US county has the largest area. Now, aggregate the county data to find the area of each state in square miles. (Use the `statefp` column in the `us_counties_2019_shp` table.) How many states are bigger than the Yukon-Koyukuk area?

Using `ST_Distance()`, determine how many miles separate these two farmers' markets: The Oakleaf Greenmarket (9700 Argyle Forest Blvd, Jacksonville, Florida) and Columbia Farmers Market (1701 West Ash Street, Columbia, Missouri). You'll need to first find the coordinates for both in the `farmers_markets` table. Tip: you can also write this query using the Common Table Expression syntax you learned in Chapter 13.

More than 500 rows in the `farmers_markets` table are missing a value in the `county` column, which is an example of dirty government data. Using the `us_counties_2019_shp` table and the `ST_Intersects()` function, perform a spatial join to find the missing county names based on the longitude and latitude of each market. Because `geog_point` in `farmers_markets` is of the `geography` type and its SRID is 4326, you'll need to cast `geom` in the `census` table to the `geography` type and change its SRID using `ST_SetSRID()`.

The `nyc_yellow_taxi_trips` table you created in Chapter 12 contains the longitude and latitude where each trip began and ended. Use PostGIS functions to turn the drop-off coordinates into a `geometry` type and count the state/county pairs where each drop-off occurred. As with the previous exercise, you'll need to join to the `us_counties_2019_shp` table and use its `geom` column for the spatial join.

16

WORKING WITH JSON DATA



JavaScript Object Notation (JSON) is a widely used text format for storing data in a platform-independent way so it can be shared between computer systems. In this chapter, you'll learn the structure of JSON as well as how to store and query JSON data types in PostgreSQL. After we explore PostgreSQL's JSON query operators, we'll analyze a month's worth of data about earthquakes.

The American National Standards Institute (ANSI) SQL standard added syntax definitions for JSON and specified functions for creating and accessing JSON objects in 2016. Major database systems have added JSON support in recent years as well, although implementations vary. PostgreSQL, for example, supports some of the ANSI standard while implementing a number of nonstandard operators. I'll note which aspects of PostgreSQL's JSON support are part of standard SQL as we work through exercises.

Understanding JSON Structure

JSON data primarily comprises two structures: an *object*, which is an unordered set of name/value pairs, and an *array*, which is an ordered collection of values. If you've used programming languages such as JavaScript, Python, or C#, these aspects of JSON should look familiar.

Inside an object, we use name/value pairs as a structure for storing and referencing individual data items. The object in its entirety is enclosed within curly brackets, and each name, more often referred to as a *key*, is enclosed in double quotes, followed by a colon and its corresponding value. The object can encapsulate multiple key/value pairs, separated by commas. Here's an example using movie information:

```
{"title": "The Incredibles", "year": 2004}
```

The keys are `title` and `year`, and their values are `"The Incredibles"` and `2004`. If the value is a string, it goes in double quotes. If it's a number, a Boolean value, or a `null`, we omit the quotes. If you're familiar with the Python language, you'll recognize this structure as a *dictionary*.

An array is an ordered list of values enclosed in square brackets. We separate each value in the array with a comma. For example, we might list movie genres like so:

```
["animation", "action"]
```

Arrays are common in programming languages, and we've used them already in SQL queries. In Python, this structure is called a *list*.

We can create many permutations of these structures, including nesting objects and arrays inside each other. For example, we can create an array of objects or use an array as the value of a key. We can add or omit key/value pairs or create additional arrays of objects without violating a preset schema. This flexibility—in contrast to the strict definition of a SQL table—is both part of the appeal of using JSON as a data store as well as one of the biggest difficulties in working with JSON data.

As an example, [Listing 16-1](#) shows information about two of my favorite films stored as JSON. The outermost structure is an array with two elements—one object for each film. We know the outermost structure is an array because the entire JSON begins and ends with square brackets.

```
[{1
    "title": "The Incredibles",
    "year": 2004,
2"rating": {
    "MPAA": "PG"
```

```

        },
3"characters": [
    "name": "Mr. Incredible",
    "actor": "Craig T. Nelson"
], {
    "name": "Elastigirl",
    "actor": "Holly Hunter"
}, {
    "name": "Frozone",
    "actor": "Samuel L. Jackson"
}],
4"genre": ["animation", "action", "sci-fi"]
}, {
    "title": "Cinema Paradiso",
    "year": 1988,
    "characters": [
        {
            "name": "Salvatore",
            "actor": "Salvatore Cascio"
        },
        {
            "name": "Alfredo",
            "actor": "Philippe Noiret"
        }
    ],
    "genre": ["romance", "drama"]
}
]

```

[Listing 16-1](#): JSON with information about two films

Inside the outermost array, each film object is surrounded by curly brackets. The open brace at 1 starts the object for the first film *The Incredibles*. For both films, we store the `title` and `year` as key/value pairs, and they have string and integer values, respectively. The third key, `rating` 2, has a JSON object for its value. That object contains a single key/value pair showing the film's rating from the Motion Picture Association of America.

Here we can see the flexibility JSON affords us as a storage medium. First, if we later wanted to add another country's rating for the film, we could easily add a second key/value pair to the `rating` value object. Second, we're not required to include `rating`—or any key/value pair—in every film object. In fact, I omitted a `rating` for *Cinema Paradiso*. If a particular piece of data isn't available, in this case a rating, some systems that generate JSON might simply exclude that pair. Other systems might include `rating` but with a `null` value. Both are valid, and that flexibility is one of JSON's advantages: its data definition, or *schema*, can flex as needed.

The final two key/value pairs show other ways to structure JSON. For characters **3**, the value is an array of objects, with each object surrounded by curly brackets and separated by a comma. The value for genre **4** is an array of strings.

Considering When to Use JSON with SQL

There are advantages to using *NoSQL* or *document* databases that store data in JSON or other text-based data formats, as opposed to the relational tables SQL uses. Document databases are flexible in terms of data definitions. You can redefine a data structure on the fly if needed. Document databases are often also used for high-volume applications because they can be scaled by adding servers. On the flip side, you may give up SQL advantages such as easily added constraints that enforce data integrity and support for transactions.

The arrival of JSON support in SQL has made it possible to enjoy the best of both worlds by adding JSON data as columns in relational tables. The decision to use a SQL or NoSQL database should be multifaceted. PostgreSQL performs favorably relative to NoSQL in terms of speed, but we must also consider the kinds and volume of data being stored, the applications being served, and more.

That said, some cases where you might want to take advantage of JSON in SQL include the following:

When users or applications need to arbitrarily create key/value pairs. For example, if tagging a collection of medical research papers, one user might want to add a key to track chemical names, and another user might want a key to track food names.

When storing related data in a JSON column instead of a separate table. An employees table could have the usual columns for name and contact information plus a JSON column with a flexible collection of key/value pairs that might hold additional attributes that don't apply to every employee, such as company awards or performance metrics.

When saving time by analyzing JSON data fetched from other systems without first parsing it into a set of tables.

Keep in mind that using JSON in PostgreSQL or other SQL databases can also present challenges. Constraints that are trivial to set up on regular SQL tables

can be more difficult to set and enforce on JSON data. JSON data can consume more space as key names get repeated in text along with the quotes, commas, and braces that define its structure. Finally, the flexibility of JSON can create issues for the code that interacts with it—whether SQL or another language—if keys unexpectedly disappear or the data type of a value changes.

Keeping all this in mind, let's review PostgreSQL's two JSON data types and load some JSON into a table.

Using json and jsonb Data Types

PostgreSQL provides two data types for storing JSON. Both allow insertion of valid JSON only—text that includes required elements of the JSON specification, such as open and closing curly brackets around an object, commas separating objects, and proper quoting of keys. If you try to insert invalid JSON, the database will generate an error.

The main difference between the two is that one stores JSON as text and the other as binary data. The binary implementation is newer to PostgreSQL and generally preferred because it's faster at querying and has indexing capabilities.

The two types are as follows:

json

Stores JSON as text, keeping white space and maintaining the order of keys. If a single JSON object contains a particular key more than once (which is valid), the `json` type will preserve each of the repeated key/value pairs. Finally, each time a database function processes `json`-stored text, it must parse the object to interpret its structure. This can make reads from the database slower than with the `jsonb` type. Indexing is not supported. Typically, the `json` type is useful when an application has duplicate keys or needs to preserve the order of keys.

jsonb

Stores JSON in a binary format, removing white space and not maintaining the order of keys. If a single JSON object contains a particular key more than once, the `jsonb` type will preserve only the last of the key/value pairs. The binary format adds some overhead to writing data to the table, but processing is faster. Indexing is supported.

Neither `json` nor `jsonb` is part of the ANSI SQL standard, which doesn't specify a JSON data type and leaves it to database makers to decide how to implement support. The PostgreSQL documentation at <https://www.postgresql.org/docs/current/datatype-json.html> recommends using `jsonb` unless there's a need to preserve the order of key/value pairs.

We'll use `jsonb` exclusively in the remainder of the chapter, both because of speed considerations but also because many of PostgreSQL's JSON functions work the same way with both `json` and `jsonb`—and there are more functions available for `jsonb`. We'll continue by adding the films JSON from [Listing 16-1](#) to a table and exploring JSON query syntax.

Importing and Indexing JSON Data

The file `films.json` in the Chapter 16 folder of the book's resources at <https://nostarch.com/practical-sql-2nd-edition/> contains a modified form of the JSON in [Listing 16-1](#). View the file with a text editor, and you'll see each film's JSON object is placed on a single line, with no line breaks between elements. I've also removed the outermost square brackets and the comma separating the two film objects. Each remains a valid JSON object:

```
{"title": "The Incredibles", "year": 2004, --snip-- }
 {"title": "Cinema Paradiso", "year": 1988, --snip-- }
```

I set up the file this way so that PostgreSQL's `COPY` command will interpret each film's JSON object as a separate row on import, the same way it does when importing a CSV file. The code in [Listing 16-2](#) makes a simple `films` table with a surrogate primary key and a `jsonb` column called `film`.

```
CREATE TABLE films (
    id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    film jsonb NOT NULL
);

COPY films (film)
| FROM C:\YourDirectory\films.json';

? CREATE INDEX idx_film ON films USING GIN (film);
```

[Listing 16-2](#): Creating a table to hold JSON data and adding an index

Note that the `COPY` statement ends with the `FROM` clause `1` instead of continuing to include a `WITH` statement as in previous examples. The reason we no longer need the `WITH` statement, which we've used to specify options for file headers and CSV formatting, is that this file has no header and isn't delimited. We just want the database to read each line and process it.

After import, we add an index `2` to the `jsonb` column using the GIN index type. We discussed the generalized inverted index (GIN) with full-text search in Chapter 14. GIN's implementation of indexing the location of words or key values within text is particularly suited to JSON data. Note that because index entries point to rows in a table, `jsonb` column indexing works best when each row contains a relatively small chunk of JSON—as opposed to a table with one row that has a single, enormous JSON value and repeated keys.

Execute the commands to create and fill the table and add the index. Run `SELECT * FROM films;` and you should see two rows containing the autogenerated `id` and the JSON object text. Now you're ready to explore querying the data using PostgreSQL's JSON operators.

Using json and jsonb Extraction Operators

To retrieve values from our stored JSON, we can use PostgreSQL-specific *extraction operators*, which return either a JSON object, an element of an array, or an element that exists at a path in the JSON structure we specify. [Table 16-1](#) shows the operators and their functions, which can vary based on the data type of the input. Each works with `json` and `jsonb` data types.

Table 16-1: *json* and *jsonb* Extraction Operators

Operator, syntax	Function	Returns
<i>json -> text</i> <i>jsonb -> text</i>	Extracts a key value, specified as text	<i>json</i> or <i>jsonb</i> (matching the input)
<i>json ->> text</i> <i>jsonb ->> text</i>	Extracts a key value, specified as text	text
<i>json -> integer</i> <i>jsonb -> integer</i>	Extracts an array element, specified as an integer denoting its array position	<i>json</i> or <i>jsonb</i> (matching the input)
<i>json ->> integer</i> <i>jsonb ->> integer</i>	Extracts an array element, specified as an integer denoting its array position	text
<i>json #> text array</i> <i>jsonb #> text array</i>	Extracts a JSON object at a specified path	<i>json</i> or <i>jsonb</i> (matching the input)
<i>json #>> text array</i> <i>jsonb #>> text array</i>	Extracts a JSON object at a specified path	text

Let's try the operators with our films JSON to learn more about how they vary in function.

Key Value Extraction

In [Listing 16-3](#) we use the `->` and `->>` operators followed by text naming the key value to retrieve. In that context, with text input, these are called *field extraction operators* because they extract a field, or key value, from the JSON. The difference between the two is that `->` returns the key value as JSON in the same type as stored, and `->>` returns the key value as text.

```
SELECT id, film ->1 'title' AS title  
FROM films  
ORDER BY id;
```

```
SELECT id, film ->>2 'title' AS title  
FROM films  
ORDER BY id;
```

```
SELECT id, film ->3 'genre' AS genre  
FROM films  
ORDER BY id;
```

[Listing 16-3](#): Retrieving a JSON key value with field extraction operators

In the SELECT list, we specify our JSON column name followed by the operator and the key name in single quotes. In the first example, the syntax `-> 'title'` 1 returns the value of the `title` key as JSON in the same data type as stored, `jsonb`. Run the first query, and you should see the output like this:

id	title
1	"The Incredibles"
2	"Cinema Paradiso"

In pgAdmin, the data type listed in the `title` column header should indicate `jsonb`, and the film titles remain quoted, as they are in the JSON object.

Changing the field extraction operator to `->> 2` returns the film titles as text instead:

id	title
1	The Incredibles
2	Cinema Paradiso

Finally, we'll return an array. In our films JSON, the value of the key genre is an array of values. Using the field extraction operator `-> 3` returns the array as JSON:

id	genre
<hr/>	
1	["animation", "action", "sci-fi"]
2	["romance", "drama"]

If we used `->>` here, we'd return the arrays as text. Let's look at how to extract elements from an array.

Array Element Extraction

To retrieve a specific value from an array, we follow the `->` and `->>` operators with an integer specifying the value's position, or *index*, in the array. We call these *element extraction operators* because they retrieve an element from a JSON array. As with field extraction, `->` returns the value as JSON in the same type as stored, and `->>` returns it as text.

[Listing 16-4](#) shows four examples using the array values of "genre".

```
SELECT id, film -> 'genre' -> 01 AS genres
FROM films
ORDER BY id;
```

```
SELECT id, film -> 'genre' -> -12 AS genres
FROM films
ORDER BY id;
```

```
SELECT id, film -> 'genre' -> 23 AS genres
FROM films
ORDER BY id;
```

```
SELECT id, film -> 'genre' ->> 04 AS genres
FROM films
ORDER BY id;
```

[Listing 16-4](#): Retrieving a JSON array value with element extraction operators

We must first retrieve the array value from the key as JSON and then retrieve the desired element from the array. In the first example, we specify the JSON

column `film`, followed by the field extraction operator `->` and the `genre` key name in single quotes. This returns the `genre` value as `jsonb`. We follow the key name with `->` and the integer `0 1` to get the first element.

Why not use `1` for the first value in the array? In many languages, including Python and JavaScript, index values start at zero, and that's also true when accessing JSON arrays with SQL.

NOTE

SQL arrays have a different ordering scheme than JSON arrays in PostgreSQL. The first element in a SQL array is at position 1; in a JSON array, the first element is at position 0.

Run the first query, and your results should look like this, showing the first element in each film's `genre` array, returned as `jsonb`:

```
id    genres
-- -----
1    "animation"
2    "romance"
```

We can also access the last element of the array, even if we aren't sure of its index, because the number of genres per film can vary. We count backward from the end of the list using a negative index number. Supplying `-1 2` tells `->` to get the first element from the end of the list:

```
id    genres
-- -----
1    "sci-fi"
2    "drama"
```

We can count back further if we want—an index of `-2` will get the next-to-last element.

Note that PostgreSQL won't return an error if there's no element at the supplied index position; it will simply return a `NULL` for that row. For example, if we supply `2 3` for the index, we see results for one of our films and a `NULL` for the other:

```
id  genres
-- -----
1  "sci-fi"
2
```

We get a `NULL` back for *Cinema Paradiso* because it has only two elements in its `genre` value array, and index 2 (since we count up starting with zero) represents the third element. Later in the chapter, we'll learn how to count array lengths.

Finally, changing the element extraction operator to `->> 4` returns the desired element as a `text` data type rather than JSON:

```
id  genres
-- -----
1  animation
2  romance
```

This is the same pattern as we saw when extracting key values: `->` returns a JSON data type, and `->>` returns text.

Path Extraction

Both `#>` and `#>>` are *path extraction operators* that return an object located at a JSON path. A path is a series of keys or array indices that lead to the location of a value. In our example JSON, it might be just the `title` key if we want the name of the film. Or it could be more complex, such as the `characters` key followed by an index value of 1, then the `actor` key; this would provide the path to the name of the actor at index 1. The `#>` path extraction operator returns a JSON data type matching the stored data, and `#>>` returns text.

Consider the MPAA rating for the film *The Incredibles*, which appears in our JSON like this:

```
"rating": {
    "MPAA": "PG"
}
```

The structure is a key named `rating` with an object for its value; inside that object is a key/value pair with `MPAA` as the key name. Thus, the path to the film's MPAA rating begins with the `rating` key and ends with the `MPAA` key. To denote the path's elements, we use the PostgreSQL string syntax for arrays,

creating a comma-separated list inside curly brackets and single quotes. We then feed that string to the path extraction operators. [Listing 16-5](#) shows three examples of setting paths.

```
SELECT id, film #> '{rating, MPAA}'1 AS mpaa_rating
FROM films
ORDER BY id;
```

```
SELECT id, film #> '{characters, 0, name}'2 AS name
FROM films
ORDER BY id;
```

```
SELECT id, film #>> '{characters, 0, name}'3 AS name
FROM films
ORDER BY id;
```

[Listing 16-5](#): Retrieving a JSON key value with path extraction operators

To get each film's MPAA rating, we specify the path in an array: {rating, MPAA} **1** with each item separated by commas. Run the query, and you should see these results:

id	mpaa_rating
1	"PG"
2	

The query returns the PG rating for *The Incredibles* and a NULL for *Cinema Paradiso* because, in our data, the latter film has no MPAA rating present.

The second example works with the array of characters, which in our JSON looks like this:

```
"characters": [ {
    "name": "Salvatore",
    "actor": "Salvatore Cascio"
}, {
    "name": "Alfredo",
    "actor": "Philippe Noiret"
}]
```

The `characters` array shown is for the second movie, but both films have a similar structure. Array objects each represent a character and the name and the actor who played them. To locate the name of the first character in the array, we specify a path **2** that starts at the `characters` key, continues to the first element of the array using the index `0`, and ends at the `name` key. The query results should look like this:

<code>id</code>	<code>name</code>
<hr/>	
1	"Mr. Incredible"
2	"Salvatore"

The `#>` operator returns results as a JSON data type, in our case `jsonb`. If we want the results as text, we use `#>> 3` with the same path.

Containment and Existence

The final collection of operators we'll explore performs two kinds of evaluations. The first concerns *containment* and checks whether a specified JSON value contains a second specified JSON value. The second tests for *existence*: whether a string of text within a JSON object exists as a top-level key (or as an element of an array nested inside a deeper object). Both kinds of operators return a Boolean value, which means we can use them in a `WHERE` clause to filter query results.

This set of operators works only with the `jsonb` data type—another good reason to favor `jsonb` over `json`—and can make use of our GIN index for efficient searching. [Table 16-2](#) lists the operators with their syntax and function.

Table 16-2: *jsonb Containment and Existence Operators*

Operator, syntax	Function	Return s
<code>jsonb @> jsonb</code>	Tests whether the first JSON value contains the second JSON value	boolean
<code>jsonb <@ jsonb</code>	Tests whether the second JSON value contains the first JSON value	boolean
<code>jsonb ? text</code>	Tests whether the text exists as a top-level (not nested) key or an array value	boolean
<code>jsonb ? text array</code>	Tests whether any of the text elements in the array exist as a top-level (not nested) key or as an array value	boolean
<code>jsonb ?& text array</code>	Tests whether all of the text elements in the array exist as a top-level (not nested) key or as an array value	boolean

Using Containment Operators

In [Listing 16-6](#), we use `@>` to evaluate whether one JSON value contains a second JSON value.

```
SELECT id, film ->> 'title' AS title,
       film @>1 '{"title": "The Incredibles"}'::jsonb AS
is_incredible
FROM films
ORDER BY id;
```

[Listing 16-6:](#) Demonstrating the `@>` containment operator

In our `SELECT` list, we check whether the JSON stored in the `film` column in each row contains the key/value pair for *The Incredibles*. We use the `@>` containment operator **1** in an expression that generates a column with the Boolean result `true` if `film` contains `"title": "The Incredibles"`. We give the name of our JSON column, `film`, then the `@>` operator, and then a string (cast to `jsonb`) specifying the key/value pair. In our `SELECT` list, we also return the text of the film title as a column. Running the query should produce these results:

id	title	is_incredible
1	The Incredibles	true
2	Cinema Paradiso	false

As expected, the expression evaluates to `true` for *The Incredibles* and `false` for *Cinema Paradiso*.

Because the expression evaluates to a Boolean result, we can use it in a query's WHERE 2 clause, as shown in [Listing 16-7](#).

```
SELECT film ->> 'title' AS title,
       film ->> 'year' AS year
  FROM films
 WHERE film @> '{"title": "The Incredibles"}'::jsonb;
```

[Listing 16-7:](#) Using a containment operator in a WHERE clause

Here we again check that the JSON in the `film` column contains the key/value pair for the title of *The Incredibles*. By placing the evaluation in a WHERE clause, the query should return just the row where the expression returns `true`:

title	year
The Incredibles	2004

Finally, in [Listing 16-8](#), we flip the order of evaluation to check whether the key/value pair specified is contained within the `film` column.

```
SELECT film ->> 'title' AS title,
       film ->> 'year' AS year
  FROM films
 WHERE '{"title": "The Incredibles"}'::jsonb <@3 film;
```

[Listing 16-8:](#) Demonstrating the <@ containment operator

Here we use the <@ operator 3 instead of @> to flip the order of evaluation. This expression also evaluates to `true`, returning the same result as the previous query.

Using Existence Operators

Next, in [Listing 16-9](#), we explore three existence operators. These check whether the text we supply exists as a top-level key or as an element of an array. All return a Boolean value.

```
SELECT film ->> 'title' AS title
FROM films
WHERE film ?1 'rating';

SELECT film ->> 'title' AS title,
      film ->> 'rating' AS rating,
      film ->> 'genre' AS genre
FROM films
WHERE film ?|2 '{rating, genre}';

SELECT film ->> 'title' AS title,
      film ->> 'rating' AS rating,
      film ->> 'genre' AS genre
FROM films
WHERE film ?&3 '{rating, genre}';
```

[Listing 16-9](#): Demonstrating existence operators

The `? operator` checks for the existence of a single key or array element. In the first query's `WHERE` clause, we give the `film` column, the `? operator` `1`, and then the string `rating`. This syntax says, “In each row, does `rating` exist as a key in the JSON in the `film` column?” When we run the query, the results show the one film that has a `rating` key, *The Incredibles*.

The `?|` and `?&` operators act as `or` and `and`. For example, using `?| 2` tests whether either `rating` or `genre` exist as top-level keys. Running that second query returns both films, because both have at least one of those keys. Using `?& 3`, however, tests whether both `rating` and `genre` exist as keys, and that's true for only *The Incredibles*.

All these operators provide options for fine-tuning your exploration of your JSON data. Now, let's use some of them on a larger dataset.

Analyzing Earthquake Data

In this section, we'll analyze a collection of JSON data about earthquakes compiled by the US Geological Survey, an agency of the US Department of the

Interior that monitors natural phenomenon including volcanoes, landslides, and water quality. The USGS uses a network of seismographs that record the earth's vibrations, compiling data on each seismic event's location and intensity. Minor earthquakes occur around the world many times a day; the big ones are less frequent but potentially devastating.

For our exercise, I fetched a month's worth of JSON-formatted earthquake data from a USGS *application programming interface*, better known as an API. An *API* is a resource for transmitting data and commands between computers, and JSON is often used for APIs. You'll find the data in the file *earthquakes.json* in the folder for this chapter included in the book's resources.

Exploring and Loading the Earthquake Data

[Listing 16-10](#) shows the data structure for each earthquake record in the file, along with a selection of its key/value pairs (your *Chapter_16.sql* file has the nonsnipped version).

```
{  
    "type": "Feature", 1  
    "properties":2 {  
        "mag": 1.44,  
        "place": "134 km W of Adak, Alaska",  
        "time": 1612051063470,  
        "updated": 1612139465880,  
        "tz": null,  
        --snip--  
        "felt": null,  
        "cdi": null,  
        "mmi": null,  
        "alert": null,  
        "status": "reviewed",  
        "tsunami": 0,  
        "sig": 32,  
        "net": "av",  
        "code": "91018173",  
        "ids": ",av91018173,",  
        "sources": ",av,",  
        "types": ",origin,phase-data,",  
        "nst": 10,  
        "dmin": null,  
        "rms": 0.15,  
        "gap": 174,  
        "magType": "ml",
```

```

    "type": "earthquake",
    "title": "M 1.4 - 134 km W of Adak, Alaska"
},
"geometry":3 {
    "type": "Point",
    "coordinates": [-178.581, 51.84183333333333, 22.48]
},
"id": "av91018173"
}

```

[Listing 16-10:](#) JSON with data on one earthquake

This data is in *GeoJSON* format, a JSON-based specification for spatial data. GeoJSON will include one or more Feature objects, denoted by inclusion of the key/value pair "type": "Feature" **1**. Each Feature describes a single spatial object and contains both descriptive attributes (such as event time or related codes) under properties **2** plus a geometry **3** key that includes the coordinates of the spatial object. In our data, each geometry is a Point, a simple feature with the coordinates of one earthquake's longitude, latitude, and depth in kilometers. We discussed Points and simple features in Chapter 15 when working with PostGIS; GeoJSON incorporates it and other spatial simple features. You can read more about the GeoJSON specification at <https://geojson.org/> and see definitions of the keys in the USGS documentation at <https://earthquake.usgs.gov/data/comcat/data-eventterms.php>.

Let's load our data into a table called `earthquakes` using the code in [Listing 16-11](#).

```

CREATE TABLE earthquakes (
    id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    earthquake jsonb1 NOT NULL
);

COPY earthquakes (earthquake)
FROM C:\YourDirectory\earthquakes.json';

2 CREATE INDEX idx_earthquakes ON earthquakes USING GIN
(earthquake);

```

[Listing 16-11:](#) Creating and loading an earthquakes table

As with our `films` table, we use `COPY` to copy the data into a single `jsonb` column **1** and add a GIN index **2**. Running `SELECT * FROM earthquakes;` should return 12,899 rows. Now let's see what we can learn from the data.

Working with Earthquake Times

The `time` key/value pair represents the moment the earthquake occurred. In [Listing 16-12](#), we retrieve the value of `time` using a path extraction operator.

```
SELECT id, earthquake #>> '{properties, time}'1 AS time
FROM earthquakes
ORDER BY id LIMIT 5;
```

[Listing 16-12](#): Retrieving the earthquake time

In the `SELECT` list, we give the `earthquake` column followed by a `#>>` path extraction operator and the path **1** to the `time` value denoted as an array. The `#>>` operator will return our value as text. Running the query should return five rows:

id	time
--	-----
1	1612137592990
2	1612137479762
3	1612136740672
4	1612136207600
5	1612135893550

If those values don't look like times to you, that's not surprising. By default, the USGS represents time as milliseconds since the Unix epoch at 00:00 UTC on January 1, 1970. That's a variant of the standard epoch time we covered in Chapter 12, which measures seconds since the epoch. We can convert this USGS `time` value to something understandable using `to_timestamp()` and a little math, as shown in [Listing 16-13](#).

```
SELECT id, earthquake #>> '{properties, time}' as time,
1 to_timestamp(
    (earthquake #>> '{properties, time}')::bigint /
10002
) AS time_formatted
```

```
FROM earthquakes
ORDER BY id LIMIT 5;
```

[Listing 16-13](#): Converting the time value to a timestamp

Inside the parentheses of the `to_timestamp()` **1** function, we repeat the code to extract the `time` value. The `to_timestamp()` function requires a number representing seconds, but the extracted value is text and in milliseconds, so we also cast the extracted text to `bigint` and divide by 1,000 **2** to convert it to seconds.

On my machine, the query generates the following results showing the extracted `time` value and its converted timestamp (your values will vary depending on your PostgreSQL server's time zone, so `time_formatted` will show when the earthquake occurred in your server's time zone time):

id	time	time_formatted
1	1612137592990	2021-01-31 18:59:52-05
2	1612137479762	2021-01-31 18:57:59-05
3	1612136740672	2021-01-31 18:45:40-05
4	1612136207600	2021-01-31 18:36:47-05
5	1612135893550	2021-01-31 18:31:33-05

Now that we have an understandable timestamp, let's find the oldest and newest earthquake times using the `min()` and `max()` aggregate functions in [Listing 16-14](#).

```
SELECT min1(to_timestamp(
    (earthquake #>> '{properties, time}')::bigint /
1000
) ) AT TIME ZONE 'UTC'2 AS
min_timestamp,
max3(to_timestamp(
    (earthquake #>> '{properties, time}')::bigint /
1000
) ) AT TIME ZONE 'UTC' AS
max_timestamp
FROM earthquakes;
```

[Listing 16-14](#): Finding the minimum and maximum earthquake times

We place `to_timestamp()` and our milliseconds-to-seconds conversion inside both the `min()` 1 and `max()` 3 functions in our `SELECT` list. This time, we add the keywords `AT TIME ZONE 'UTC'` 2 after both functions; regardless of our server time zone settings, the results will display the timestamps in UTC, as USGS records them. Your results should look like this:

min_timestamp	max_timestamp
2021-01-01 00:01:39	2021-01-31 23:59:52

This collection of earthquakes spans a month—from early morning January 1, 2021, through the end of day on January 31. That’s helpful context as we continue to dig for usable information.

Finding the Largest and Most-Reported Earthquakes

Next, we’ll look at two data points that measure an earthquake’s size and the degree to which citizens reported feeling it and apply JSON extraction techniques to simple sorting of results.

Extracting by Magnitude

The USGS reports each earthquake’s magnitude in the `mag` key, beneath `properties`. Magnitude, according to the USGS, is a number representing the size of an earthquake at its source. Its scale is logarithmic: a magnitude 4 earthquake has seismic waves whose amplitude is about 10 times bigger than a quake with a magnitude of 3. With that context, let’s find the five largest earthquakes in our data using the code in [Listing 16-15](#).

```
SELECT earthquake #>> '{properties, place}'1 AS place,
       to_timestamp((earthquake #>> '{properties,
time}')::bigint / 1000)
          AT TIME ZONE 'UTC' AS time,
          (earthquake #>> '{properties, mag}')::numeric AS
magnitude
FROM earthquakes
? ORDER BY (earthquake #>> '{properties, mag}')::numeric3 DESC
NULLS LAST
LIMIT 5;
```

[Listing 16-15](#): *Finding the five earthquakes with the largest magnitude*

We again use path extraction operators to retrieve our desired elements, including values for `place` 1 and `mag`. To show the largest five in our results, we add an `ORDER BY` clause 2 with `mag`. We cast the value to numeric 3 here and in the `SELECT` because we want to display and sort the value as a number rather than as text. We also add the `DESC NULLS LAST` keywords, which sorts the results in descending order and places `NULL` values (of which there are two) last. Your results should look like this:

	place	time
magnitude		
211 km SE of Pondaguitan, Philippines	2021-01-21 12:23:04	
7		
South Shetland Islands	2021-01-23 23:36:50	
6.9		
30 km SSW of Turt, Mongolia	2021-01-11 21:32:59	
6.7		
28 km SW of Pocito, Argentina	2021-01-19 02:46:21	
6.4		
Kermadec Islands, New Zealand	2021-01-08 00:28:50	
6.3		

The largest, of magnitude 7, was located beneath the ocean southeast of the small city of Pondaguitan in the Philippines. The second was in the Antarctic near the South Shetland Islands.

Extracting by Citizen Reports

The USGS operates a Did You Feel It? website at <https://earthquake.usgs.gov/data/dyfi/> where people can report their earthquake experiences. Our JSON includes the number of reports for each earthquake under the key `felt`, beneath `properties`. Let's see which earthquakes in our data generated the most reports using the code in [Listing 16-16](#).

```
SELECT earthquake #>> '{properties, place}' AS place,
       to_timestamp((earthquake #>> '{properties,
time}')::bigint / 1000)
          AT TIME ZONE 'UTC' AS time,
       (earthquake #>> '{properties, mag}')::numeric AS
magnitude,
       (earthquake #>> '{properties, felt}')::integer1 AS
felt
```

```
FROM earthquakes
ORDER BY (earthquake #>> '{properties, felt}')::integer2 DESC
NULLS LAST
LIMIT 5;
```

[Listing 16-16](#): Finding earthquakes with the most Did You Feel It? reports

Structurally, this query is similar to [Listing 16-15](#) that found the largest quakes. We add a path extraction operator for the `felt` **1** key, casting the returned text value to an `integer` type. We cast to `integer` so the extracted text is treated as a number for sorting and display. Finally, we place the extraction code in `ORDER BY 2`, using `NULLS LAST` because there are many earthquakes with no reports and we want those to appear last in the list. You should see these results:

place	time	magnitude	felt
4km SE of Aromas, CA	2021-01-17 04:01:27	4.2	19907
2km W of Concord, CA	2021-01-14 19:18:10	3.63	5101
10km NW of Pinnacles, CA	2021-01-02 14:42:23	4.32	3076
2km W of Willowbrook, CA	2021-01-20 16:31:58	3.52	2086
3km NNW of Santa Rosa, CA	2021-01-19 04:22:20	2.68	1765

The top five are in California, which makes sense. Did You Feel It? is a US government-run system, so we'd expect more US reports—particularly in earthquake-prone California. Also, some of the largest quakes in our data occurred beneath oceans or in remote regions. The quake with more than 19,900 reports was moderate, but its nearness to cities meant more chance for people to notice it.

Converting Earthquake JSON to Spatial Data

Our JSON data has longitude and latitude values for each earthquake, meaning we can perform spatial analysis using the GIS techniques discussed in Chapter 15. For example, we'll use a PostGIS distance function to locate earthquakes that occurred within 50 miles from a city. First, though, we must convert the coordinates stored in JSON to a PostGIS data type.

The longitude and latitude values are found in the array of the `coordinates` key, under `geometry`. Here's an example:

```
"geometry": {
    "type": "Point",
    "coordinates": [-178.581, 51.841833333333, 22.48]
}
```

The first coordinate, at position 0 in the array, represents longitude; the second, at position 1, is latitude. The third value denotes depth in kilometers, which we won't use. To extract these elements as text, we make use of a #>> path operator, as in [Listing 16-17](#).

```
SELECT id,
       earthquake #>> '{geometry, coordinates}' AS
coordinates,
       earthquake #>> '{geometry, coordinates, 0}' AS
longitude,
       earthquake #>> '{geometry, coordinates, 1}' AS latitude
FROM earthquakes
ORDER BY id
LIMIT 5;
```

[Listing 16-17](#): Extracting the earthquake's location data

The query should return five rows:

id	coordinates	longitude	latitude
1	[-122.852, 38.8228333, 2.48]	-122.852	38.8228333
2	[-148.3859, 64.2762, 16.2]	-148.3859	64.2762
3	[-152.489, 59.0143, 73]	-152.489	59.0143
4	[-115.82, 32.7493333, 9.85]	-115.82	32.7493333
5	[-115.6446667, 33.1711667, 5.89]	-115.6446667	33.1711667

A quick visual compare of our result to the JSON longitude and latitude values tells us we've extracted the values properly. Next, we'll use a PostGIS function to convert those values to a Point in the geography data type.

NOTE

Your analysis database must have PostGIS enabled for this part of the chapter. If you skipped Chapter 15, you can enable PostGIS by running CREATE EXTENSION postgis; in pgAdmin.

[Listing 16-18](#) generates a Point of type geography for each earthquake, which we can use as input for PostGIS spatial functions.

```
SELECT ST_SetSRID(
    ST_MakePoint1(
        (earthquake #>> '{geometry, coordinates,
0}')::numeric,
        (earthquake #>> '{geometry, coordinates,
1}')::numeric
    ),
    43262)::geography AS earthquake_point
FROM earthquakes
ORDER BY id;
```

[Listing 16-18: Converting JSON location data to PostGIS geography](#)

Inside `ST_MakePoint()`¹, we place our code to extract longitude and latitude, casting both values to type `numeric` as required by the function. We nest that function inside `ST_SetSRID()` to set a spatial reference system identifier (SRID) for the resulting Point. In Chapter 15, you learned that the SRID specifies a coordinate grid for plotting spatial objects. The SRID value `4326`² denotes the commonly used WGS 84 coordinate system. Finally, we cast the entire output to the `geography` type. The first several rows should look like this:

```
earthquake_point
-----
0101000020E6100004A0C022B87B65EC0A6C7009A52694340
0101000020E610000D8F0F44A598C62C0EFC9C342AD115040
0101000020E6100000cff753E3A50F63C0992A1895D4814D40
--snip--
```

We can't interpret those strings of digits and letters directly, but we can use pgAdmin's Geometry Viewer to see the Points plotted on a map. With your query results visible in the pgAdmin Data Output pane, click the eye icon in the `earthquake_point` result header. You should see the earthquakes plotted on a map that uses OpenStreetMap as the base layer, as in [Figure 16-1](#).

Even with only a month of data, it's easy to see the abundance of earthquakes concentrated around the edges of the Pacific Ocean, in the so-called Ring of Fire where tectonic plates meet and volcanos are more active.

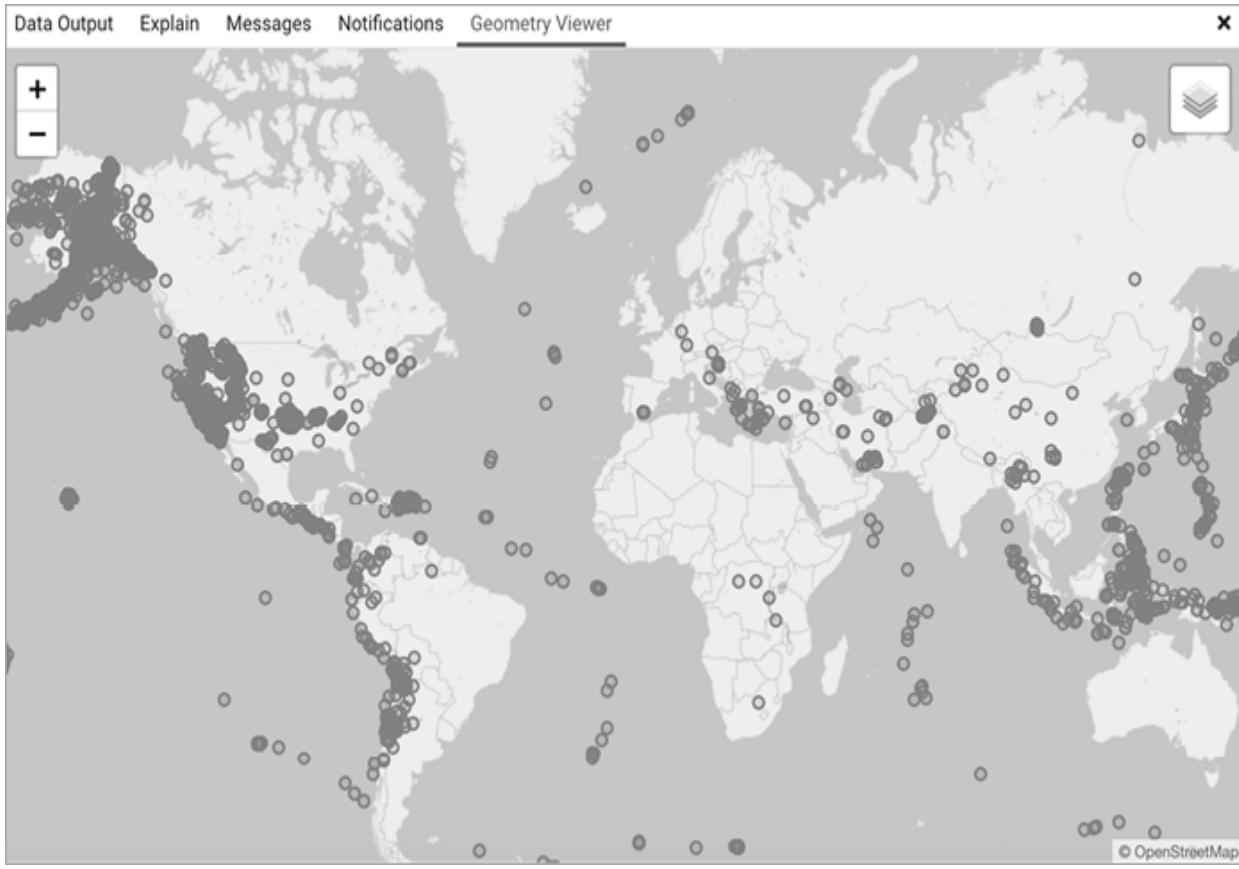


Figure 16-1: Viewing earthquake locations in pgAdmin

Finding Earthquakes Within a Distance

Next, let's narrow our study to earthquakes that occurred near Tulsa, Oklahoma —a part of the country that has seen increased seismic activity since 2009 as a result of oil and gas processing, according to the USGS.

To perform more complex GIS tasks like this, it's easier if we permanently convert the JSON coordinates to a column of PostGIS type geography in the `earthquakes` table. That allows us to avoid the clutter of adding conversion code in each query.

[Listing 16-19](#) adds a column called `earthquake_point` to the `earthquakes` table and fills the new column with the JSON coordinates converted to type `geography`.

```
| ALTER TABLE earthquakes ADD COLUMN earthquake_point  
|   geography(POINT, 4326);
```

```

2 UPDATE earthquakes
    SET earthquake_point =
        ST_SetSRID(
            ST_MakePoint(
                (earthquake #>> '{geometry, coordinates,
0}')::numeric,
                (earthquake #>> '{geometry, coordinates,
1}')::numeric
            ),
            4326)::geography;

3 CREATE INDEX quake_pt_idx ON earthquakes USING GIST
(earthquake_point);

```

[Listing 16-19](#): Converting JSON coordinates to a PostGIS geometry column

We use ALTER TABLE **1** to add a column `earthquake_point` of type `geography`, specifying that the column will hold Points with an SRID of 4326. Next, we UPDATE **2** the table, setting the `earthquake_point` column using the same syntax as in [Listing 16-18](#), and add a spatial index using GIST **3** to the new column.

That done, we can use [Listing 16-20](#) to find earthquakes within 50 miles of Tulsa.

```

SELECT earthquake #>> '{properties, place}' AS place,
       to_timestamp((earthquake -> 'properties' ->
'time')::bigint / 1000)
           AT TIME ZONE 'UTC' AS time,
       (earthquake #>> '{properties, mag}')::numeric AS
magnitude,
       earthquake_point
FROM earthquakes
| WHERE ST_DWithin(earthquake_point,
2 ST_GeogFromText('POINT(-95.989505
36.155007)'),
80468)
ORDER BY time;

```

[Listing 16-20](#): Finding earthquakes within 50 miles of downtown Tulsa, Oklahoma

In the WHERE clause **1**, we employ the `ST_DWithin()` function, which returns a Boolean value of `true` if one spatial object is within a specified distance of

another object. Here, we want to evaluate each earthquake Point to check whether it's within 50 miles of downtown Tulsa. We designate the city's coordinates in `ST_GeogFromText()`² and supply the value of 50 miles using its meters equivalent, 80468, as meters is the required input. The query should return 19 rows (I've omitted the `earthquake_point` column and truncated the results for brevity):

place	time	magnitude
4 km SE of Owasso, Oklahoma	2021-01-04 19:46:58	1.53
6 km SSE of Cushing, Oklahoma	2021-01-05 08:04:42	0.91
2 km SW of Hulbert, Oklahoma	2021-01-05 21:08:28	1.95
--snip--		

View the earthquake locations by clicking the eye icon atop the `earthquake_point` column in the results in pgAdmin. You should see 19 dots around the city, as in [Figure 16-2](#) (and you can adjust the underlying map style by clicking the layer icon at top right).

Achieving these results required some coding gymnastics that would have been unnecessary if the data had arrived in a shapefile or in a typical SQL table. Nevertheless, it's possible to extract meaningful insights from JSON data using PostgreSQL's support for the format. In the last part of the chapter, we'll cover useful PostgreSQL functions for generating and manipulating JSON.

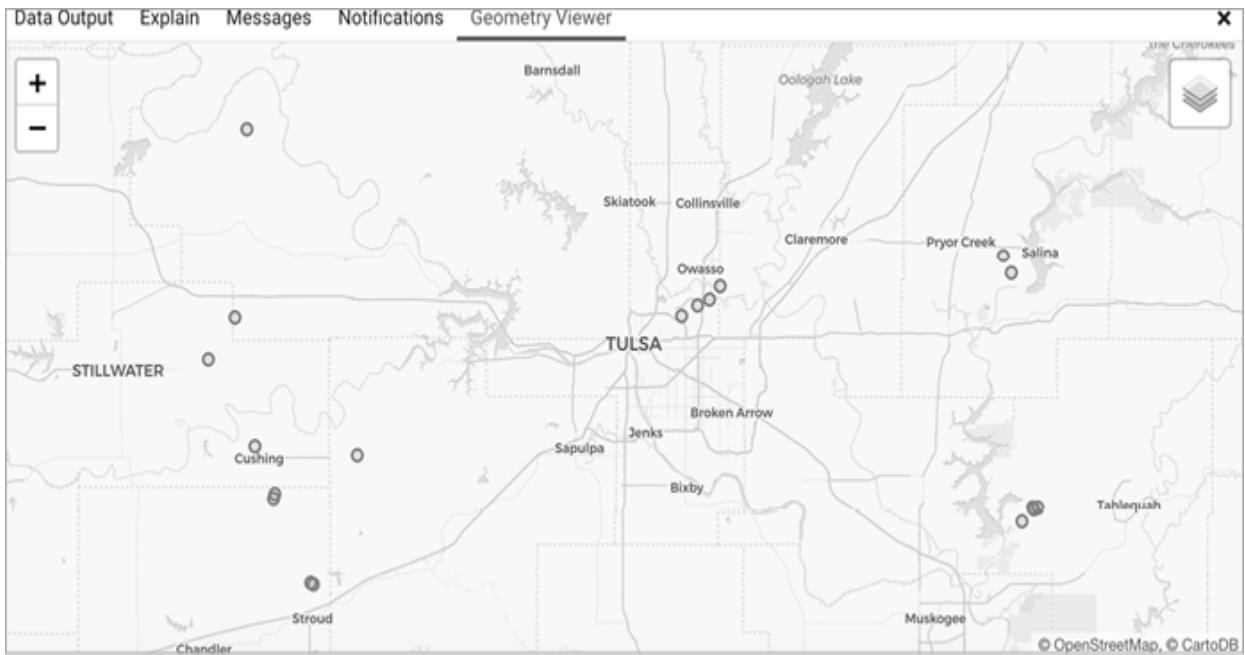


Figure 16-2: Viewing earthquakes near Tulsa, Oklahoma, in pgAdmin

Generating and Manipulating JSON

We can use PostgreSQL functions to create JSON from existing rows in a SQL table or to modify JSON stored in a table to add, subtract, or change keys and values. The PostgreSQL documentation at <https://www.postgresql.org/docs/current/functions-json.html> lists several dozen JSON-related functions—we'll work through a few you might find handy.

Turning Query Results into JSON

Because JSON is primarily a format for sharing data, it's useful to be able to quickly convert the results of a SQL query into JSON for delivery to another computer system. [Listing 16-21](#) uses the PostgreSQL-specific `to_json()` function to turn rows from the `employees` table you made in Chapter 7 into JSON.

```
| SELECT to_json(employees) AS json_rows
  FROM employees;
```

[Listing 16-21: Turning query results into JSON with `to_json\(\)`](#)

The `to_json()` function does what it says: transforms a supplied SQL value to JSON. To convert all values in each row of the `employees` table, we use `to_json()` in a `SELECT 1` and supply the table name as the function's argument; that returns each row as a JSON object with column names as keys:

```
        json_rows
-----
-----
{"emp_id":1,"first_name":"Julia","last_name":"Reyes","salary":115300.00,"dept_id":1}
 {"emp_id":2,"first_name":"Janet","last_name":"King","salary":98000.00,"dept_id":1}
 {"emp_id":3,"first_name":"Arthur","last_name":"Pappas","salary":72700.00,"dept_id":2}
 {"emp_id":4,"first_name":"Michael","last_name":"Taylor","salary":89500.00,"dept_id":2}
```

We can modify our query a few ways to limit which columns to include in the results. In [Listing 16-22](#), we use a `row()` constructor as the argument for `to_json()`.

```
SELECT to_json(row(emp_id, last_name)) 1 AS json_rows
FROM employees;
```

[Listing 16-22](#): Specifying columns to convert to JSON

A `row()` constructor (which is ANSI SQL compliant) builds a `row` value from the arguments passed to it. In this case, we supply the column names `emp_id` and `last_name` 1 and place `row()` inside `to_json()`. This syntax returns just those columns in the JSON result:

```
        json_rows
-----
-----
{"f1":1,"f2":"Reyes"}
 {"f1":2,"f2":"King"}
 {"f1":3,"f2":"Pappas"}
 {"f1":4,"f2":"Taylor"}
```

Notice, however, that the keys are named `f1` and `f2` instead of their source column names. That's a side effect of `row()`, which doesn't preserve column names when it builds the `row` record. We can set the names of the keys, which

is often done to keep the names short and reduce JSON file size, improving transfer speeds. [Listing 16-23](#) shows how via a subquery.

```
SELECT to_json(employees) AS json_rows
FROM (
    1 SELECT emp_id, last_name AS ln2 FROM employees
) AS employees;
```

[Listing 16-23](#): Generating key names with a subquery

We write a subquery **1** that grabs the columns we want and alias the result as `employees`. In the process, we alias a column name **2** to shorten its appearance as a key in the JSON.

The results should look like this:

json_rows
{ "emp_id":1,"ln":"Reyes"}
{ "emp_id":2,"ln":"King"}
{ "emp_id":3,"ln":"Pappas"}
{ "emp_id":4,"ln":"Taylor"}

Finally, [Listing 16-24](#) shows how to compile all the rows of JSON into a single array of objects. You may want to do this if you're providing this data to another application that will iterate over the array of objects to perform a task, such as a calculation, or to render data on a device.

```
| SELECT json_agg(to_json(employees)) AS json
|   FROM (
|       SELECT emp_id, last_name AS ln FROM employees
|   ) AS employees;
```

[Listing 16-24](#): Aggregating the rows and converting to JSON

We wrap `to_json()` in the PostgreSQL-specific `json_agg()` **1** function, which aggregates values, including `NULL`, into a JSON array. Its output should look like this:

json
[{"emp_id":1,"ln":"Reyes"}, {"emp_id":2,"ln":"King"}, {"emp_id":3,"ln":"Pappas"}, {"emp_id":4,"ln":"Taylor"}]

```
-----  
[{"emp_id":1,"ln":"Reyes"}, {"emp_id":2,"ln":"King"},  
 {"emp_id":3,"ln":"Pappas"}, --snip-- ]
```

These are simple examples, but you can build more complex JSON structures using subqueries to generate nested objects. We'll consider one way to do that as part of our “Try It Yourself” exercises at the end of the chapter.

Adding, Updating, and Deleting Keys and Values

We can add to, update, and delete from JSON with a combination of concatenation and PostgreSQL-specific functions. Let's work through some examples.

Adding or Updating a Top-Level Key/Value Pair

In [Listing 16-25](#), we return to our `films` table and add a top-level key/value pair `"studio": "Pixar"` to the film *The Incredibles* using two different techniques:

```
UPDATE films  
SET film = film || 1 '{"studio": "Pixar"}'::jsonb  
WHERE film @> '{"title": "The Incredibles"}'::jsonb;  
  
UPDATE films  
SET film = film || jsonb_build_object('studio', 'Pixar') 2  
WHERE film @> '{"title": "The Incredibles"}'::jsonb;
```

[Listing 16-25](#): Adding a top-level key/value pair via concatenation

Both examples use `UPDATE` statements to set new values for the `jsonb` column `film`. In the first, we use the PostgreSQL concatenation operator `|| 1` to combine the existing film JSON with the new key value/pair that we cast to `jsonb`. In the second, we use concatenation again but with `jsonb_build_object()`. This function takes a series of key and value names as arguments and returns a `jsonb` object, letting us concatenate several key/value pairs at a time if we wanted.

Both statements will insert the new key/value pair if the key doesn't exist in the JSON being concatenated; it will overwrite a key that's present. There's no functional difference between the two statements, so feel free to use whichever

you prefer. Note that this behavior is specific to `jsonb`, which doesn't allow duplicate key names.

If you `SELECT * FROM films;` and double-click the updated data in the `film` column, you should see the new key/value pair:

```
--snip--  
    "rating": {  
        "MPAA": "PG"  
    },  
    "studio": "Pixar",  
    "characters": [  
--snip--
```

Updating a Value at a Path

Currently we have two entries for the `genre` key for *Cinema Paradiso*:

```
"genre": ["romance", "drama"]
```

To add a third entry to the array, we use the function `jsonb_set()`, which allows us to specify a value to update at a specific JSON path. In [Listing 16-26](#), we use the `UPDATE` statement and `jsonb_set()` to add the genre `World War II`.

```
UPDATE films  
SET film = jsonb_set(film, 1  
                      '{genre}', 2  
                      film #> '{genre}' || '['"World War II"]', 3  
                      true4)  
WHERE film @> '{"title": "Cinema Paradiso"}'::jsonb;
```

[Listing 16-26](#): Adding an array value at a path with `jsonb_set()`

In `UPDATE`, we `SET` the value of `film` to the result of `jsonb_set()` and use `WHERE` to limit the update to just the row with *Cinema Paradiso*. The function's first argument **1** is the target JSON we want to modify, here `film`. The second argument is the path **2** to the array value—the `genre` key. Third, we give the new value for `genre`, which we specify as the current value of `genre` concatenated with an array **3** with one value, `"World War II"`. That

concatenation will produce an array with three elements. The final argument is an optional Boolean value **4** that dictates whether `jsonb_set()` should create the value if it's not already present. It's redundant here since `genre` already exists; I've shown it for reference.

Run the query and then perform a quick `SELECT` to check the updated JSON. You should see the `genre` array including three values: `["romance", "drama", "World War II"]`.

Deleting a Value

We can remove keys and values from a JSON object by pairing two operators. [Listing 16-27](#) shows two `UPDATE` examples.

```
UPDATE films
SET film = film -1 'studio'
WHERE film @> '{"title": "The Incredibles"}'::jsonb;

UPDATE films
SET film = film #-2 '{genre, 2}'
WHERE film @> '{"title": "Cinema Paradiso"}'::jsonb;
```

[Listing 16-27](#): Deleting values from JSON

The minus sign **1** acts as a *deletion operator*, removing the key `studio` and its value, which we added earlier for *The Incredibles*. Supplying a text string after the minus sign indicates we want to remove a key and its value; supplying an integer will remove the element at that index.

The **#- 2** sign is a *path deletion operator* that removes the JSON element that exists at a path we specify. The syntax is similar to that of the path extraction operators `#>` and `#>>`. Here, we use `{genre, 2}` to indicate the third element of the array for `genre` (remember, JSON array indexes begin counting at zero). This will remove the value `World War II` that we added earlier to *Cinema Paradiso*.

Run both statements and then use `SELECT` to view the altered `film` JSON. You should see both elements removed.

Using JSON Processing Functions

To finish our JSON studies, we'll review a selection of PostgreSQL-specific functions for processing JSON data, including expanding array values into table rows and formatting output. You can find a complete listing of functions in the PostgreSQL documentation at <https://www.postgresql.org/docs/current/functions-json.html>.

Finding the Length of an Array

Counting the number of items in an array is a routine programming and analysis task. We might, for example, want to know how many actors are stored for each film in our JSON. To do this, we can use the `jsonb_array_length()` function in [Listing 16-28](#).

```
SELECT id,
       film ->> 'title' AS title,
       1 jsonb_array_length(film -> 'characters') AS
       num_characters
  FROM films
 ORDER BY id;
```

[Listing 16-28](#): Finding the length of an array

As its only argument, the function `1` takes an expression that extracts the value of the `character` key from `film`. Running the query should produce these results:

id	title	num_characters
1	The Incredibles	3
2	Cinema Paradiso	2

The output correctly shows that we have three characters for *The Incredibles* and two for *Cinema Paradiso*. Note there's a similar `json_array_length()` function for the `json` type.

Returning Array Elements as Rows

The `jsonb_array_elements()` and `jsonb_array_elements_text()` functions convert array elements into rows, with one row per element. This is a useful tool for data processing. To convert JSON into structured SQL data, for

example, we could use this function to generate the rows to `INSERT` into a table or to generate rows that we can aggregate by grouping and counting.

[Listing 16-29](#) uses both functions to turn the `genre` key's array values into rows. Each function takes a `jsonb` array as an argument. The difference between the two is that `jsonb_array_elements()` returns the array elements as rows of `jsonb` values, while `jsonb_array_elements_text()` returns elements as, you guessed it, `text`.

```
SELECT id,
       jsonb_array_elements(film -> 'genre') AS genre_jsonb,
       jsonb_array_elements_text(film -> 'genre') AS
genre_text
FROM films
ORDER BY id;
```

[Listing 16-29](#): Returning array elements as rows

Running the code should produce these results:

id	genre_jsonb	genre_text
1	"animation"	animation
1	"action"	action
1	"sci-fi"	sci-fi
2	"romance"	romance
2	"drama"	drama

On an array with a simple list of values, that works nicely, but if an array contains a collection of JSON objects with their own key/value pairs, like `character` in our `film` JSON, we need additional processing to unpack the values first. [Listing 16-30](#) walks through the process.

```
SELECT id,
       jsonb_array_elements(film -> 'characters') 1
FROM films
ORDER BY id;

? WITH characters (id, json) AS (
    SELECT id,
           jsonb_array_elements(film -> 'characters')
    FROM films
)
```

```
3 SELECT id,
       json ->> 'name' AS name,
       json ->> 'actor' AS actor
  FROM characters
 ORDER BY id;
```

[Listing 16-30](#): Returning key values from each item in an array

We use `jsonb_array_elements()` to return the elements of the `characters` **1** array, which should return each JSON object in the array as a row:

id	jsonb_array_elements
1	{"name": "Mr. Incredible", "actor": "Craig T. Nelson"}
1	{"name": "Elastigirl", "actor": "Holly Hunter"}
1	{"name": "Frozone", "actor": "Samuel L. Jackson"}
2	{"name": "Salvatore", "actor": "Salvatore Cascio"}
2	{"name": "Alfredo", "actor": "Philippe Noiret"}

To convert the `name` and `actor` values to columns, we employ a common table expression (CTE) as covered in Chapter 13. Our CTE **2** uses `jsonb_array_elements()` to generate a simple temporary `characters` table with two columns: the film's `id` and the unpacked array values in a column called `json`. We follow with a `SELECT` statement **3** that queries the temporary table, extracting the values of `name` and `actor` from the `json` column:

id	name	actor
1	Mr. Incredible	Craig T. Nelson
1	Elastigirl	Holly Hunter
1	Frozone	Samuel L. Jackson
2	Salvatore	Cascio
2	Alfredo	Philippe Noiret

Those values are neatly parsed into a standard SQL structure and suitable for further analysis using standard SQL.

Wrapping Up

JSON is such a ubiquitous format that it's likely you'll encounter it often in your journey analyzing data. You've learned that PostgreSQL easily handles loading,

indexing, and parsing JSON, but JSON sometimes requires extra steps to process that aren't needed with data handled via standard SQL conventions. As with many areas of coding, your decision on whether to make use of JSON will depend on your specific circumstances. Now, you're equipped to understand the context.

JSON itself is a standard, but the data types and the majority of functions and syntax in this chapter were PostgreSQL-specific. That's because the ANSI SQL standard leaves it to database vendors to decide how to implement most JSON support. If your work involves using Microsoft SQL Server, MySQL, SQLite, or another system, consult their documentation. You'll find many similarities in capabilities even if the function names differ.

TRY IT YOURSELF

Use your new JSON skills to answer these questions:

The earthquakes JSON has a key `tsunami` that's set to a value of `1` for large earthquakes in oceanic regions (though it doesn't mean a tsunami actually happened). Using either path or field extraction operators, find earthquakes with a `tsunami` value of `1` and include their location, time, and magnitude in your results.

Use the following `CREATE TABLE` statement to add the table `earthquakes_from_json` to your `analysis` database:

```
CREATE TABLE earthquakes_from_json (
    id text PRIMARY KEY,
    title text,
    type text,
    quake_date timestamp with time zone,
    mag numeric,
    place text,
    earthquake_point geography(POINT, 4326),
    url text
);
```

Using field and path extraction operators, write an `INSERT` statement to fill the table with the correct values for each earthquake. Refer to the full sample earthquake JSON in your `Chapter_16.sql` file for any key names and paths you need.

Bonus (difficult) question: Try writing a query to generate the following JSON using the data in the `teachers` and `teachers_lab_access` tables from Chapter 13:

```
{
    "id": 6,
    "fn": "Kathleen",
    "ln": "Roush",
    "lab_access": [
        {
            "lab_name": "Science B",
            "access_time": "2022-12-17T16:00:00-05:00"
        },
        {
            "lab_name": "Science A",
            "access_time": "2022-12-07T10:02:00-05:00"
        }
    ]
}
```

It's helpful to remember that the `teachers` table has a one-to-many relationship with `teachers_lab_access`; the first three keys must come from `teachers`, and the JSON objects in the array of `lab_access` will come from `teachers_lab_access`. Hint: you'll need to use a subquery in your `SELECT` list and a function called `json_agg()` to create the `lab_access` array. If you're stumped, take a peek in your `Try_It_Yourself.sql` file included with the book's resources, where I've placed the answers to all these exercises.

17

SAVING TIME WITH VIEWS, FUNCTIONS, AND TRIGGERS



One advantage of using a programming language is that we can automate repetitive, boring tasks. That's what this chapter is about: taking the queries or steps you might do over and over and turning them into reusable database objects that you code once and can call later to let the database do the work. Programmers call this the DRY principle: Don't Repeat Yourself.

You'll start by learning to store queries as reusable database *views*. Next, you'll explore how to create database functions you can use to operate on your data, the same way you've used built-in functions like `round()` and `upper()`. Then you'll set up *triggers* to run your functions automatically when certain events occur on a table. All these techniques are useful not only for reducing repetitive work but for ensuring data integrity too.

We'll practice these techniques on tables created from examples in earlier chapters. All the code for this chapter is available for download along with the book's resources at <https://nostarch.com/practical-sql-2nd-edition/>.

Using Views to Simplify Queries

A *view* is essentially a stored query with a name that you can work with as if it were a table. For example, a view might store a query that calculates total population by state. As with a table, you could query that view, join the view to tables (or to other views), and use the view to update or insert data into a table it's based on, albeit with some caveats. The stored query in a view can be simple, referencing just one table, or complex, with multiple table joins.

Views are especially useful in the following scenarios:

Avoiding duplicate effort: They let you write a complex query once and access the results when needed.

Reducing clutter: They can trim the amount of information you need to wade through by showing only columns relevant to your needs.

Providing security: Views can limit access to only certain columns in a table.

In this section, we'll look at two kinds of views. The first—a standard view—contains PostgreSQL syntax that's largely in line with the ANSI SQL standard for views. Every time you access a standard view, the stored query runs and generates a temporary set of results. The second is a *materialized view*, which is specific to PostgreSQL, Oracle, and a limited number of other database systems. When you create a materialized view, the data returned by its query is stored permanently in the database like a table; you can refresh the view to update the stored data if needed.

NOTE

To ensure data security and prevent users from seeing sensitive information, you must restrict access by setting account permissions in PostgreSQL and also define the view using a `security_barrier` attribute. Typically, a database administrator handles these tasks, but if you want to explore this issue further, read the PostgreSQL documentation on user roles at <https://www.postgresql.org/docs/current/sql-createrole.html>, on the GRANT command at <https://www.postgresql.org/docs/current/sql-grant.html>, and on `security_barrier` at <https://www.postgresql.org/docs/current/rules-privileges.html>.

Views are easy to create and maintain. Let's work through several examples to see how they work.

Creating and Querying Views

In this section, we'll return to the census estimates table `us_counties_pop_est_2019` you imported in Chapter 5. [Listing 17-1](#) creates a standard view that returns just the population of Nevada counties. The original table has sixteen columns; the view will return just four of them. This would be useful for making a subset of Nevada census data quickly accessible when we're referring to it often or using the data in an application.

```
| CREATE OR REPLACE VIEW nevada_counties_pop_2019 AS
 2 SELECT county_name,
        state_fips,
        county_fips,
        pop_est_2019
   FROM us_counties_pop_est_2019
 WHERE state_name = 'Nevada';
```

[Listing 17-1](#): Creating a view that displays Nevada 2019 counties

We define the view using the keywords `CREATE OR REPLACE VIEW` **1** followed by the view's name, `nevada_counties_pop_2019`, and then `AS`. (We can name the view any way we'd like; I prefer a name that's descriptive of the view's results.) Next, we use a standard SQL `SELECT` **2** to fetch the 2019 population estimate (the `pop_est_2019` column) for each Nevada county from the `us_counties_pop_est_2019` table.

Notice the `OR REPLACE` keywords after `CREATE`. These are optional and tell the database that if a view with this name already exists, then replace it with the new definition. It's helpful to include these keywords if you're iterating on creating a view and want to refine the query. There is one caveat: if you're replacing an existing view, the new query **2** must generate the same column names with the same data types and in the same order as the one it's replacing. You can add columns, but they must be placed at the end of the column list. If you try to do otherwise, the database will respond with an error message.

Run the code in [Listing 17-1](#) using pgAdmin. The database should respond with the message `CREATE VIEW`. To find the new view, in pgAdmin's object browser, right-click the `analysis` database and click **Refresh**. Choose **Schemas▶public▶Views** to see all views. When you right-click your new view and click **Properties**, you should see a more verbose version of the query (with

the table name prepended to each column name) on the Code tab in the dialog that opens. That's a handy way to inspect views you might find in a database.

NOTE

As with other database objects, you can delete a view using the `DROP` command. In this example, the syntax would be `DROP VIEW nevada_counties_pop_2019;`.

This type of view—one that isn't materialized—holds no data at this point; instead, the stored `SELECT` query it contains will run when you access the view from another query. For example, the code in [Listing 17-2](#) returns all columns in the view. As with a typical `SELECT` query, we can use `ORDER BY` to sort results, this time using the county's Federal Information Processing Standards (FIPS) code—the standard designator the US Census Bureau and other federal agencies use to specify each county and state. We also add a `LIMIT` clause to display just five rows.

```
SELECT *
FROM nevada_counties_pop_2019
ORDER BY county_fips
LIMIT 5;
```

[Listing 17-2](#): Querying the `nevada_counties_pop_2010` view

Aside from the five-row limit, the result should be the same as if you had run the `SELECT` query used to create the view in [Listing 17-1](#):

geo_name	state_fips	county_fips	pop_2010
Churchill County	32	001	24909
Clark County	32	003	2266715
Douglas County	32	005	48905
Elko County	32	007	52778
Esmeralda County	32	009	873

This simple example isn't useful unless quickly listing Nevada county population is a task you'll perform frequently. So, let's imagine a question data-minded analysts in a political research organization might ask often: what was

the percent change in population for each county in Nevada (or any other state) from 2010 to 2019?

We wrote a query to answer this question in Chapter 7, and though it wasn't onerous to create, it did require joining tables on two columns and using a percent change formula that involved rounding and type casting. To avoid repeating that work, we can create a view that stores a query similar to the one in Chapter 7 as a view, as shown in [Listing 17-3](#).

```
| CREATE OR REPLACE VIEW county_pop_change_2019_2010 AS
2  SELECT c2019.county_name,
   c2019.state_name,
   c2019.state_fips,
   c2019.county_fips,
   c2019.pop_est_2019 AS pop_2019,
   c2010.estimates_base_2010 AS pop_2010,
3  round( (c2019.pop_est_2019::numeric -
c2010.estimates_base_2010)
    / c2010.estimates_base_2010 * 100, 1 ) AS
   pct_change_2019_2010
4  FROM us_counties_pop_est_2019 AS c2019
      JOIN us_counties_pop_est_2010 AS c2010
    ON c2019.state_fips = c2010.state_fips
   AND c2019.county_fips = c2010.county_fips;
```

[Listing 17-3](#): Creating a view showing population change for US counties

We start the view definition with `CREATE OR REPLACE VIEW` **1**, followed by the name of the view and `AS`. The `SELECT` query **2** names columns from the census tables and includes a column definition with a percent change calculation **3** that you learned about in Chapter 6. Then we join the 2019 and 2010 census tables **4** using the state and county FIPS codes. Run the code, and the database should again respond with `CREATE VIEW`.

Now that we've created the view, we can use the code in [Listing 17-4](#) to run a simple query using the new view that retrieves data for Nevada counties.

```
SELECT county_name,
       state_name,
       pop_2019,
1  pct_change_2019_2010
FROM county_pop_change_2019_2010
```

```
? WHERE state_name = 'Nevada'  
    ORDER BY county_fips  
    LIMIT 5;
```

[Listing 17-4](#): Selecting columns from the county_pop_change_2019_2010 view

In [Listing 17-2](#), in the query that referenced our nevada_counties_pop_2019 view, we retrieved every column in the view by using the asterisk wildcard after SELECT. [Listing 17-4](#) shows that, as with a query on a table, we can name specific columns when querying a view. Here, we specify four of the county_pop_change_2019_2010 view's seven columns. One is

pct_change_2019_2010 1, which returns the result of the percent change calculation we're looking for. As you can see, it's much simpler to write the column name like this than the whole formula. We're also filtering the results using a WHERE clause 2, similar to how we'd filter any query.

After querying the four columns from the view, the results should look like this:

county_name	state_name	pop_2019	pct_change_2019_2010
Churchill County	Nevada	24909	0.1
Clark County	Nevada	2266715	16.2
Douglas County	Nevada	48905	4.1
Elko County	Nevada	52778	7.8
Esmeralda County	Nevada	873	11.4

Now we can revisit this view as often as we like to pull data for presentations or to answer questions about the percent change in population for any county in America from 2010 to 2019.

Looking at just these five rows, you can see a couple of interesting stories emerge: the continued rapid growth of Clark County, which includes the city of Las Vegas, as well as a strong percent increase in Esmeralda County, one of the smallest counties in the United States and home to several ghost towns.

Creating and Refreshing a Materialized View

A materialized view differs from a standard view in that upon its creation, the materialized view's stored query is executed, and the results it generates are saved in the database. In effect, this creates a new table. The view retains its stored query, so you can update the saved data by issuing a command to refresh

the view. A good use for materialized views is to preprocess complex queries that take a while to run and make those results available for faster querying.

Let's drop the nevada_counties_pop_2019 view and re-create it as a materialized view using the code in [Listing 17-5](#).

```
| DROP VIEW nevada_counties_pop_2019;  
? CREATE MATERIALIZED VIEW nevada_counties_pop_2019 AS  
    SELECT county_name,  
          state_fips,  
          county_fips,  
          pop_est_2019  
    FROM us_counties_pop_est_2019  
    WHERE state_name = 'Nevada';
```

[Listing 17-5](#): Creating a materialized view

First, we use a `DROP VIEW` 1 statement to remove the `nevada_counties_pop_2019` view from the database. Then, we run `CREATE MATERIALIZED VIEW` 2 to make the view. Notice that the syntax is the same as the one for making a standard view, except for the added `MATERIALIZED` keyword and the omission of `OR REPLACE`, which is not available in the materialized view syntax. After running the statement, the database should respond with the message `SELECT 17`, telling you that the view's query produced 17 rows to be stored in the view. We can now query this data as with a standard view.

Let's say that the population estimates stored in `us_counties_pop_est_2019` are revised. To update the data stored in the materialized view, we can use the `REFRESH` keyword, as in [Listing 17-6](#).

```
REFRESH MATERIALIZED VIEW nevada_counties_pop_2019;
```

[Listing 17-6](#): Refreshing a materialized view

Executing this statement reruns the query stored in the `nevada_counties_pop_2019` view; the server will respond with the message `REFRESH MATERIALIZED VIEW`. The view will now reflect any updates to the data referenced by the view's query. When you have a query that takes some time to run, you can save time by storing its results in a materialized view that's

refreshed periodically, letting users quickly access the stored data rather than run a lengthy query.

NOTE

Using REFRESH MATERIALIZED VIEW CONCURRENTLY will prevent locking out SELECT statements that are executed against the view during the refresh. See <https://www.postgresql.org/docs/current/sql-refreshmaterializedview.html> for details.

To delete a materialized view, we use a `DROP MATERIALIZED VIEW` statement. Also, note that materialized views appear in a different part of pgAdmin's object browser, under **Schemas▶public▶Materialized Views**.

Inserting, Updating, and Deleting Data Using a View

With nonmaterialized views, you can update or insert data in the underlying table being queried as long as the view meets certain conditions. One requirement is that the view must reference a single table or updatable view. If the view's query joins tables, as with the population change view we just built in the previous section, you can't perform inserts or updates to the original table directly. Also, the view's query can't contain `DISTINCT`, `WITH`, `GROUP BY`, or other clauses. (See a complete list of restrictions at <https://www.postgresql.org/docs/current/sql-createview.html>)

You already know how to directly insert and update data on a table, so why do it through a view? One reason is that a view is one way you can exercise control over which data a user can update. Let's work through an example to see how this works.

Creating a View of Employees

In the Chapter 7 lesson on joins, we created and filled the `departments` and `employees` tables with four rows about people and where they work (if you skipped that section, you can revisit [Listing 7-1](#)). Running a quick `SELECT * FROM employees ORDER BY emp_id;` query shows the table's contents, as you can see here:

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

Let's say we want to use a view to give users in the Tax Department (its dept_id is 1) the ability to add, remove, or update their employees' names without letting them change salary information or the data of employees in another department. To do this, we can set up a view using [Listing 17-7](#).

```
CREATE OR REPLACE VIEW employees_tax_dept WITH
(security_barrier) 1 AS
    SELECT emp_id,
           first_name,
           last_name,
           dept_id
      FROM employees
 2 WHERE dept_id = 1
 3 WITH LOCAL CHECK OPTION;
```

[Listing 17-7](#): Creating a view on the employees table

This view is similar to others we've created so far, but with a few additions. First, in the CREATE OR REPLACE VIEW statement, we add the keywords WITH (security_barrier) 1. This enables a level of database security to prevent a malicious user from getting around restrictions that the view places on rows and columns. (See <https://www.postgresql.org/docs/current/rules-privileges.html> for how someone might subvert a view if you omit this type of security.)

In the view's SELECT query, we pick the columns we want to show from the employees table and use WHERE to filter the results on dept_id = 1 2 to list only Tax Department staff. The view itself will restrict updates or deletes to rows matching the condition in the WHERE clause. Adding the keywords WITH LOCAL CHECK OPTION 3 restricts inserts as well, allowing users to add new Tax Department employees only (if the view definition omitted those keywords, you could use it to insert a row with a dept_id of 3, for example). The LOCAL CHECK OPTION also prevents a user from changing an employee's dept_id to a value other than 1.

Create the `employees_tax_dept` view by running the code in [Listing 17-7](#). Then run `SELECT * FROM employees_tax_dept ORDER BY emp_id;`, which should provide these two rows:

emp_id	first_name	last_name	dept_id
1	Julia	Reyes	1
2	Janet	King	1

The result shows the employees who work in the Tax Department; they're two of the four rows in the entire `employees` table.

Now, let's look at how inserts and updates work via this view.

Inserting Rows Using the `employees_tax_dept` View

We can use a view to insert or update data, but instead of using the table name in the `INSERT` or `UPDATE` statement, we substitute the view name. After we add or change data using a view, the change is applied to the underlying table, which in this case is `employees`. The view then reflects the change via the query it runs.

[Listing 17-8](#) shows two examples that attempt to add new employee records via the `employees_tax_dept` view. The first succeeds, but the second fails.

```
| 1 INSERT INTO employees_tax_dept (emp_id, first_name, last_name,
|   dept_id)
|   VALUES (5, 'Suzanne', 'Legere', 1);
|
| ? INSERT INTO employees_tax_dept (emp_id, first_name, last_name,
|   dept_id)
|   VALUES (6, 'Jamil', 'White', 2);
|
| } SELECT * FROM employees_tax_dept ORDER BY emp_id;
|
| ! SELECT * FROM employees ORDER BY emp_id;
```

[Listing 17-8](#): Successful and rejected inserts via the `employees_tax_dept` view

In the first `INSERT 1`, which uses the insert syntax you learned in Chapter 2, we supply the first and last names of Suzanne Legere plus her `emp_id` and `dept_id`. Because the new row will satisfy the `LOCAL CHECK` in the view—it

contains the same columns and `dept_id` is 1—the insert succeeds when it executes.

But when we run the second `INSERT 2` to add an employee named Jamil White using a `dept_id` of 2, the operation fails with the error message `new row violates check option for view "employees_tax_dept"`. The reason is that when we created the view, we used a `WHERE` clause to return only rows with `dept_id = 1`. The `dept_id` of 2 doesn't pass the `LOCAL CHECK`, so it's prevented from being inserted.

Run the `SELECT` statement 3 on the view to check that Suzanne Legere was successfully added:

emp_id	first_name	last_name	dept_id
1	Julia	Reyes	1
2	Janet	King	1
5	Suzanne	Legere	1

We also query the `employees` table 4 to see that, in fact, Suzanne Legere was added to the full table. The view queries the `employees` table each time we access it.

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
5	Suzanne	Legere		1

As you can see from the addition of Suzanne Legere, the data we add using a view is also added to the underlying table. However, because the view doesn't include the `salary` column, the value in her row is `NULL`. If you attempt to insert a salary value using this view, you would receive the error message `column "salary" of relation "employees_tax_dept" does not exist`. The reason is that even though the `salary` column exists in the underlying `employees` table, it's not referenced in the view. Again, this is one way to limit access to sensitive data. Check the links I provided in the note in the section “Using Views to Simplify Queries” to learn more about granting permissions to users and adding `WITH (security_barrier)` if you plan to take on database administrator responsibilities.

Updating Rows Using the employees_tax_dept View

The same restrictions on accessing data in an underlying table apply when we update data using the employees_tax_dept view. [Listing 17-9](#) shows a standard query to change the spelling of Suzanne's last name using UPDATE (as a person with more than one uppercase letter in their last name, I can confirm such corrections aren't unusual).

```
UPDATE employees_tax_dept
SET last_name = 'Le Gere'
WHERE emp_id = 5;

SELECT * FROM employees_tax_dept ORDER BY emp_id;
```

[Listing 17-9](#): Updating a row via the employees_tax_dept view

Run the code, and the result from the SELECT query should show the updated last name, which occurs in the underlying employees table:

emp_id	first_name	last_name	dept_id
1	Julia	Reyes	1
2	Janet	King	1
5	Suzanne	Le Gere	1

Suzanne's last name is now correctly spelled as Le Gere, not Legere.

However, if we try to update the name of an employee who's not in the Tax Department, the query fails just as it did when we tried to insert Jamil White in [Listing 17-8](#). Trying to use this view to update the salary of an employee—even one in the Tax Department—will also fail. If the view doesn't reference a column in the underlying table, you can't access that column through the view. Again, the fact that updates on views are restricted in this way offers ways to secure and hide certain pieces of data.

Deleting Rows Using the employees_tax_dept View

Now, let's explore how to delete rows using a view. The restrictions on which data you can affect apply here as well. For example, if Suzanne Le Gere gets a better offer from another firm and decides to leave, you could remove her from employees through the employees_tax_dept view. [Listing 17-10](#) shows the query in the standard DELETE syntax.

```
DELETE FROM employees_tax_dept  
WHERE emp_id = 5;
```

[Listing 17-10](#): Deleting a row via the `employees_tax_dept` view

Run the query, and PostgreSQL should respond with `DELETE 1`. However, when you try to delete a row for an employee in a department other than the Tax Department, PostgreSQL won't allow it and will report `DELETE 0`.

In summary, views not only give you control over access to data, but also give you shortcuts for working with data. Next, let's explore how to use functions to save keystrokes and time.

Creating Your Own Functions and Procedures

You've used functions throughout the book, such as to capitalize letters with `upper()` or add numbers with `sum()`. Behind these functions is a significant amount of (sometimes complex) programming that executes a series of actions and may, depending on the job of the function, return a response. We'll avoid complicated code here, but we'll build some basic functions that you can use as a launchpad for your own ideas. Even simple functions can help you avoid repeating code.

Much of the syntax in this section is specific to PostgreSQL, which supports both user-defined functions and *procedures* (the difference between the two is subtle, and I'll give examples of both). You can define functions and procedures using plain SQL, but you also can choose from other options. One is a PostgreSQL-specific *procedural language* called PL/pgSQL that adds features not found in standard SQL, such as logical control structures (`IF ... THEN ... ELSE`). Other options include PL/Python and PL/R for the Python and R programming languages.

Note that major database systems including Microsoft SQL Server, Oracle, and MySQL implement their own variations of functions and procedures. If you're using another database management system, this section will be useful for understanding concepts related to functions, but you'll need to check your database's documentation for specifics on its implementation of functions.

Creating the `percent_change()` Function

A function processes data and returns a value. As an example, let's write a function to simplify a staple of data analysis: calculating the percent change between two values. In Chapter 6, you learned that we express the percent change formula this way:

```
percent change = (New Number - Old Number) / Old Number
```

Rather than writing that formula each time we need it, we can create a function called `percent_change()` that takes the new and old numbers as inputs and returns the result rounded to a user-specified number of decimal places. Let's walk through the code in [Listing 17-11](#) to see how to declare a simple function that uses SQL.

```
| CREATE OR REPLACE FUNCTION
|   percent_change(new_value numeric,
|                   old_value numeric,
|                   decimal_places integer 3DEFAULT 1)
| RETURNS numeric AS
|   $ 'SELECT round(
|           ((new_value - old_value) / old_value) * 100,
|           decimal_places
|         )';
| LANGUAGE SQL
| IMMUTABLE
| RETURNS NULL ON NULL INPUT;
```

[Listing 17-11](#): Creating a `percent_change()` function

A lot is happening in this code, but it's not as complicated as it looks. We start with the command `CREATE OR REPLACE FUNCTION` 1. As with the syntax to create a view, the `OR REPLACE` keywords are optional. We then give the name of the function 2 and, in parentheses, a list of *arguments* that determine the function's inputs. Each argument will serve as an input to the function and gets a name and data type. For example, `new_value` and `old_value` are `numeric` and require that the user of the function supply input values matching that type, whereas `decimal_places` (which specifies the number of places to round results) is `integer`. For `decimal_places`, we specify 1 as the `DEFAULT 3` value —this makes the argument optional and, if it's omitted by the user, will set the argument to 1 by default.

We then use the keywords `RETURNS numeric AS` **4** to tell the function to return its calculation as type `numeric`. If this were a function to concatenate strings, we might return `text`.

Next, we write the meat of the function that performs the calculation. Inside single quotes, we place a `SELECT` query **5** that includes the percent change calculation nested inside a `round()` function. In the formula, we use the function's argument names instead of numbers.

We then supply a series of keywords that define the function's attributes and behavior. The `LANGUAGE` **6** keyword specifies that we've written this function using plain SQL as opposed to one of other languages PostgreSQL supports for creating functions. Next, the `IMMUTABLE` keyword **7** indicates that the function cannot modify the database and will always return the same result for a given set of arguments. The line `RETURNS NULL ON NULL INPUT` **8** guarantees that the function will supply a `NULL` response if any input that is not supplied by default is a `NULL`.

Run the code using pgAdmin to create the `percent_change()` function. The server should respond with the message `CREATE FUNCTION`.

Using the `percent_change()` Function

To test the new `percent_change()` function, run it by itself using `SELECT`, as shown in [Listing 17-12](#).

```
SELECT percent_change(110, 108, 2);
```

[Listing 17-12](#): Testing the `percent_change()` function

This example uses a value of 110 for the new number, 108 for the old number, and 2 as the desired number of decimal places to round the result.

Run the code; the result should look like this:

```
percent_change
-----
1.85
```

The result tells us there's a 1.85 percent increase between 108 and 110. Experiment with other numbers to see how the results change. Also, try

changing the `decimal_places` argument to values including 0, or omit it, to see how that affects the output. You should see results that have more or fewer numbers after the decimal point, based on your input.

We created this function to avoid writing the full percent change formula in queries. Let's use it to calculate percent change using a version of the census estimates population change query we wrote in Chapter 7, as shown in [Listing 17-13](#).

```
SELECT c2019.county_name,
       c2019.state_name,
       c2019.pop_est_2019 AS pop_2019,
       1 percent_change(c2019.pop_est_2019,
                         c2010.estimates_base_2010) AS
       pct_chg_func,
       2 round( (c2019.pop_est_2019::numeric -
c2010.estimates_base_2010)
           / c2010.estimates_base_2010 * 100, 1 ) AS
       pct_change_formula
FROM us_counties_pop_est_2019 AS c2019
JOIN us_counties_pop_est_2010 AS c2010
ON c2019.state_fips = c2010.state_fips
   AND c2019.county_fips = c2010.county_fips
ORDER BY pct_chg_func DESC
LIMIT 5;
```

[Listing 17-13](#): Testing `percent_change()` on census data

[Listing 17-13](#) modifies the original query from Chapter 7 to add the `percent_change()` function **1** as a column in `SELECT`. We also include the explicit percent change formula **2** so we can compare results. As inputs, we use the 2019 population estimate column (`c2019.pop_est_2019`) as the new number and the 2010 estimates base as the old (`c2010.estimates_base_2010`).

The query results should display the five counties with the greatest percent change in population, and the results from the function should match the results from the formula entered directly into the query. Note that each value in the `pct_chg_func` column has one decimal place, the function's default value, because we didn't provide the optional third argument. Here's the result with both the function and the formula:

county_name	state_name	pop_2019	pct_chg_func
pct_chg_formula			
<hr/>			
McKenzie County	North Dakota	15024	136.3
136.3			
Loving County	Texas	169	106.1
106.1			
Williams County	North Dakota	37589	67.8
67.8			
Hays County	Texas	230191	46.5
46.5			
Wasatch County	Utah	34091	44.9
44.9			

Now that we know the function works as intended, we can use `percent_change()` any time we need to solve that calculation—and that's much faster than writing out the formula!

Updating Data with a Procedure

As implemented in PostgreSQL, a *procedure* is a close relative of a function, albeit with some significant differences. Both procedures and functions can perform data operations that don't return a value, such as an update.

Procedures, on the other hand, don't have a clause to return a value, while functions do. Also, procedures can incorporate the transaction commands we covered in Chapter 10 such as `COMMIT` and `ROLLBACK`, and functions cannot. Many database managers implement procedures, which are sometimes referred to as *stored procedures*. PostgreSQL added procedures as of version 11 and are part of the SQL standard, though PostgreSQL syntax is not fully compatible.

We can simplify routine updates to data using procedures. In this section, we'll write a procedure that updates a record of the correct number of personal days off a teacher gets (in addition to vacation days) based on the time elapsed since their hire date.

For this exercise, we'll return to the `teachers` table from the first lesson in Chapter 2. If you skipped “Creating a Table” in that chapter, create the `teachers` table and insert the data now using the example code in Listings 2-2 and 2-3.

Let's add a column to `teachers` to hold the teachers' personal days using the code in [Listing 17-14](#). The new column will be empty until we fill it later using a

procedure.

```
ALTER TABLE teachers ADD COLUMN personal_days integer;

SELECT first_name,
       last_name,
       hire_date,
       personal_days
FROM teachers;
```

[Listing 17-14](#): Adding a column to the teachers table and seeing the data

[Listing 17-14](#) updates the teachers table using `ALTER` and adds the `personal_days` column using the keywords `ADD COLUMN`. We then run the `SELECT` statement to view the data, in which we also include the names and hire dates of each teacher. When both queries finish, you should see the following six rows:

first_name	last_name	hire_date	personal_days
Janet	Smith	2011-10-30	
Lee	Reynolds	1993-05-22	
Samuel	Cole	2005-08-01	
Samantha	Bush	2011-10-30	
Betty	Diaz	2005-08-30	
Kathleen	Roush	2010-10-22	

The `personal_days` column contains only `NULL` values because we haven't inserted anything yet.

Now, let's create a procedure called `update_personal_days()` that populates the `personal_days` column with their earned personal days (in addition to vacation days). We'll use the following criteria:

Less than 10 years since hire: 3 personal days

10 to less than 15 years since hire: 4 personal days

15 to less than 20 years since hire: 5 personal days

20 years to less than 25 years since hire: 6 personal days

25 years or more since hire: 7 personal days

The code in [Listing 17-15](#) creates a procedure. This time, instead of using plain SQL, we'll incorporate elements of the PL/pgSQL procedural language, which is an additional language PostgreSQL supports for writing functions. Let's walk through some differences.

```
CREATE OR REPLACE PROCEDURE update_personal_days()
AS $$
1 BEGIN
    UPDATE teachers
    SET personal_days =
        3 CASE WHEN (now() - hire_date) >= '10 years'::interval
              AND (now() - hire_date) < '15
years'::interval THEN 4
              WHEN (now() - hire_date) >= '15 years'::interval
              AND (now() - hire_date) < '20
years'::interval THEN 5
              WHEN (now() - hire_date) >= '20 years'::interval
              AND (now() - hire_date) < '25
years'::interval THEN 6
              WHEN (now() - hire_date) >= '25 years'::interval
THEN 7
              ELSE 3
        END;
4 RAISE NOTICE 'personal_days updated!';
END;
$$
LANGUAGE plpgsql;
```

[Listing 17-15](#): Creating an `update_personal_days()` function

We begin with `CREATE OR REPLACE PROCEDURE` and give the procedure a name. This time, we provide no arguments because no user input is required—the procedure operates on predetermined columns with set values for calculating intervals.

Often, when writing PL/pgSQL-based functions, the PostgreSQL convention is to use the non-ANSI SQL standard dollar-quote (`$$`) to mark the start **1** and end **5** of the string that contains all the function's commands. (As with the `percent_change()` SQL function earlier, you could use single quote marks to enclose the string, but then any single quotes in the string would need to be doubled, and that not only looks messy but can be confusing.) So, everything

between the pair of \$\$ is the code that does the work. You can also add some text between the dollar signs, like \$namestring\$, to create a unique pair of beginning and ending quotes. This is useful, for example, if you need to quote a query inside the function.

Right after the first \$\$ we start a BEGIN ... END; **2** block. This is a PL/pgSQL convention that delineates the start and end of a section of code within a function or procedure; as with dollar quotes, it is possible to nest one BEGIN ... END; inside another to facilitate logical groupings of code. Inside that block, we place an UPDATE statement that uses a CASE statement **3** to determine the number of days each teacher gets. We subtract the hire_date from the current date, which is retrieved from the server by the now() function. Depending on which range now() - hire_date falls into, the CASE statement returns the number of personal days corresponding to the range. We use the PL/pgSQL keywords RAISE NOTICE **4** to display a message that the procedure is done. Finally, we use the LANGUAGE keyword **6** so the database knows to interpret what we've written according to the syntax specific to PL/pgSQL.

Run the code in [Listing 17-15](#) to create the update_personal_days() procedure. To invoke the procedure, we use the CALL command, which is part of the ANSI SQL standard:

```
CALL update_personal_days();
```

When the procedure runs, the server responds with the notice it raises, which is personal_days updated!.

When you rerun the SELECT statement in [Listing 17-14](#), you should see that each row of the personal_days column is filled with the appropriate values. Note that results will vary depending on when you run this function, because calculations using now() change as time passes.

first_name	last_name	hire_date	personal_days
Janet	Smith	2011-10-30	3
Lee	Reynolds	1993-05-22	7
Samuel	Cole	2005-08-01	5
Samantha	Bush	2011-10-30	3
Betty	Diaz	2005-08-30	5
Kathleen	Roush	2010-10-22	4

You could use the `update_personal_days()` function to regularly update data manually after performing certain tasks, or you could use a task scheduler such as pgAgent (a separate open source tool) to run it automatically. You can learn about pgAgent and other tools in “PostgreSQL Utilities, Tools, and Extensions” in the appendix.

Using the Python Language in a Function

Previously, I mentioned that PL/pgSQL is the default procedural language within PostgreSQL, but the database also supports creating functions using open source languages, such as Python and R. This support allows you to take advantage of features and modules from those languages within functions you create. For example, with Python, you can use the `pandas` library for analysis. The documentation at <https://www.postgresql.org/docs/current/server-programming.html> provides a comprehensive review of the languages included with PostgreSQL, but here I’ll show you a simple function using Python.

To enable PL/Python, you must create the extension using the code in [Listing 17-16](#).

```
CREATE EXTENSION plpython3u;
```

[Listing 17-16](#): Enabling the PL/Python procedural language

If you get an error, such as `image not found`, that means the PL/Python extension is not installed on your system. Depending on the operating system, installation of PL/Python typically requires installation of Python and additional configuration beyond the basic PostgreSQL install. For this, refer to the installation instructions for your operating system in Chapter 1.

After enabling the extension, we can create a function using syntax similar to the examples you’ve tried so far, but using Python for the body of the function. [Listing 17-17](#) shows how to use PL/Python to create a function called `trim_county()` that removes the word *County* from the end of a string. We’ll use this function to clean up names of counties in the census data.

```
CREATE OR REPLACE FUNCTION trim_county(input_string text)
| RETURNS text AS $$
|     import re2
|     3 cleaned = re.sub(r' County', '', input_string)
```

```
    return cleaned
$$
| LANGUAGE plpython3u;
```

[Listing 17-17](#): Using PL/Python to create the `trim_county()` function

The structure should look familiar. After naming the function and its text input, we use the RETURNS keyword **1** to specify that the function will send text back. After the opening \$\$ quotes, we get straight to the Python code, starting with a statement to import the Python regular expressions module, `re` **2**. Even if you don't know much about Python, you can probably deduce that the next two lines of code **3** set a variable called `cleaned` to the results of a Python regular expression function called `sub()`. That function looks for a space followed by the word *County* in the `input_string` passed into the function and substitutes an empty string, which is denoted by two apostrophes. Then the function returns the content of the variable `cleaned`. To end, we specify LANGUAGE `plpython3u` **4** to note we're writing the function with PL/Python.

Run the code to create the function, and then execute the SELECT statement in [*Listing 17-18*](#) to see it in action.

```
SELECT county_name,
       trim_county(county_name)
  FROM us_counties_pop_est_2019
 ORDER BY state_fips, county_fips
   LIMIT 5;
```

[Listing 17-18](#): Testing the `trim_county()` function

We use the `county_name` column in the `us_counties_pop_est_2019` table as input to `trim_county()`. That should return these results:

county_name	trim_county
Autauga County	Autauga
Baldwin County	Baldwin
Barbour County	Barbour
Bibb County	Bibb
Blount County	Blount

As you can see, the `trim_county()` function evaluated each value in the `county_name` column and removed a space and the word *County* when present. Although this is a trivial example, it shows how easy it is to use Python—or one of the other supported procedural languages—inside a function.

Next, you'll learn how to use triggers to automate your database.

Automating Database Actions with Triggers

A database *trigger* executes a function whenever a specified event, such as an `INSERT`, `UPDATE`, or `DELETE`, occurs on a table or a view. You can set a trigger to fire before, after, or instead of the event, and you can also set it to fire once for each row affected by the event or just once per operation. For example, let's say you delete 20 rows from a table. You could set the trigger to fire once for each of the 20 rows deleted or just one time.

We'll work through two examples. The first example keeps a log of changes made to grades at a school. The second automatically classifies temperatures each time we collect a reading.

Logging Grade Updates to a Table

Let's say we want to automatically track changes made to a student grades table in our school's database. Every time a row is updated, we want to record the old and new grade plus the time the change occurred (search online for *David Lightman and grades* and you'll see why this might be worth tracking). To handle this task automatically, we'll need three items:

A `grades_history` table to record the changes to grades in a `grades` table

A trigger to run a function every time a change occurs in the `grades` table, which we'll name `grades_update`

The function the trigger will execute, which we'll call
`record_if_grade_changed()`

Creating Tables to Track Grades and Updates

Let's start by making the tables we need. [Listing 17-19](#) includes the code to first create and fill `grades` and then create `grades_history`.

```
| CREATE TABLE grades (
    student_id bigint,
    course_id bigint,
    course text NOT NULL,
    grade text NOT NULL,
PRIMARY KEY (student_id, course_id)
);

? INSERT INTO grades
VALUES
    (1, 1, 'Biology 2', 'F'),
    (1, 2, 'English 11B', 'D'),
    (1, 3, 'World History 11B', 'C'),
    (1, 4, 'Trig 2', 'B');

3 CREATE TABLE grades_history (
    student_id bigint NOT NULL,
    course_id bigint NOT NULL,
    change_time timestamp with time zone NOT NULL,
    course text NOT NULL,
    old_grade text NOT NULL,
    new_grade text NOT NULL,
PRIMARY KEY (student_id, course_id, change_time)
);
```

[Listing 17-19](#): Creating the `grades` and `grades_history` tables

These commands are straightforward. We use `CREATE` to make a `grades` table **1** and add four rows using `INSERT` **2**, where each row represents a student's grade in a class. Then we use `CREATE TABLE` to make the `grades_history` table **3** to hold the data we log each time an existing grade is altered. The `grades_history` table has columns for the new grade, old grade, and the time of the change. Run the code to create the tables and fill the `grades` table. We insert no data into `grades_history` here because the trigger process will handle that task.

Creating the Function and Trigger

Next, let's write the `record_if_grade_changed()` function that the trigger will execute (note that the PostgreSQL documentation refers to such functions as *trigger procedures*). We must write the function before naming it in the trigger. Let's go through the code in [Listing 17-20](#).

```
CREATE OR REPLACE FUNCTION record_if_grade_changed()
  1 RETURNS trigger AS
$$
BEGIN
  2 IF NEW.grade <> OLD.grade THEN
    INSERT INTO grades_history (
      student_id,
      course_id,
      change_time,
      course,
      old_grade,
      new_grade)
    VALUES
      (OLD.student_id,
       OLD.course_id,
       now(),
       OLD.course,
       3 OLD.grade,
       4 NEW.grade);
  END IF;
  5 RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

[Listing 17-20](#): Creating the record_if_grade_changed() function

The `record_if_grade_changed()` function follows the pattern of earlier examples but with differences specific to working with triggers. First, we specify `RETURNS trigger` 1 instead of a data type. We use dollar-quotes to delineate the code portion of the function, and because

`record_if_grade_changed()` is a PL/pgSQL function, we also place the code to execute inside a `BEGIN ... END;` block. Next, we start the procedure using an `IF ... THEN` statement 2, which is one of the control structures PL/pgSQL provides. We use it here to run the `INSERT` statement only if the updated grade is different from the old grade, which we check using the `<>` operator.

When a change occurs to the `grades` table, the trigger (which we'll create next) will execute. For each row that's changed, the trigger will pass two collections of data into `record_if_grade_changed()`. The first is the row values *before* they were changed, noted with the prefix `OLD`. The second is the row values *after* they were changed, noted with the prefix `NEW`. The function can access the original row values and the updated row values, which it will use for a comparison. If

the `IF ... THEN` statement evaluates as `true`, indicating that the old and new grade values are different, we use `INSERT` to add a row to `grades_history` that contains both `OLD.grade` 3 and `NEW.grade` 4. Finally, we include a `RETURN` statement 5 with a value of `NULL`; the trigger procedure performs a database `INSERT`, so we do not need a value returned.

NOTE

Sometimes a `RETURN` value is ignored; the PostgreSQL documentation at <https://www.postgresql.org/docs/current/plpgsql-trigger.html> details the scenarios in which this occurs.

Run the code in [Listing 17-20](#) to create the function. Then, add the `grades_update` trigger to the `grades` table using [Listing 17-21](#).

```
| CREATE TRIGGER grades_update
  2 AFTER UPDATE
    ON grades
  3 FOR EACH ROW
  4 EXECUTE PROCEDURE record_if_grade_changed();
```

[Listing 17-21](#): Creating the `grades_update` trigger

In PostgreSQL, the syntax for creating a trigger follows the ANSI SQL standard (although not all aspects of the standard are supported, per the documentation at <https://www.postgresql.org/docs/current/sql-createtrigger.html>). The code begins with a `CREATE TRIGGER` 1 statement, followed by clauses that control when the trigger runs and how it behaves. We use `AFTER UPDATE` 2 to specify that we want the trigger to fire after the update occurs on the `grades` row. We could also use the `BEFORE` or `INSTEAD OF` keywords depending on the need.

We write `FOR EACH ROW` 3 to tell the trigger to execute the procedure once for each row updated in the table. For example, if someone runs an update that affects three rows, the procedure will run three times. The alternate (and default) is `FOR EACH STATEMENT`, which runs the procedure once. If we didn't care about capturing changes to each row and simply wanted to record that

grades were changed at a certain time, we could use that option. Finally, we use EXECUTE PROCEDURE 4 to name record_if_grade_changed() as the function the trigger should run.

Create the trigger by running the code in [Listing 17-21](#) in pgAdmin. The database should respond with the message CREATE TRIGGER.

Testing the Trigger

Now that we've created the trigger and the function, it should run when data in the grades table changes; let's see what the process does. First, let's check the current status of our data. When you run `SELECT * FROM grades_history;`, you'll see that the table is empty because we haven't made any changes to the grades table yet and there's nothing to track. Next, when you run `SELECT * FROM grades ORDER BY student_id, course_id;`, you should see the grade data that you inserted in [Listing 17-19](#), as shown here:

student_id	course_id	course	grade
1	1	Biology 2	F
1	2	English 11B	D
1	3	World History 11B	C
1	4	Trig 2	B

That Biology 2 grade doesn't look very good. Let's update it using the code in [Listing 17-22](#).

```
UPDATE grades
SET grade = 'C'
WHERE student_id = 1 AND course_id = 1;
```

[Listing 17-22](#): Testing the grades_update trigger

When you run the UPDATE, pgAdmin doesn't display anything to let you know that the trigger executed in the background. It just reports UPDATE 1, meaning a row was updated. But our trigger did run, which we can confirm by examining columns in grades_history using this SELECT query:

```
SELECT student_id,
       change_time,
       course,
       old_grade,
```

```
    new_grade  
FROM grades_history;
```

When you run this query, you should see that the `grades_history` table, which contains all changes to grades, now has one row:

student_id	old_grade	new_grade	change_time	course	
1	C	F	2023-09-01 15:50:43.291164-04	Biology 2	

This row displays the old Biology 2 grade of F, the new value C, and `change_time`, showing the time of update (your result should reflect your date and time). Note that the addition of this row to `grades_history` happened in the background without the knowledge of the person making the update. But the `UPDATE` event on the table caused the trigger to fire, which executed the `record_if_grade_changed()` function.

If you've ever used a content management system, such as WordPress or Drupal, this sort of revision tracking might be familiar. It provides a helpful record of changes made to content for reference, auditing, and, unfortunately, occasional finger-pointing. Regardless, the ability to trigger actions on a database automatically gives you more control over your data.

Automatically Classifying Temperatures

In Chapter 13, we used the SQL `CASE` statement to reclassify temperature readings into descriptive categories. The `CASE` statement is also part of the PL/pgSQL procedural language, and we can use its capability to assign values to variables to automatically store those category names in a table each time we add a temperature reading. If we're routinely collecting temperature readings, using this technique to automate the classification spares us from having to handle the task manually.

We'll follow the same steps we used for logging the grade changes: we first create a function to classify the temperatures and then create a trigger to run the function each time the table is updated. Use [Listing 17-23](#) to create a `temperature_test` table for the exercise.

```
CREATE TABLE temperature_test (
    station_name text,
    observation_date date,
    max_temp integer,
    min_temp integer,
    max_temp_group text,
PRIMARY KEY (station_name, observation_date)
);
```

[Listing 17-23](#): Creating a temperature_test table

The temperature_test table contains columns to hold the name of the station and date of the temperature observation. Let's imagine that we have some process to insert a row once a day that provides the maximum and minimum temperature for that location, and we need to fill the max_temp_group column with a descriptive classification of the day's high reading to provide text to a weather forecast we're distributing.

To do this, we first make a function called classify_max_temp(), as shown in [Listing 17-24](#).

```
CREATE OR REPLACE FUNCTION classify_max_temp()
RETURNS trigger AS
$$
BEGIN
    CASE
        WHEN NEW.max_temp >= 90 THEN
            NEW.max_temp_group := 'Hot';
        WHEN NEW.max_temp >= 70 AND NEW.max_temp < 90 THEN
            NEW.max_temp_group := 'Warm';
        WHEN NEW.max_temp >= 50 AND NEW.max_temp < 70 THEN
            NEW.max_temp_group := 'Pleasant';
        WHEN NEW.max_temp >= 33 AND NEW.max_temp < 50 THEN
            NEW.max_temp_group := 'Cold';
        WHEN NEW.max_temp >= 20 AND NEW.max_temp < 33 THEN
            NEW.max_temp_group := 'Frigid';
        WHEN NEW.max_temp < 20 THEN
            NEW.max_temp_group := 'Inhumane';
        ELSE NEW.max_temp_group := 'No reading';
    END CASE;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

[*Listing 17-24*](#): Creating the `classify_max_temp()` function

By now, these functions should look familiar. What's new here is the PL/pgSQL version of the CASE syntax **1**, which differs slightly from the SQL syntax. The PL/pgSQL syntax includes a semicolon after each `WHEN ... THEN` clause. Also new is the *assignment operator* `:=`, which we use to assign the descriptive name to the `NEW.max_temp_group` column based on the outcome of the CASE function. For example, the statement `NEW.max_temp_group := 'Cold'` assigns the string 'Cold' to `NEW.max_temp_group` when the temperature value is greater than or equal to 33 degrees but less than 50 degrees Fahrenheit. When the function returns the `NEW` row to be inserted in the table, it will include the string value `Cold`. Run the code to create the function.

Next, using the code in [*Listing 17-25*](#), create a trigger to execute the function each time a row is added to `temperature_test`.

```
CREATE TRIGGER temperature_insert
1 BEFORE INSERT
    ON temperature_test
2 FOR EACH ROW
3 EXECUTE PROCEDURE classify_max_temp();
```

[*Listing 17-25*](#): Creating the `temperature_insert` trigger

In this example, we classify `max_temp` and create a value for `max_temp_group` prior to inserting the row into the table. Doing so is more efficient than performing a separate update after the row is inserted. To specify that behavior, we set the `temperature_insert` trigger to fire BEFORE INSERT **1**.

We also want the trigger to fire FOR EACH ROW **2** because we want each `max_temp` recorded in the table to get a descriptive classification. The final EXECUTE PROCEDURE statement names the `classify_max_temp()` function **3** we just created. Run the CREATE TRIGGER statement in pgAdmin, and then test the setup using [*Listing 17-26*](#).

```
INSERT INTO temperature_test
VALUES
    ('North Station', '1/19/2023', 10, -3),
    ('North Station', '3/20/2023', 28, 19),
    ('North Station', '5/2/2023', 65, 42),
    ('North Station', '8/9/2023', 93, 74),
```

```
('North Station', '12/14/2023', NULL, NULL);  
SELECT * FROM temperature_test ORDER BY observation_date;
```

[Listing 17-26](#): Inserting rows to test the temperature_insert trigger

Here we insert five rows into `temperature_test`, and we expect the `temperature_insert` trigger to fire for each row—and it does! The `SELECT` statement in the listing should display these results:

station_name	observation_date	max_temp	min_temp	max_temp_group
North Station	2023-01-19	10	-3	Inhumane
North Station	2023-03-20	28	19	Frigid
North Station	2023-05-02	65	42	Pleasant
North Station	2023-08-09	93	74	Hot
North Station	2023-12-14			No reading

Thanks to the trigger and function, each `max_temp` inserted automatically receives the appropriate classification in the `max_temp_group` column—including the instance where we had no reading for that value. Note that the trigger’s update of the column will override any user-supplied values during insert.

This temperature example and the earlier grade-change auditing example are rudimentary, but they give you a glimpse of how useful triggers and functions can be in simplifying data maintenance.

Wrapping Up

Although the techniques you learned in this chapter begin to merge with those of a database administrator, you can apply the concepts to reduce the amount of time you spend repeating certain tasks. I hope these approaches will help you free up more time to find interesting stories in your data.

This chapter concludes our discussion of analysis techniques and the SQL language. The next two chapters offer workflow tips to help you increase your command of PostgreSQL. They include how to connect to a database and run

queries from your computer's command line and how to maintain your database.

TRY IT YOURSELF

Review the concepts in the chapter with these exercises:

Create a materialized view that displays the number of New York City taxi trips per hour. Use the taxi data from Chapter 12 and the query in [Listing 12-8](#). How do you refresh the view if you need to?

In Chapter 11, you learned how to calculate a rate per thousand. Turn that formula into a `rate_per_thousand()` function that takes three arguments to calculate the result: `observed_number`, `base_number`, and `decimal_places`.

In Chapter 10, you worked with the `meat_poultry_egg_establishments` table that listed food processing facilities. Write a trigger that automatically adds an inspection deadline timestamp six months in the future whenever you insert a new facility into the table. Use the `inspection_deadline` column added in [Listing 10-19](#). You should be able to describe the steps needed to implement a trigger and how the steps relate to each other.

18

USING POSTGRESQL FROM THE COMMAND LINE



In this chapter, you'll learn how to work with PostgreSQL from the *command line*, a text-based interface where you enter names of programs or other commands to perform tasks, such as editing files or listing the contents of a file directory.

The command line—also called a *command line interface*, a *console*, a *shell*, or the *terminal*—existed long before computers had a graphical user interface (GUI) with menus, icons, and buttons you could click for navigation. Back in college, to edit a file, I had to enter commands into a terminal connected to an IBM mainframe computer. Working that way felt mysterious, as though I'd attained new powers—and I had! Today, even in a GUI world, familiarity with the command line is essential for a programmer moving toward expert-level skills. Perhaps that's why when a movie wants to convey that a character really knows what they're doing on a computer, they're shown typing cryptic, text-only commands.

As we learn about this text-only world, pay attention to these advantages of mastering working from the command line instead of a GUI, such as pgAdmin:

You can often work faster by entering short commands instead of clicking through layers of menu items.

You gain access to functions that only the command line provides.

If command line access is all you have to work with (for example, when you've connected to a remote computer), you can still get work done.

We'll use `psql`, a command-line tool included with PostgreSQL that lets you run queries, manage database objects, and interact with the computer's operating system via text command. You'll learn how to set up and access your computer's command line and then launch `psql`. Along the way, we'll cover general command line syntax and additional commands for database tasks. Patience is key: even experienced experts often resort to documentation to recall the available command line options.

Setting Up the Command Line for `psql`

To start, we'll access the command line on our operating system and, if needed, set an environment variable called `PATH` that tells our system where to find `psql`. *Environment variables* hold parameters that specify system or application configurations, such as where to store temporary files; they also allow you to enable or disable options. The `PATH` environment variable stores the names of one or more directories containing executable programs, and in this instance will tell the command line interface the location of `psql`, avoiding the hassle of having to enter its full directory path each time you launch it.

Windows `psql` Setup

On Windows, you'll run `psql` within the *Command Prompt*, the application that provides that system's command line interface. Before we do that, we need to tell Command Prompt where to find `psql.exe`—the full name of the `psql` application on Windows.

Adding `psql` and Utilities to the Windows PATH

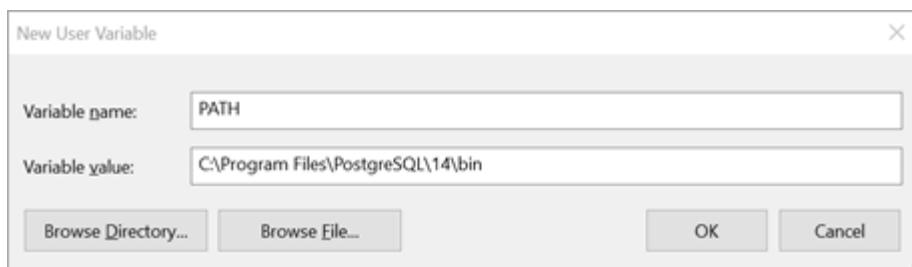
The following steps assume that you installed PostgreSQL according to the instructions described in “Windows Installation” in Chapter 1. (If you installed PostgreSQL another way, use the Windows File Explorer to search your C: drive to find the directory that holds `psql.exe`, and then replace *C:\Program Files\PostgreSQL\x\bin* in the following steps with your own path.)

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

Inside 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 should appear.

In the System Properties dialog, on the Advanced tab, click **Environment Variables**. The dialog that opens should have two sections: User variables and System variables. In the User 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 User variables section, click **New** to open a New User Variable dialog, shown in [Figure 18-1](#).



[Figure 18-1: Creating a new `PATH` environment variable in Windows 10](#)

In the Variable name box, enter **PATH**. In the Variable value box, enter **C:\Program Files\PostgreSQL\x\bin**, where *x* is the version of PostgreSQL you're using. (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** in all dialogs to close them.

If you do see an existing `PATH` variable in the User variables section, highlight it and click **Edit**. In the list of variables that displays, click **New** and enter **C:\Program Files\PostgreSQL\x\bin**, where *x* is the version of PostgreSQL you're using. (Instead of typing, you can click **Browse Directory** and navigate to the directory in the Browse For Folder dialog.) The result should look like the highlighted line in [Figure 18-2](#). When you're finished, click **OK** in all dialogs to close them.

Now when you launch Command Prompt, the `PATH` should include the directory. Note that any time you make changes to the `PATH`, you must close and reopen Command Prompt for the changes to take effect. Next, let's set up Command Prompt.

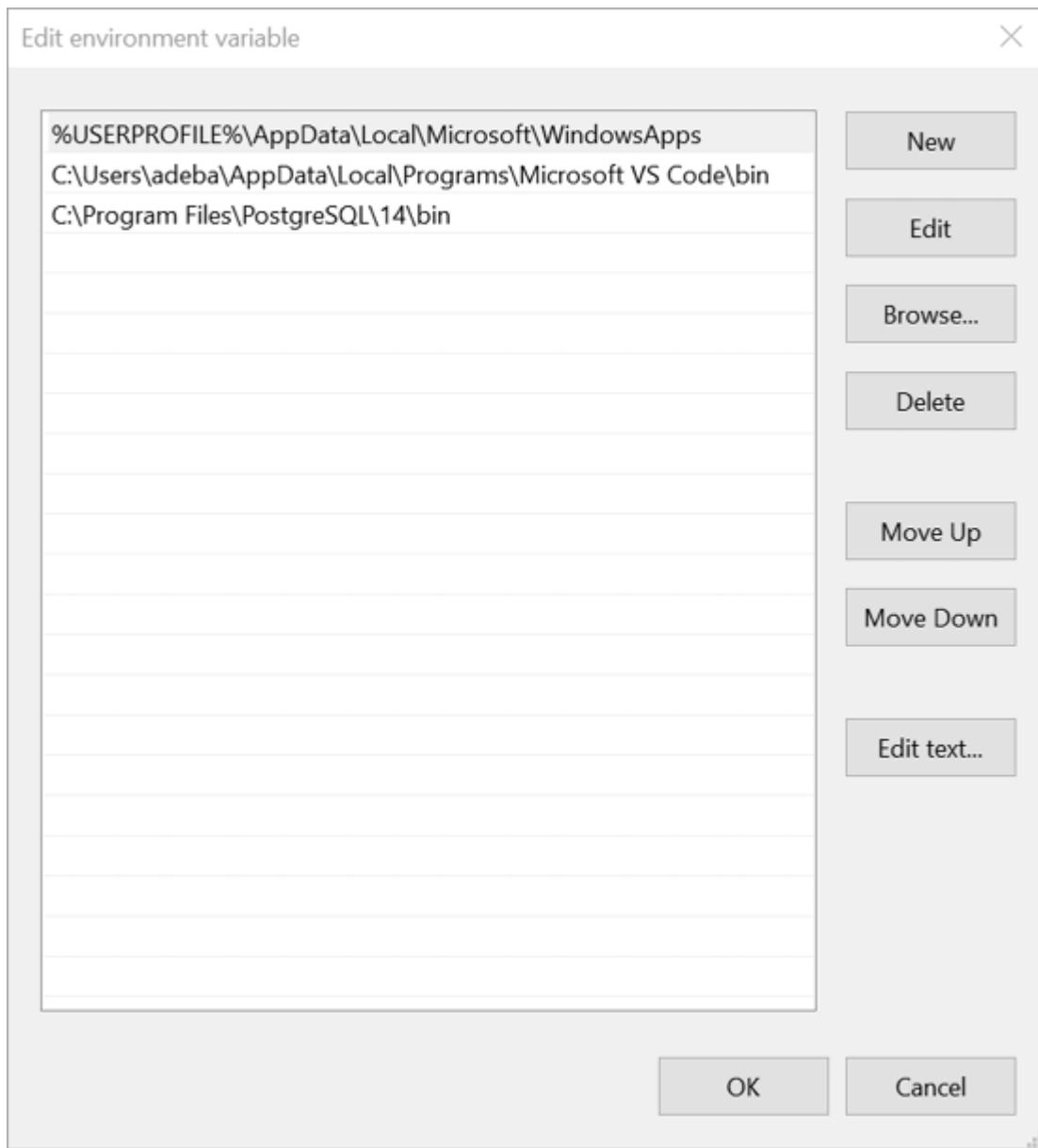


Figure 18-2: Editing existing PATH environment variables in Windows 10

Launching and Configuring Windows Command Prompt

Command Prompt is an executable file named `cmd.exe`. To launch it, select **Start▶Windows System▶Command Prompt** or enter **cmd** into the search bar. When the application opens, you should see a window with a black background that displays version and copyright information along with a prompt showing your current directory. On my Windows 10 system, Command Prompt opens to my default user directory and displays `C:\Users\adeba>`, as shown in [Figure 18-3](#).

This line is known as the *prompt*, and it shows the current working directory. For me, this is my C: drive, which is typically the main hard drive on a Windows system, and the \Users\adeba directory on that drive. The greater-than sign > indicates the area where you enter your commands.

NOTE

For fast access to Command Prompt, you can add it to your Windows taskbar. When Command Prompt is running, right-click its icon on the taskbar and then click Pin to taskbar.

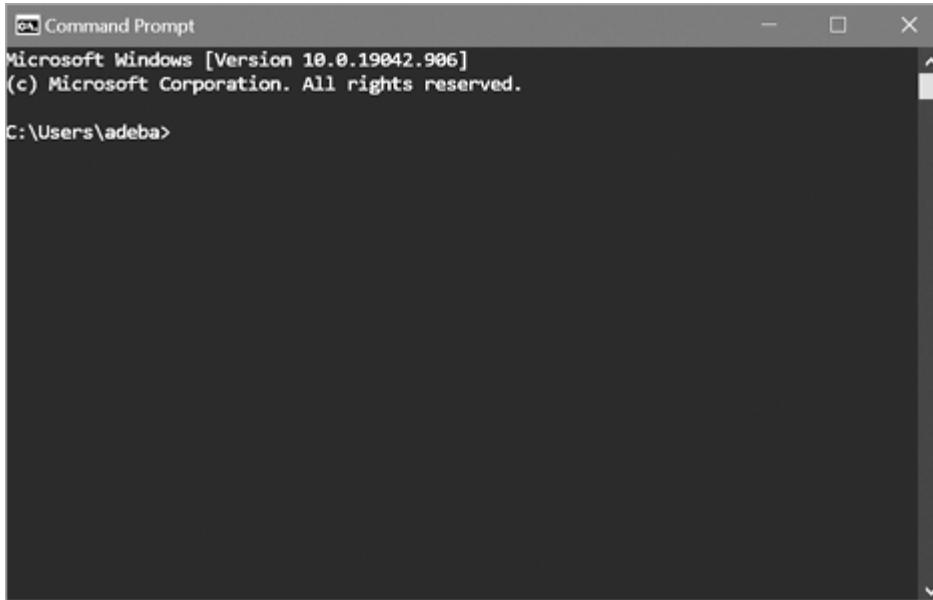


Figure 18-3: My Command Prompt in Windows 10

You can customize the font and colors plus access other settings by clicking the Command Prompt icon at the left of its window bar and selecting **Properties** from the menu. To make Command Prompt more suited for query output, I recommend setting the window size (on the Layout tab) to a width of at least 80 and a height of 25. For the font, the official PostgreSQL documentation recommends using Lucida Console to properly display all the characters.

Entering Instructions on Windows Command Prompt

Now you're ready to enter instructions in Command Prompt. Enter **help** at the prompt, and press ENTER on your keyboard to see a list of available Windows

system commands. You can view information about a command by including its name after `help`. For example, enter `help time` to display information about using the `time` command to set or view the system time.

Exploring the full workings of Command Prompt is an enormous topic beyond the scope of this book; however, I encourage you to try some of the commands in [Table 18-1](#), which contains useful frequently used commands that aren't actually necessary for the exercises in this chapter.

Table 18-1: Useful Windows Commands

Command	Function	Example	Action
cd	Change directory	<code>cd C:\my-stuff</code>	Changes to the <i>my-stuff</i> directory on the C: drive
copy	Copy a file	<code>copy C:\my-stuff\song.mp3 C:\Music\song_favorite.mp3</code>	Copies the <i>song.mp3</i> file from <i>my-stuff</i> to a new file called <i>song_favorite.mp3</i> in the <i>Music</i> directory
del	Delete	<code>del *.jpg</code>	Deletes all files with a <i>.jpg</i> extension in the current directory (asterisk wildcard)
dir	List directory contents	<code>dir /p</code>	Shows directory contents one screen at a time (using the <i>/p</i> option)
findstr	Find strings in text files matching a regular expression	<code>findstr "peach" *.txt</code>	Searches for the text <i>peach</i> in all <i>.txt</i> files in the current directory
mkdir	Make a new directory	<code>makedir C:\my-stuff\Salad</code>	Creates a <i>Salad</i> directory inside the <i>my-stuff</i> directory
move	Move a file	<code>move C:\my-stuff\song.mp3 C:\Music\</code>	Moves the file <i>song.mp3</i> to the <i>C:\Music</i> directory

With your Command Prompt open and configured, you’re ready to roll. Skip ahead to the section “Working with psql.”

macOS psql Setup

On macOS, you’ll run `psql` within Terminal, the application that provides access to that system’s command line via one of several *shell* programs such as `bash` or `zsh`. Shell programs on Unix- or Linux-based systems, including macOS, provide not only the command prompt where users enter instructions, but also their own programming language for automating tasks. For example, you can use `bash` commands to write a program to log in to a remote computer, transfer files, and log out.

If you followed the Postgres.app setup instructions for macOS in Chapter 1—including running the commands at your Terminal—you shouldn’t need additional configuration to use `psql` and associated commands. Instead, we’ll proceed to launching Terminal.

Launching and Configuring the macOS Terminal

Launch Terminal by navigating to **Applications▶Utilities▶Terminal**. When it opens, you should see a window that displays the date and time of your last login followed by a prompt that includes your computer name, current working directory, and username. On my Mac, which is set to the `bash` shell, the prompt displays as `ad:~ anthony$` and ends with a dollar sign (\$), as shown in [Figure 184](#).

The tilde (~) represents the system’s home directory, which is `/Users/anthony`. Terminal doesn’t display the full directory path, but you can see that information at any time by entering the `pwd` command (short for *print working directory*) and pressing ENTER on your keyboard. The area after the dollar sign is where you enter commands.

If your Mac is set to another shell such as `zsh`, your prompt may look different. With `zsh`, for example, the prompt ends with a percent sign (%). The particular shell you’re using does not make a difference for these exercises.

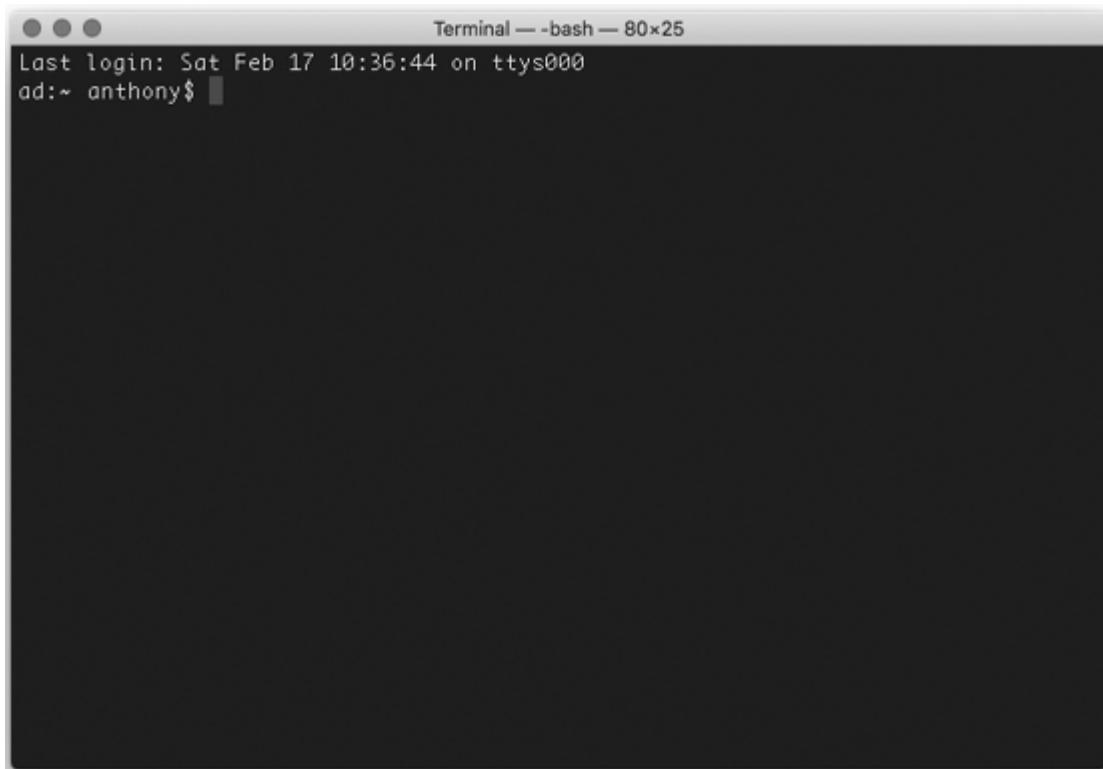


Figure 18-4: Terminal command line in macOS

NOTE

*For fast access to Terminal, add it to your macOS Dock. While Terminal is running, right-click its icon and select **Options▶Keep in Dock**.*

If you've never used Terminal, its default black-and-white color scheme might seem boring. You can change fonts, colors, and other settings by selecting **Terminal▶Preferences**. To make Terminal bigger to better fit the query output display, I recommend setting the window size (on the Window tab under Profiles) to a width of at least 80 columns and a height of 25 rows. My preferred font (on the Text tab) is Monaco 14, but experiment to find one you like.

Exploring the full workings of Terminal and related commands is an enormous topic beyond the scope of this book, but take some time to try several commands. [Table 18-2](#) lists some useful commonly used commands that aren't actually necessary for the exercises in this chapter. Enter `man` (short for *manual*) followed by a command name to get help on any command. For example, you

can use `man ls` to find out how to use the `ls` command to list directory contents.

Table 18-2: Useful Terminal Commands

Command	Function	Example	Action
cd	Change directory	<code>cd /Users/pparker/my-stuff/</code>	Changes to the <i>my-stuff</i> directory
cp	Copy files	<code>cp song.mp3 song_backup.mp3</code>	Copies the file <i>song.mp3</i> to <i>song_backup.mp3</i> in the current directory
grep	Find strings in a text file matching a regular expression	<code>grep 'us_counties_2010' *.sql</code>	Finds all lines in files with a <i>.sql</i> extension that have the text <i>us_counties_2010</i>
ls	List directory contents	<code>ls -al</code>	Lists all files and directories (including hidden) in “long” format
mkdir	Make a new directory	<code>mkdir resumes</code>	Makes a directory named <i>resumes</i> under the current working directory
mv	Move a file	<code>mv song.mp3 /Users/pparker/songs</code>	Moves the file <i>song.mp3</i> from the current directory to a <i>/songs</i> directory under a user directory
rm	Remove (delete) files	<code>rm *.jpg</code>	Deletes all files with a <i>.jpg</i> extension in the current directory (asterisk wildcard)

With your Terminal open and configured, you’re ready to roll. Skip ahead to the section “Working with psql.”

Linux psql Setup

Recall from “Linux Installation” in Chapter 1 that methods for installing PostgreSQL vary according to your Linux distribution. Nevertheless, `psql` is part of the standard PostgreSQL install, and you probably already ran `psql`

commands as part of the installation process via your distribution’s command line terminal application. Even if you didn’t, standard Linux installations of PostgreSQL will automatically add `psql` to your `PATH`, so you should be able to access it.

Launch a terminal application and proceed to the next section, “Working with `psql`.” On some distributions, such as Ubuntu, you can open a terminal by pressing CTRL-ALT-T. Also note that the Mac Terminal commands in [Table 18-2](#) apply to Linux as well and may be useful to you.

Working with `psql`

Now that you’ve identified your command line interface and set it up to recognize the location of `psql`, let’s launch `psql` and connect to a database on your local installation of PostgreSQL. Then we’ll explore executing queries and special commands for retrieving database information.

Launching `psql` and Connecting to a Database

Regardless of the operating system you’re using, you start `psql` in the same way. Open your command line interface (Command Prompt on Windows, Terminal on macOS or Linux). To launch `psql` and connect to a database, we use the following pattern at the command prompt:

```
psql -d database_name -U user_name
```

Following the `psql` application name, we provide the database name after a `-d` database argument and a username after the `-U` username argument.

For the database name, we’ll use `analysis`, which is where we created our tables and other objects for the book’s exercises. For username, we’ll use `postgres`, which is the default user created during installation. So, to connect to the `analysis` database on your local server, enter this at the command line:

```
psql -d analysis -U postgres
```

Note that you can connect to a database on a remote server by specifying the `-h` argument followed by the hostname. For example, you would use the following line if you were connecting to a database named `analysis` on a server called `example.com`:

```
psql -d analysis -U postgres -h example.com
```

Either way, if you set a password during installation, you should receive a password prompt when `psql` launches. If so, enter your password. After `psql` connects to the database, you should then see a prompt that looks like this:

```
psql (13.3)
Type "help" for help.

analysis=#
```

Here, the first line lists the version number of `psql` and the server you're connected to. Your version will vary depending on when you installed PostgreSQL. The prompt where you'll enter commands is `analysis=#`, which refers to the name of the database, followed by an equal sign (=) and a hash mark (#). The hash mark indicates that you're logged in with *superuser* privileges, which give you unlimited ability to access and create objects and set up accounts and security. If you're logged in as a user without superuser privileges, the last character of the prompt will be a greater-than sign (>). As you can see, the user account you logged in with here (`postgres`) is a superuser.

NOTE

PostgreSQL installations create a default superuser account called `postgres`. If you're running Postgres.app on macOS, that installation created an additional superuser account that has your system username and no password.

Finally, on Windows systems, you may see a warning message after launching `psql` about the console code page differing from the Windows code page. This is related to a mismatch in character sets between Command Prompt and the rest of the Windows system. For the exercises in this book, you can safely ignore that warning. If you prefer, you can eliminate it on a per-session basis by entering the command `cmd.exe /c chcp 1252` in your Windows Command Prompt before launching `psql`.

Getting Help or Exiting

At the `psql` prompt, you can get help for both `psql` and general SQL with a set of *meta-commands*, detailed in [Table 18-3](#). Meta-commands, which begin with a backslash (\), go beyond offering help—we'll cover several that return information about the database, let you adjust settings, or process data.

Table 18-3: Help Commands Within `psql`

Comma nd	Displays
\?	Commands available within <code>psql</code> , such as <code>\dt</code> to list tables.
\? options	Options for use with the <code>psql</code> command, such as <code>-U</code> to specify username.
\? variables	Variables for use with <code>psql</code> , such as <code>VERSION</code> for the current <code>psql</code> version.
\h	List of SQL commands. Add a command name to see detailed help for it (for example, <code>\h INSERT</code>).

Even experienced users often need a refresher on commands and options, so having the details in the `psql` application is handy. To exit `psql`, use the meta-command `\q` (for *quit*).

Changing the Database Connection

When working with SQL, it's not unusual to be working with multiple databases, so you need a way to switch between databases. You can do this easily at the `psql` prompt using the `\c` meta-command.

For example, while connected to your `analysis` database, at the `psql` prompt enter the following command to create a database named `test`:

```
analysis=# CREATE DATABASE test;
```

Then, to connect to the new `test` database you just created, enter `\c` followed by the name of the database at the `psql` prompt (and provide your PostgreSQL password if asked):

```
analysis=# \c test
```

The application should respond with the following message:

```
You are now connected to database "test" as user "postgres".  
test=#
```

The prompt will show you which database you're connected to. To log in as a different user—for example, using a username the macOS installation created—you could add that username after the database name. On my Mac, the syntax looks like this:

```
analysis-# \c test anthony
```

The response would be as follows:

```
You are now connected to database "test" as user "anthony".  
test=#
```

To reduce clutter, you can remove the `test` database you created. First, use `\c` to disconnect from `test` and connect to the `analysis` database (we can remove a database only if no one is connected to it). Once you've connected to `analysis`, enter `DROP DATABASE test;` at the `psql` prompt.

Setting Up a Password File

If you'd rather not see a password prompt when starting `psql`, you can set up a file to store database connection information that includes the server name, your username, and password. On startup, `psql` will read the file and bypass the password prompt if the file contains an entry that matches the database connection and username.

On Windows 10, the file must be named `pgpass.conf` and must reside in the following directory: `C:\USERS\YourUsername\AppData\Roaming\postgresql\`. You may need to create the `postgresql` directory. On macOS and Linux, the file must be named `.pgpass` and must reside in your user home directory. The documentation at <https://www.postgresql.org/docs/current/libpq-pgpass.html> notes that on macOS and Linux, you may need to set file permissions after creating the file by running `chmod 0600 ~/.pgpass` at the command line.

Create the file using a text editor and save it with the correct name and location for your system. Inside, you'll need to add a line for each database connection in the following format:

```
hostname:port:database:username:password
```

For example, to set up a connection for the `analysis` database and `postgres` username, enter this line, substituting your password:

```
localhost:5432:analysis:postgres:password
```

You can substitute an asterisk in any of the first four parameters to serve as a wildcard. For example, to supply a password for any local database with the `postgres` username, substitute an asterisk for the database name:

```
localhost:5432:*:postgres:password
```

Saving your password will save you some typing, but be mindful of best practices regarding security. Always secure your computer with a strong password and/or physical security key, and don't create a password file on any public or shared system.

Running SQL Queries on psql

We've configured `psql` and connected to a database, so now let's run some SQL queries. We'll start with a single-line query and then run a multiple-line query.

You enter SQL into `psql` directly at the prompt. For example, to see a few rows from the 2019 census table we've used throughout the book, enter a query at the prompt, as shown in [Listing 18-1](#).

```
analysis=# SELECT county_name FROM us_counties_pop_est_2019
ORDER BY county_name LIMIT 3;
```

[Listing 18-1](#): Entering a single-line query in `psql`

Press ENTER to execute the query, and `psql` should display the following results in your terminal in text including the number of rows returned:

```
county_name
-----
Abbeville County
Acadia Parish
Accomack County
```

```
(3 rows)
```

```
analysis=#
```

Below the result, you can see the analysis=# prompt again, ready for further input. You can use the up and down arrows on your keyboard to scroll through recent queries and press ENTER to execute them again, avoiding having to retype them.

Entering a Multiline Query

You're not limited to single-line queries. If you have a query that spans multiple lines, you can enter lines one at a time, pressing ENTER after each, and psql knows not execute the query until you provide a semicolon. Let's reenter the query in [Listing 18-1](#) over multiple lines, shown in [Listing 18-2](#).

```
analysis=# SELECT county_name
analysis-# FROM us_counties_pop_est_2019
analysis-# ORDER BY county_name
analysis-# LIMIT 3;
```

[Listing 18-2: Entering a multiline query in psql](#)

Note that when your query extends past one line, the symbol between the database name and the hash mark changes from an equal sign to a hyphen. This multiline query executes only when you press ENTER after the final line, which ends with a semicolon.

Checking for Open Parentheses in the psql Prompt

Another helpful feature of psql is that it shows when you haven't closed a pair of parentheses. [Listing 18-3](#) shows this in action.

```
analysis=# CREATE TABLE wineries (
analysis(# id bigint,
analysis(# winery_name text
analysis(# );
CREATE TABLE
```

[Listing 18-3: Showing open parentheses in the psql prompt](#)

Here, you create a simple table called `wineries` that has two columns. After entering the first line of the `CREATE TABLE` statement and an open parenthesis `(`), the prompt then changes from `analysis=#` to `analysis(#` to include an open parenthesis. This reminds you an open parenthesis needs closing. The prompt maintains that configuration until you add the closing parenthesis.

NOTE

If you have a lengthy query saved in a text file, such as one from this book's resources, you can copy it to your computer clipboard and paste it into psql (CTRL-V on Windows, COMMAND-V on macOS, and SHIFT-CTRL-V on Linux). That saves you from typing the whole query. After you paste the query text into psql, press ENTER to execute it.

Editing Queries

To modify the most recent query you've executed in `psql`, you can edit it using the `\e` or `\edit` meta-command. Enter `\e` to open the last-executed query in a text editor. The editor `psql` uses by default depends on your operating system.

On Windows, `psql` will open Notepad, a simple GUI text editor. Edit your query, save it by choosing **File▶Save**, and then exit Notepad using **File▶Exit**. When Notepad closes, `psql` should execute your revised query.

On macOS and Linux, `psql` uses a command line application called `vim`, which is a favorite among programmers but can seem inscrutable for beginners. Check out a helpful `vim` cheat sheet at <https://vim.rtorr.com/>. For now, you can use the following steps to make simple edits:

When `vim` opens the query in your terminal, press `I` to activate insert mode.

Make your edits to the query.

Press `ESC` and then `SHIFT-:` to display a colon command prompt at the bottom left of the `vim` screen, which is where you enter commands to control `vim`.

Enter `wq` (for *write, quit*) and press `ENTER` to save your changes.

Now when `vim` exits, the `psql` prompt should execute your revised query. Press the up-arrow key to see the revised text.

Navigating and Formatting Results

The query you ran in Listings 18-1 and 18-2 returned only one column and a handful of rows, so its output was contained nicely in your command line interface. But for queries with more columns or rows, the output can fill more than one screen, making it difficult to navigate. Fortunately, you have several ways to customize the display style of the output using the `\pset` meta-command.

Setting Paging of Results

One way to adjust the output format is to specify how `psql` handles scrolling through lengthy query results, known as *paging*. By default, if your results return more rows than will fit on one screen, `psql` will display the first screen's worth of rows and then let you scroll through the rest. For example, [Listing 18-4](#) shows what happens at the `psql` prompt when we remove the `LIMIT` clause from the query in [Listing 18-1](#).

```
analysis=# SELECT county_name FROM us_counties_pop_est_2019
ORDER BY county_name;

          county_name
-----
Abbeville County
Acadia Parish
Accomack County
Ada County
Adair County
Adair County
Adair County
Adair County
Adams County
Adams County
Adams County
Adams County
-- More --
```

[Listing 18-4](#): A query with scrolling results

Recall that this table has 3,142 rows. [Listing 18-4](#) shows only the first 12 on the current screen (the number of visible rows depends on your terminal configuration). On Windows, the indicator `-- More --` tells you that additional results are available, and you can press ENTER to scroll through them. On

macOS and Linux, the indicator will be a colon. Scrolling through a few thousand rows will take a while. Press Q to exit the results and return to the `psql` prompt.

To bypass manual scrolling and have all your results immediately display, you can change the pager setting using the `\pset pager` meta-command. Run that command at your `psql` prompt, and it should return the message `Pager usage is off`. Now when you rerun the query in [Listing 18-3](#) with the pager setting turned off, you should see results like this:

```
--snip--  
York County  
York County  
York County  
York County  
Young County  
Yuba County  
Yukon-Koyukuk Census Area  
Yuma County  
Yuma County  
Zapata County  
Zavala County  
Ziebach County  
(3142 rows)
```

```
analysis=#
```

You're immediately taken to the end of the results without having to scroll. To turn paging back on, run `\pset pager` again.

Formatting the Results Grid

You also can use `\pset` with the following options to format the results:

border int

Use this option to specify whether the results grid has no border (0), internal lines dividing columns (1), or lines around all cells (2). For example, `\pset border 2` sets lines around all cells.

format unaligned

Use the option `\pset format unaligned` to display the results in lines separated by a delimiter rather than in columns, similar to what you would see in a CSV file. The separator defaults to a pipe symbol (`|`). You can set a different separator using the `fieldsep` command. For example, to set a comma as the separator, run `\pset fieldsep ',', '`. To revert to a column view, run `\pset format aligned`. You can use the `psql` meta-command `\a` to toggle between aligned and unaligned views.

footer

Use this option to toggle the results footer, which displays the result row count, on or off.

null

Use this option to set how `psql` displays `NULL` values. By default, they show as blanks. You can run `\pset null '(null)'` to replace blanks with `(null)` when the column value is `NULL`.

You can explore additional options in the PostgreSQL documentation at <https://www.postgresql.org/docs/current/app-psql.html>. In addition, it's possible to set up a `.psqlrc` file on macOS or Linux or a `psqlrc.conf` file on Windows to hold your configuration preferences and load them each time `psql` starts. A good example is provided at <https://www.citusdata.com/blog/2017/07/16/customizing-my-postgres-shell-using-psqlrc/>.

Viewing Expanded Results

Sometimes, it's helpful to view results arranged not in the typical table style of rows and columns, but instead in a vertical list. This is useful when the number of columns is too big to fit on-screen in the normal horizontal results grid and also for scanning values in columns row by row. In `psql`, you can switch to a vertical list view using the `\x` (for *expanded*) meta-command. The best way to understand the difference between normal and expanded view is by looking at an example. [Listing 18-5](#) shows the normal display you see when querying the `grades` table in Chapter 17 using `psql`.

```
analysis=# SELECT * FROM grades ORDER BY student_id,
course_id;
student_id | course_id |      course      | grade
```

```
-----+-----+-----+-----  
1 | 1 | Biology 2 | C  
1 | 2 | English 11B | D  
1 | 3 | World History 11B | C  
1 | 4 | Trig 2 | B  
(4 rows)
```

[Listing 18-5](#): Normal display of the `grades` table query

To change to the expanded view, enter `\x` at the `psql` prompt, which should display the message `Expanded display is on`. Then, when you run the same query again, you should see the expanded results, as shown in [*Listing 18-6*](#).

```
analysis=# SELECT * FROM grades ORDER BY student_id,  
course_id;  
-[ RECORD 1 ]-----  
student_id | 1  
course_id | 1  
course | Biology 2  
grade | C  
-[ RECORD 2 ]-----  
student_id | 1  
course_id | 2  
course | English 11B  
grade | D  
-[ RECORD 3 ]-----  
student_id | 1  
course_id | 3  
course | World History 11B  
grade | C  
-[ RECORD 4 ]-----  
student_id | 1  
course_id | 4  
course | Trig 2  
grade | B
```

[Listing 18-6](#): Expanded display of the `grades` table query

The results appear in vertical blocks separated by record numbers. Depending on your needs and the type of data you're working with, this format might be easier to read. You can revert to column display by entering `\x` again at the `psql` prompt. In addition, setting `\x auto` will make PostgreSQL automatically display the results in a table or expanded view based on the size of the output.

Next, let's explore how to use `psql` to dig into database information.

Meta-Commands for Database Information

You can have `psql` display details about databases, tables, and other objects via a particular set of meta-commands. To see how these work, we'll explore the meta-command that displays the tables in your database, including how to append a plus sign (+) to the command to expand the output and add an optional pattern to filter the output.

To see a list of tables, you can enter `\dt` at the `psql` prompt. Here's a snippet of the output on my system:

List of relations			
Schema	Name	Type	Owner
public	acs_2014_2018_stats	table	anthony
public	cbp_naics_72_establishments	table	anthony
public	char_data_types	table	anthony
public	check_constraint_example	table	anthony
public	crime_reports	table	anthony
--snip--			

This result lists all tables in the current database alphabetically.

You can filter the output by adding a pattern the database object name must match. For example, use `\dt us*` to show only tables whose names begin with `us` (the asterisk acts as a wildcard). The results should look like this:

List of relations			
Schema	Name	Type	Owner
public	us_counties_2019_shp	table	anthony
public	us_counties_2019_top10	table	anthony
public	us_counties_pop_est_2010	table	anthony
public	us_counties_pop_est_2019	table	anthony
public	us_exports	table	anthony

[Table 18-4](#) shows several additional commands you might find helpful, including \l, which lists the databases on your server. Adding a plus sign to each command, as in \dt+, adds more information to the output, including object sizes.

Table 18-4: Example of `psql \d` Commands

Command	Displays
\d [pattern]	Columns, data types, plus other information on objects
\di [pattern]	Indexes and their associated tables
\dt [pattern]	Tables and the account that owns them
\du [pattern]	User accounts and their attributes
\dv [pattern]	Views and the account that owns them
\dx [pattern]	Installed extensions
\l [pattern]	Databases

The entire list of commands is available in the PostgreSQL documentation at <https://www.postgresql.org/docs/current/app-psql.html>, or you can see details by using the \? command noted earlier.

Importing, Exporting, and Using Files

In this section, we'll explore how to use psql to import and export data from the command line, which can be necessary when you're connected to remote servers, such as Amazon Web Services instances of PostgreSQL. We'll also use psql to read and execute SQL commands stored in a file and learn the syntax for sending psql output to a file.

Using \copy for Import and Export

In Chapter 5, you learned how to use the PostgreSQL COPY command to import and export data. It's a straightforward process, but it has one significant limitation: the file you're importing or exporting must be on the same machine as the PostgreSQL server. That's fine if you're working on your local machine, as you've been doing with these exercises. But if you're connecting to a database on a remote computer, you might not have access to its file system. You can get around this restriction by using the \copy meta-command in psql.

The `\copy` meta-command works just like the PostgreSQL `COPY`, except when you execute it at the `psql` prompt, it will route data from your machine to the server you're connected to, whether local or remote. We won't actually connect to a remote server to try this since it's rare to find a public remote server we could connect to, but you can still learn the syntax by using the commands on our local `analysis` database.

In [*Listing 18-7*](#), at the `psql` prompt we use a `DELETE` statement to remove all the rows from the small `state_regions` table you created in Chapter 10 and then import data using `\copy`. You'll need to change the file path to match the location of the file on your computer.

```
analysis=# DELETE FROM state_regions;
DELETE 56
analysis=# \copy state_regions FROM
'C:\YourDirectory\state_regions.csv' WITH (FORMAT CSV,
HEADER);
COPY 56
```

[*Listing 18-7: Importing data using \copy*](#)

Next, to import the data, we use `\copy` with the same syntax used with PostgreSQL `COPY`, including a `FROM` clause with the file path on your machine, and a `WITH` clause that specifies the file is a CSV and has a header row. When you execute the statement, the server should respond with `COPY 56`, letting you know the rows have been successfully imported.

If you're connected to a remote server via `psql`, you would use the same `\copy` syntax, and the command would route your local file to the remote server for importing. In this example, we used `\copy FROM` to import a file. We could also use `\copy TO` for exporting. Let's look at an alternate way to import or export data (or run other SQL commands) via `psql`.

Passing SQL Commands to `psql`

By placing a command in quotes after the `-c` argument, we can send it to our connected server, local or remote. The command can be a single SQL statement, multiple SQL statements separated by semicolons, or a meta-command. This can allow us to run `psql`, connect to a server, and execute a command in a single command line statement—handy if we want to incorporate `psql` statements into shell scripts to automate tasks.

For example, we can import data to the `state_regions` table with the statement in [Listing 18-8](#), which must be entered on one line at your command prompt (and not inside `psql`).

```
psql -d analysis -U postgres -c1 'COPY state_regions FROM
STDIN2 WITH (FORMAT CSV, HEADER);' <3
C:\YourDirectory\state_regions.csv
```

[Listing 18-8](#): Importing data using `psql` with `COPY`

To try it, you'll need to first run `DELETE FROM state_regions;` inside `psql` to clear the table. Then exit `psql` by typing the meta-command `\q`.

At your command prompt, enter the statement in [Listing 18-8](#). We first use `psql` and the `-d` and `-U` commands to connect to your `analysis` database. Then comes the `-c` command **1**, which we follow with the PostgreSQL statement for importing the data. The statement is similar to `COPY` statements we've used with one exception: after `FROM`, we use the keyword `STDIN 2` instead of the complete file path and filename. `STDIN` means “standard input,” which is a stream of input data that can come from a device, a keyboard, or in this case the file `state_regions.csv`, which we direct **3** to `psql` using the less-than (`<`) symbol. You'll need to supply the full path to the file.

Running this entire command at your command prompt should import the CSV file and generate the message `COPY 56`.

Saving Query Output to a File

It's sometimes helpful to save the query results and messages generated during a `psql` session to a file, such as to keep a history of your work or to use the output in a spreadsheet or other application. To send query output to a file, you can use the `\o` meta-command along with the full path and name of an output file that `psql` will create.

NOTE

On Windows, file paths for the `\o` command must use either Linux-style forward slashes, such as `C:/my-stuff/my-file.txt`, or double backslashes, such as `C:\\my-stuff\\my-file.txt`.

For example, in [Listing 18-9](#) we change the psql format style from a table to CSV and then output query results directly to a file.

```
| analysis=# \pset format csv
|   Output format is csv.

| analysis=# SELECT * FROM grades ORDER BY student_id,
| course_id;
? student_id,course_id,course,grade
1,1,Biology 2,F
1,2,English 11B,D
1,3,World History 11B,C
1,4,Trig 2,B

} analysis=# \o 'C:/YourDirectory/query_output.csv'

analysis=# SELECT * FROM grades ORDER BY student_id,
course_id;
| analysis=#


---


```

[Listing 18-9:](#) Saving query output to a file

First, we set the output format **1** using the meta-command `\pset format csv`. When you run a simple `SELECT` on the `grades` table, the output **2** should return as values separated by commas. Next, to send that data to a file the next time you run the query, use the `\o` meta-command and then provide a complete path to a file called `query_output.csv` **3**. When you run the `SELECT` query again, there should be no output to the screen **4**. Instead, you'll find a file with the contents of the query in the directory specified at **3**.

Note that every time you run a query from this point, the output is appended to the same file specified after the `\o` (for *output*) command. To stop saving output to that file, you can either specify a new file or enter `\o` with no filename to resume having results output to the screen.

Reading and Executing SQL Stored in a File

To run SQL stored in a text file, you execute `psql` on the command line and supply the filename after an `-f` (for file) argument. This syntax lets you quickly run a query or table update from the command line or in conjunction with a system scheduler to run a job at regular intervals.

Let's say you saved the SELECT query from [Listing 18-9](#) in a file called *display-grades.sql*. To run the saved query, use the following psql syntax at your command line:

```
psql -d analysis -U postgres -f C:\YourDirectory\display-grades.sql
```

When you press ENTER, psql should launch, run the stored query in the file, display the results, and exit. For repetitive tasks, this workflow can save considerable time because you avoid launching pgAdmin or rewriting a query. You also can stack multiple queries in the file so they run in succession, which, for example, you might do if you want to run several updates on your database.

Additional Command Line Utilities to Expedite Tasks

PostgreSQL also has its own set of command line utilities that you can enter in your command line interface without launching psql. A listing is available at <https://www.postgresql.org/docs/current/reference-client.html>, and I'll explain several in Chapter 19 that are specific to database maintenance. Here I'll cover two that are particularly useful: creating a database at the command line with the createdb utility and loading shapefiles into a PostGIS database via the shp2pgsql utility.

Adding a Database with createdb

Earlier in the chapter, you used `CREATE DATABASE` to add the database `test` to your PostgreSQL server. We can achieve the same thing using `createdb` at the command line. For example, to create a new database on your server named `box_office`, run the following at your command line:

```
createdb -U postgres -e box_office
```

The `-U` argument tells the command to connect to the PostgreSQL server using the `postgres` account. The `-e` argument (for *echo*) prints the commands generated by `createdb` as output. Running this command creates the database and prints output to the screen ending with `CREATE DATABASE box_office;`. You can then connect to the new database via psql using the following line:

```
psql -d box_office -U postgres
```

The `createdb` command accepts arguments to connect to a remote server (just like `psql` does) and to set options for the new database. A full list of arguments is available at <https://www.postgresql.org/docs/current/app-createdb.html>. Again, the `createdb` command is a time-saver that comes in handy when you don't have access to a GUI.

Loading Shapefiles with shp2pgsql

In Chapter 15, you learned about shapefiles, which contain data describing spatial objects. On Windows and some Linux distributions, you can import shapefiles into a PostGIS-enabled database using the Shapefile Import/Export Manager GUI tool (generally) included with PostGIS. However, the Shapefile Import/Export Manager is not always included with PostGIS on macOS or some flavors of Linux. In those cases (or if you'd rather work at the command line), you can import a shapefile using the PostGIS command line tool `shp2pgsql`.

To import a shapefile into a new table from the command line, use the following syntax:

```
shp2pgsql -I -s SRID -W encoding shapefile_name table_name |  
psql -d database -U user
```

A lot is happening in this single line. Here's a breakdown of the arguments (if you skipped Chapter 15, you might need to review it now):

-I Uses GiST to add an index on the new table's geometry column.

-s Lets you specify an SRID for the geometric data.

-W Lets you specify encoding. (Recall that we used `Latin1` for census shapefiles.)

shapefile_name The name (including full path) of the file ending with the `.shp` extension.

table_name The name of the table the shapefile is imported to.

Following these arguments, you place a pipe symbol (`|`) to direct the output of `shp2pgsql` to `psql`, which has the arguments for naming the database and user. For example, to load the `tl_2019_us_county.shp` shapefile into a

`us_counties_2019_shp` table in the `analysis` database, you can run the following command. Note that although this command wraps onto two lines here, it should be entered as one line in the command line:

```
shp2pgsql -I -s 4269 -W Latin1 tl_2019_us_county.shp  
us_counties_2019_shp | psql -d analysis -U postgres
```

The server should respond with a number of SQL `INSERT` statements before creating the index and returning you to the command line. It might take some time to construct the entire set of arguments the first time around, but after you've done one, subsequent imports should take less time. You can simply substitute file and table names into the syntax you already wrote.

Wrapping Up

Feeling mysterious and powerful yet? Indeed, when you delve into a command line interface and make the computer do your bidding using text commands, you enter a world of computing that resembles a sci-fi movie sequence. Not only does working from the command line save you time, it also helps you overcome barriers you might hit when working in environments that don't support graphical tools. In this chapter, you learned the basics of working with the command line plus PostgreSQL specifics. You discovered your operating system's command line application and set it up to work with `psql`. Then you connected `psql` to a database and learned how to run SQL queries via the command line. Many experienced computer users prefer to use the command line for its simplicity and speed once they become familiar with using it. You might, too.

In Chapter 19, we'll review common database maintenance tasks including backing up data, changing server settings, and managing the growth of your database. These tasks will give you more control over your working environment and help you better manage your data analysis projects.

TRY IT YOURSELF

To reinforce the techniques in this chapter, choose an example from an earlier chapter and try working through it using only the command line. Chapter 15, “Analyzing Spatial Data with PostGIS,” is a good choice because it gives you the opportunity to work with `psql` and the shapefile loader `shp2pgsql`. That said, I encourage you to choose any example that you think you would benefit from reviewing.

19

MAINTAINING YOUR DATABASE



To wrap up our exploration of SQL, we'll look at key database maintenance tasks and options for customizing PostgreSQL. In this chapter, you'll learn how to track and conserve space in your databases, how to change system settings, and how to back up and restore databases. How often you'll need to perform these tasks depends on your current role and interests. If you want to be a *database administrator* or a *backend developer*, the topics covered here are vital.

It's worth noting that database maintenance and performance tuning are large enough topics that they often occupy entire books, and this chapter mainly serves as an introduction to a handful of essentials. If you want to learn more, a good place to begin is with the resources in the appendix.

Let's start with the PostgreSQL VACUUM feature, which lets you shrink the size of tables by removing unused rows.

Recovering Unused Space with VACUUM

The PostgreSQL VACUUM command helps manage the size of a database, which—as discussed in “Improving Performance When Updating Large Tables” in Chapter 10—can grow as a result of routine operations.

For example, when you update a row value, the database creates a new version of that row with the updated value and retains (but hides) the old version of the row. The PostgreSQL documentation refers to these rows that you can't see as *dead tuples*, with *tuples*—an ordered list of elements—being the name for the internal implementation of rows in a PostgreSQL database. The same thing happens when you delete a row. Though the row is no longer visible to you, it lives on as a dead row in the table.

This is by design so the database can provide certain features in environments where multiple transactions are occurring, and an old version of a row might be needed by transactions other than the current one.

The VACUUM command cleans up these dead rows. Running VACUUM on its own designates the space occupied by dead rows as available for the database to use again (assuming that any transactions using the rows have been completed). In most cases, VACUUM doesn't return the space to your system's disk; it just flags that space as available for new data. To actually shrink the size of the data file, you can run VACUUM FULL, which rewrites the table to a new version that doesn't include the dead row space. It drops the old version.

Although VACUUM FULL frees space on your system's disk, there are a couple of caveats to keep in mind. First, VACUUM FULL takes more time to complete than VACUUM. Second, it must have exclusive access to the table while rewriting it, which means that no one can update data during the operation. The regular VACUUM command can run while updates and other operations are happening. Finally, not all dead space in a table is bad. In many cases, having available space to put new tuples instead of needing to ask the operating system for more disk space can improve performance.

You can run either VACUUM or VACUUM FULL on demand, but PostgreSQL by default runs an *autovacuum* background process that monitors the database and runs VACUUM as needed. Later in this chapter, I'll show you how to monitor autovacuum as well as run the VACUUM command manually. But first, let's look at how a table grows as a result of updates and how you can track this growth.

Tracking Table Size

We'll create a small test table and monitor its growth as we fill it with data and perform an update. The code for this exercise, as with all resources for the book, is available at <https://nostarch.com/practical-sql-2nd-edition/>.

Creating a Table and Checking Its Size

[Listing 19-1](#) creates a `vacuum_test` table with a single column to hold an integer. Run the code, and then we'll measure the table's size.

```
CREATE TABLE vacuum_test (
    integer_column integer
);
```

[Listing 19-1](#): Creating a table to test vacuuming

Before we fill the table with test data, let's check how much space it occupies on disk to establish a reference point. We can do so in two ways: check the table properties via the pgAdmin interface or run queries using PostgreSQL administrative functions. In pgAdmin, click once on a table to highlight it, and then click the **Statistics** tab. Table size is one of about two dozen indicators in the list.

I'll focus on the running queries technique here because knowing these queries is helpful if for some reason pgAdmin isn't available or you're using another graphical user interface (GUI). [Listing 19-2](#) shows how to check the `vacuum_test` table size using PostgreSQL functions.

```
SELECT 1pg_size.pretty(
    2pg_total_relation_size('vacuum_test')
);
```

[Listing 19-2](#): Determining the size of `vacuum_test`

The outermost function, `pg_size.pretty()` 1, converts bytes to a more easily understandable format in kilobytes, megabytes, or gigabytes. Wrapped inside is the `pg_total_relation_size()` function 2, which reports how many bytes a table, its indexes, and any offline compressed data takes up on disk. Because the table is empty at this point, running the code in pgAdmin should return a value of 0 bytes, like this:

```
pg_size.pretty
-----
0 bytes
```

You can get the same information using the command line. Launch `psql` as you learned in Chapter 18. Then, at the prompt, enter the meta-command `\dt+` `vacuum_test`, which should display the following information including table size (I've omitted one column for space):

List of relations					
Schema	Name	Type	Owner	Persistence	Size
<hr/>					
public	vacuum_test	table	postgres	permanent	0 bytes

Again, the current size of the `vacuum_test` table should display 0 bytes.

Checking Table Size After Adding New Data

Let's add some data to the table and then check its size again. We'll use the `generate_series()` function introduced in Chapter 12 to fill the table's `integer_column` with 500,000 rows. Run the code in [Listing 19-3](#) to do this.

```
INSERT INTO vacuum_test
SELECT * FROM generate_series(1,500000);
```

[Listing 19-3](#): Inserting 500,000 rows into `vacuum_test`

This standard `INSERT INTO` statement adds the results of `generate_series()`, which is a series of values from 1 to 500,000, as rows to the table. After the query completes, rerun the query in [Listing 19-2](#) to check the table size. You should see the following output:

```
pg_size.pretty
-----
17 MB
```

The query reports that the `vacuum_test` table, now with a single column of 500,000 integers, uses 17MB of disk space.

Checking Table Size After Updates

Now, let's update the data to see how that affects the table size. We'll use the code in [Listing 19-4](#) to update every row in the `vacuum_test` table by adding 1

to the `integer_column` values, replacing the existing value with a number that's one greater.

```
UPDATE vacuum_test
SET integer_column = integer_column + 1;
```

Listing 19-4: Updating all rows in `vacuum_test`

Run the code, and then test the table size again.

```
pg_size.pretty
-----
35 MB
```

The table size doubled from 17MB to 35MB! The increase seems excessive, because the `UPDATE` simply replaced existing numbers with values of a similar size. As you might have guessed, the reason for this increase in table size is that for every updated value, PostgreSQL creates a new row, and the dead row remains in the table. Even though you see only 500,000 rows, the table has double that number. This behavior can lead to surprises for database owners who don't monitor disk space.

Before looking at how using `VACUUM` and `VACUUM FULL` affects the table's size on disk, let's review the process that runs `VACUUM` automatically as well as how to check on statistics related to table vacuums.

Monitoring the Autovacuum Process

PostgreSQL's autovacuum process monitors the database and launches `VACUUM` automatically when it detects a large number of dead rows in a table. Although autovacuum is enabled by default, you can turn it on or off and configure it using the settings I'll cover later in "Changing Server Settings." Because autovacuum runs in the background, you won't see any immediately visible indication that it's working, but you can check its activity by querying data that PostgreSQL collects about system performance.

PostgreSQL has its own *statistics collector* that tracks database activity and usage. You can look at the statistics by querying one of several views the system provides. (See a complete list of views for monitoring the state of the system in the PostgreSQL documentation under "The Statistics Collector":

<https://www.postgresql.org/docs/current/monitoring-stats.html>) To check the activity

of autovacuum, we query the `pg_stat_all_tables` view, as shown in [Listing 19-5](#).

```
SELECT 1relname,
       2last_vacuum,
       3last_autovacuum,
       4vacuum_count,
       5autovacuum_count
  FROM pg_stat_all_tables
 WHERE relname = 'vacuum_test';
```

[Listing 19-5](#): Viewing autovacuum statistics for `vacuum_test`

As you learned in Chapter 17, a view provides the results of a stored query. The query stored by the view `pg_stat_all_tables` returns a column called `relname` 1, which is the name of the table, plus columns with statistics related to index scans, rows inserted and deleted, and other data. For this query, we're interested in `last_vacuum` 2 and `last_autovacuum` 3, which contain the last time the table was vacuumed manually and automatically, respectively. We also ask for `vacuum_count` 4 and `autovacuum_count` 5, which show the number of times the vacuum was run manually and automatically.

By default, autovacuum checks tables every minute. So, if a minute has passed since you last updated `vacuum_test`, you should see details of vacuum activity when you run the query in [Listing 19-5](#). Here's what my system shows (note that I've removed the seconds from the time to save space here):

relname	last_vacuum	last_autovacuum	vacuum_count	autovacuum_count
vacuum_test		2021-09-02 14:46	0	1

The table shows the date and time of the last autovacuum, and the `autovacuum_count` column shows one occurrence. This result indicates that autovacuum executed a `VACUUM` command on the table once. However, because we've not vacuumed manually, the `last_vacuum` column is empty, and the `vacuum_count` is 0.

NOTE

The autovacuum process also runs the `ANALYZE` command, which gathers data on the contents of tables. PostgreSQL stores this information and uses it to execute queries efficiently in the future. You can run `ANALYZE` manually if needed.

Recall that `VACUUM` designates dead rows as available for the database to reuse but typically doesn't reduce the size of the table on disk. You can confirm this by rerunning the code in [Listing 17-2](#), which shows the table remains at 35MB even after the automatic vacuum.

Running VACUUM Manually

To run `VACUUM` manually, you can use the single line of code in [Listing 19-6](#).

```
VACUUM vacuum_test;
```

[Listing 19-6:](#) Running `VACUUM` manually

This command should return the message `VACUUM` from the server. Now when you fetch statistics again using the query in [Listing 17-5](#), you should see that the `last_vacuum` column reflects the date and time of the manual vacuum you just ran, and the number in the `vacuum_count` column should increase by one.

In this example, we executed `VACUUM` on our test table, but you can also run `VACUUM` on the entire database by omitting the table name. In addition, you can add the `VERBOSE` keyword to return information such as the number of rows found in a table and the number of rows removed, among other information.

Reducing Table Size with VACUUM FULL

Next, we'll run `VACUUM` with the `FULL` option, which actually returns the space taken up by dead tuples back to disk. It does this by creating a new version of a table with the dead rows discarded.

To see how `VACUUM FULL` works, run the command in [Listing 19-7](#).

```
VACUUM FULL vacuum_test;
```

[*Listing 19-7*](#): Using `VACUUM FULL` to reclaim disk space

After the command executes, test the table size again. It should be back down to 17MB, the size it was when we first inserted data.

It's never prudent or safe to run out of disk space, so minding the size of your database files as well as your overall system space is a worthwhile routine to establish. Using `VACUUM` to prevent database files from growing bigger than they have to is a good start.

Changing Server Settings

You can alter the settings for your PostgreSQL server by editing values in `postgresql.conf`, one of several configuration text files that control server settings. Other files include `pg_hba.conf`, which controls connections to the server, and `pg_ident.conf`, which database administrators can use to map usernames on a network to usernames in PostgreSQL. See the PostgreSQL documentation on these files for details; here we'll just cover `postgresql.conf` because it contains settings you may likely want to change. Most of the values in the file are set to defaults you may never need to adjust, but it's worth exploring in case you're fine-tuning your system. Let's start with the basics.

Locating and Editing `postgresql.conf`

The location of `postgresql.conf` varies depending on your operating system and install method. You can run the command in [*Listing 19-8*](#) to locate the file.

```
SHOW config_file;
```

[*Listing 19-8*](#): Showing the location of `postgresql.conf`

When I run the command on macOS, it shows the path to the file, as shown here:

```
/Users/anthony/Library/Application Support/Postgres/var-  
13/postgresql.conf
```

To edit `postgresql.conf`, navigate in your file system to the directory displayed by `SHOW config_file;` and open the file using a text editor. Don't use a rich-text

editor like Microsoft Word, as it may add additional formatting to the file.

NOTE

It's a good idea to save an unaltered copy of `postgresql.conf` for reference in case you make a change that breaks the system and you need to revert to the original version.

When you open the file, the first several lines should read as follows:

```
# -----
# PostgreSQL configuration file
# -----
#
# This file consists of lines of the form:
#
#   name = value
--snip--
```

The `postgresql.conf` file is organized into sections that specify settings for file locations, security, logging of information, and other processes. Many lines begin with a hash mark (#), which indicates the line is commented out and the setting shown is the active default.

For example, in the `postgresql.conf` file section “Autovacuum Parameters,” the default is for autovacuum to be turned on (which is a good, standard practice). The hash mark (#) in front of the line means that the line is commented out and the default value is in effect:

```
#autovacuum = on                      # Enable autovacuum subprocess?
'on'
```

To change this or other default settings, you would remove the hash mark, adjust the setting value, and save `postgresql.conf`. Some changes, such as to memory allocations, require a restart of the server; they're noted in `postgresql.conf`. Other changes require only a reload of settings files. You can reload settings files by running the function `pg_reload_conf()` under an account with superuser permissions or by executing the `pg_ctl` command, which we'll cover in the next section.

[***Listing 19-9***](#) shows settings you may want to change, excerpted from the *postgresql.conf* section “Client Connection Defaults.” Use your text editor to search the file for the following.

```
| datestyle = 'iso, mdy'  
|  
2 timezone = 'America/New_York'  
|  
3 default_text_search_config = 'pg_catalog.english'
```

[***Listing 19-9***](#): Sample *postgresql.conf* settings

You can use the `datestyle` setting **1** to specify how PostgreSQL displays dates in query results. This setting takes two parameters separated by a comma: the output format and the ordering of month, day, and year. The default for the output format is the ISO format `YYYY-MM-DD` we’ve used throughout this book, which I recommend for its cross-national portability. However, you can also use the traditional SQL format `MM/DD/YYYY`, the expanded Postgres format `Sun Nov 12 22:30:00 2023 EST`, or the German format `DD.MM.YYYY` with dots between the date, month, and year (`12.11.2023`). To specify the format using the second parameter, arrange *m*, *d*, and *y* in the order you prefer.

The `timezone` **2** parameter sets the server time zone. [***Listing 19-9***](#) shows the value `America/New_York`, which reflects the time zone on my machine when I installed PostgreSQL. Yours should vary based on your location. When setting up PostgreSQL for use as the backend to a database application or on a network, administrators often set this value to `UTC` and use that as a standard on machines across multiple locations.

The `default_text_search_config` **3** value sets the language used by the full-text search operations. Here, mine is set to `english`. Depending on your needs, you can set this to `spanish`, `german`, `russian`, or another language of your choice.

These three examples represent only a handful of settings available for adjustment. Unless you end up deep in system tuning, you probably won’t have to tweak much else. Also, use caution when changing settings on a network server used by multiple people or applications; changes can have unintended consequences, so it’s worth communicating with colleagues first.

Next, let’s look at how to use `pg_ctl` to make changes take effect.

NOTE

The PostgreSQL `ALTER SYSTEM` command can also be used to update settings.

The command creates settings in the file `postgresql.auto.conf` that will override values in `postgresql.conf`. See <https://www.postgresql.org/docs/current/sql-altersystem.html> for details.

Reloading Settings with pg_ctl

The command line utility `pg_ctl` allows you to perform actions on a PostgreSQL server, such as starting and stopping it and checking its status. Here, we'll use the utility to reload the settings files so the changes we make will take effect. Running the command reloads all settings files at once.

You'll need to open and configure a command line prompt the same way you did in Chapter 18 when you learned how to set up and use `psql`. After you launch a command prompt, use one of the following commands to reload, replacing the path with the path to the PostgreSQL data directory:

On Windows, use `pg_ctl reload -D "C:\path\to\data\directory"`.

On macOS or Linux, use `pg_ctl reload -D '/path/to/data/directory'`.

To find the location of your PostgreSQL data directory, run the query in [Listing 19-10](#).

```
SHOW data_directory;
```

[Listing 19-10](#): Showing the location of the data directory

Place the path after the `-D` argument, between double quotes on Windows and single quotes on macOS or Linux. You run this command on your system's command prompt, not inside the `psql` application. Enter the command and press ENTER; it should respond with the message `server signaled`. The settings files will be reloaded, and changes should take effect.

If you've changed settings that require a server restart, replace `reload` in [Listing 19-10](#) with `restart`.

NOTE

On Windows, you may need to run Command Prompt with administrator privileges to execute `pg_ctl` statements. Navigate to Command Prompt in your Start menu, right-click, and select **Run as Administrator**.

Backing Up and Restoring Your Database

You might want to back up your entire database either for safekeeping or for transferring data to a new or upgraded server. PostgreSQL offers command line tools that make backup and restore operations easy. The next few sections show examples of how to export data from a database or a single table to a file, as well as how to restore data from an export files.

Using pg_dump to Export a Database or Table

The PostgreSQL command line tool `pg_dump` creates an output file that contains all the data from your database; SQL commands for re-creating tables, views, functions, and other database objects; and commands for loading the data into tables. You can also use `pg_dump` to save only selected tables in your database. By default, `pg_dump` outputs a text file; I'll discuss an alternate custom compressed format first and then discuss other options.

To export the `analysis` database we've used for our exercises to a file, run the command in [Listing 19-11](#) at your system's command prompt (not in `psql`).

```
pg_dump -d analysis -U user_name -Fc -v -f
analysis_backup.dump
```

[Listing 19-11:](#) Exporting the `analysis` database with `pg_dump`

Here, we start the command with `pg_dump` and use similar connection arguments as with `psql`. We specify the database to export with the `-d` argument, followed by the `-U` argument and your username. Next, we use the `-Fc` argument to specify that we want to generate this export in a custom PostgreSQL compressed format and the `-v` argument to generate verbose output. Then we use the `-f` argument to direct the output of `pg_dump` to a text file named `analysis_backup.dump`. To place the file in a directory other than the

one your terminal prompt is currently open to, you can specify the complete directory path before the filename.

When you execute the command, depending on your installation, you might see a password prompt. Fill in that password, if prompted. Then, depending on the size of your database, the command could take a few minutes to complete. You'll see a series of messages about the objects the command is reading and outputting. When it's done, it should return you to a new command prompt, and you should see a file named *analysis_backup.dump* in your current directory.

To limit the export to one or more tables that match a particular name, use the *-t* argument followed by the name of the table in single quotes. For example, to back up just the *train_rides* table, use the following command:

```
pg_dump -t 'train_rides' -d analysis -U user_name -Fc -v -f  
train_backup.dump
```

Now let's look at how to restore the data from the export file, and then we'll explore additional *pg_dump* options.

Restoring a Database Export with pg_restore

The *pg_restore* utility restores data from your exported database file. You might need to restore your database when migrating data to a new server or when upgrading to a new major version of PostgreSQL. To restore the *analysis* database (assuming you're on a server where *analysis* doesn't exist), at the command prompt, run the command in [Listing 19-12](#).

```
pg_restore -C -v -d postgres -U user_name analysis_backup.dump
```

[Listing 19-12](#): Restoring the *analysis* database with *pg_restore*

After *pg_restore*, you add the *-C* argument, which tells the utility to create the *analysis* database on the server. (It gets the database name from the export file.) Then, as you saw previously, the *-v* argument provides verbose output, and *-d* specifies the name of the database to connect to, followed by the *-U* argument and your username. Press ENTER, and the restore will begin. When it's done, you should be able to view your restored database via *psql* or in *pgAdmin*.

Exploring Additional Backup and Restore Options

You can configure pg_dump with multiple options to include or exclude certain database objects, such as tables matching a name pattern, or to specify the output format. For example, when we backed up the analysis database, we specified the -Fc argument with pg_dump to generate the backup in a custom PostgreSQL compressed format. By excluding the -Fc argument, the utility will output in plain text, and you can view the contents of the backup with a text editor. For details, check the full pg_dump documentation at <https://www.postgresql.org/docs/current/app-pgdump.html>. For corresponding restore options, check the pg_restore documentation at <https://www.postgresql.org/docs/current/app-pgrestore.html>.

You also may want to explore the pg_basebackup command, which can back up multiple databases running on a PostgreSQL server. See <https://www.postgresql.org/docs/current/app-pgbasebackup.html> for details. An even more robust backup solution is pgBackRest (<https://pgbackrest.org/>), a free, open source application with options such as cloud integration for storage and the ability to create full, incremental, or differential backups.

Wrapping Up

In this chapter, you learned how to track and conserve space in your databases using the VACUUM feature in PostgreSQL. You also learned how to change system settings as well as back up and restore databases using other command line tools. You may not need to perform these tasks every day, but the maintenance tricks you learned here can help enhance the performance of your databases. Note that this is not a comprehensive overview of the topic; see the appendix for more resources on database maintenance.

In the next and final chapter of this book, I'll share guidelines for identifying hidden trends and telling an effective story using your data.

TRY IT YOURSELF

Using the techniques you learned in this chapter and earlier in the book, create a database and add a small table and some data. Then back up the database, delete it, and restore it using `pg_dump` and `pg_restore`.

If you make your backup using the default text format instead of compressed, you can use a text editor to explore the file created by `pg_dump` to examine how it organizes the statements to create objects and insert data.

20

TELLING YOUR DATA'S STORY



Learning SQL can be fun in and of itself, but it serves a greater purpose: it helps unearth the stories in your data. As you've learned, SQL gives you the tools to find interesting trends, insights, or anomalies in your data and then make smart decisions based on what you've learned. But how do you identify these trends from just a collection of rows and columns? And how can you glean meaningful insights from these trends after identifying them?

In this chapter, I outline a process I've used as a journalist and product developer to discover stories in data and communicate my findings. I'll start with how to generate ideas by asking good questions and gathering and exploring data. Then I explain the analysis process, which culminates in presenting your findings clearly. Identifying trends in your dataset and creating a narrative of your findings sometimes requires considerable experimentation and enough fortitude to weather the occasional dead end. Regard these tips as less of a checklist and more of a guideline to help ensure a thorough analysis that minimizes mistakes.

Start with a Question

Curiosity, intuition, or sometimes just dumb luck can often spark ideas for data analysis. If you're a keen observer of your surroundings, you might notice changes in your community over time and wonder if you can measure that change. Consider your local real estate market. If you see "For Sale" signs popping up around town more than usual, you might start asking questions. Is there a dramatic increase in home sales this year compared with last year? If so, by how much? Which neighborhoods are riding the wave? These questions create a great opportunity for data analysis. If you're a journalist, you might find a story. If you run a business, you might see a marketing opportunity.

Likewise, if you surmise that a trend is occurring in your industry, confirming it might provide you with a business opportunity. For example, if you suspect that sales of a particular product are sluggish, you can analyze data to confirm the hunch and adjust inventory or marketing efforts appropriately.

Keep track of these ideas and prioritize them according to their potential value. Analyzing data to satisfy your curiosity is perfectly fine, but if the answers can make your institution more effective or your company more profitable, that's a sign they're worth pursuing.

Document Your Process

Before you delve into analysis, consider how to make your process transparent and reproducible. For the sake of credibility, others in your organization as well as those outside it should be able to reproduce your work. In addition, make sure you document enough of your process so that if you set the project aside for several weeks, you won't have trouble getting up to speed when you return to it.

There isn't one right way to document your work. Taking notes on research or creating step-by-step SQL queries that another person could follow to replicate your data import, cleaning, and analysis can make it easier for others to verify your findings. Some analysts store notes and code in a text file. Others use version control systems, such as GitHub, or work in code notebooks. What's important is you create a system of documentation and use it consistently.

Gather Your Data

After you've hatched an idea for analysis, the next step is to find data that relates to the trend or question. If you're working in an organization that already has its own data on the topic, lucky you—you're set! In that case, you might be able to tap internal marketing or sales databases, customer relationship management (CRM) systems, or subscriber or event registration data. But if your topic encompasses broader issues involving demographics, the economy, or industry-specific subjects, you'll need to do some digging.

A good place to start is to ask experts about the sources they use. Analysts, government decision-makers, and academics can point you to available data and describe its usefulness. Federal, state, and local governments, as you've seen throughout the book, produce volumes of data on all kinds of topics. In the United States, check out the federal government's data catalog site at <https://www.data.gov/> or individual federal agency sites, such as the National Center for Education Statistics (NCES) at <https://nces.ed.gov/> or the Bureau of Labor Statistics at <https://www.bls.gov/>.

You can also browse local government websites. Any time you see a form for users to fill out or a report formatted in rows and columns, those are signs that structured data might be available for analysis. All is not lost if you have access only to unstructured data, though—as you learned in Chapter 14, you can even mine unstructured data, such as text files, for analysis.

If the data you want to analyze was collected over multiple years, I recommend examining five or ten years or more, instead of just one or two, if possible. Analyzing a snapshot of data collected over a month or a year can yield interesting results, but many trends play out over a longer period of time and may not be evident if you look at a single year of data. I'll discuss this further in the section "Identify Key Indicators and Trends over Time."

No Data? Build Your Own Database

Sometimes, no one has the data you need in a format you can use. If you have time, patience, and a methodology, you might be able to build your own dataset. That is what my *USA Today* colleague Robert Davis and I did when we wanted to study issues related to the deaths of college students on campuses in the United States. Not a single organization—not the schools or state or federal officials—could tell us how many college students were dying each year from accidents, overdoses, or illnesses on campus. We decided to collect our own data and structure the information into tables in a database.

We started by researching news articles, police reports, and lawsuits related to student deaths. After finding reports of more than 600 student deaths from 2000 to 2005, we followed up with interviews with education experts, police, school officials, and parents. From each report, we cataloged details such as each student's age, school, cause of death, year in school, and whether drugs or alcohol played a role. Our findings led to the publication of the article "In College, First Year Is by Far the Riskiest" in *USA Today* in 2006. The story featured the key finding from the analysis of our SQL database: freshmen were particularly vulnerable and accounted for the highest percentage of the student deaths we studied.

You too can create a database if you lack the data you need. The key is to identify the pieces of information that matter and then systematically collect them.

Assess the Data's Origins

After you've identified a dataset, find as much information about its origins and maintenance methods as you can. Governments and institutions gather data in all sorts of ways, and some methods produce data that is more credible and standardized than others.

For example, you've already seen that US Department of Agriculture (USDA) food producer data included the same company names spelled in multiple ways. It's worth knowing why. (Perhaps the data is manually copied from a written form to a computer.) Similarly, the New York City taxi data you analyzed in Chapter 12 records the start and end times of each trip. This begs the question of when the timer starts and stops—when the passenger gets in and out of the vehicle, or is there some other trigger? You should know these details not only to draw better conclusions from analysis but also to pass them along to others who might be interpreting your analysis.

The origins of a dataset might also affect how you analyze the data and report your findings. For example, with US Census Bureau data, it's important to know that the decennial census conducted every 10 years is a complete count of the population, whereas the American Community Survey (ACS) is drawn from only a sample of households. As a result, ACS counts have a margin of error, but the decennial census doesn't. It would be irresponsible to report on the ACS without considering that the margin of error could render differences between numbers insignificant.

Interview the Data with Queries

Once you have your data, understand its origins, and have it loaded into your database, you can explore it with queries. Throughout the book, I call this step *interviewing data*, which is what you should do to find out more about the contents of your data and whether they contain any red flags.

A good place to start is with aggregates. Counts, sums, sorting, and grouping by column values should reveal minimum and maximum values, potential issues with duplicate entries, and a sense of the general scope of your data. If your database contains multiple, related tables, try joins to make sure you understand how the tables relate. Using `LEFT JOIN` and `RIGHT JOIN`, as you learned in Chapter 7, should show whether key values from one table are missing in another. That may or may not be a concern, but at least you'll be able to identify potential problems to address. Jot down a list of questions or concerns you have, and then move on to the next step.

Consult the Data's Owner

After exploring your database and forming early conclusions about the quality and trends you observed, take time to bring questions or concerns to a person who knows the data well. That person could work at the government agency or firm that gave you the data, or the person might be an analyst who has worked with the data before. This step is your chance to clarify your understanding of the data, verify initial findings, and discover whether the data has any issues that make it unsuitable for your needs.

For example, if you're querying a table and notice values in columns that seem to be gross outliers (such as dates in the future for events that were supposed to have happened in the past), you should ask about that discrepancy. If you expect to find someone's name in a table (perhaps even your own name) and it's not there, that should prompt another question. Is it possible you don't have the whole dataset, or is there a problem with data collection?

The goal is to get expert help to do the following:

Understand the limits of the data. Make sure you know what the data includes, what it excludes, and any caveats about content that might affect how you perform your analysis.

Make sure you have a complete dataset. Verify that you have all the records you should expect to see and that if any data is missing, you understand why.

Determine whether the dataset suits your needs. Consider looking elsewhere for more reliable data if your source acknowledges problems with the data's quality.

Every dataset and situation is unique, but consulting another user or owner of the data can help you avoid unnecessary missteps.

Identify Key Indicators and Trends over Time

When you're satisfied that you understand the data and are confident in its trustworthiness, completeness, and appropriateness to your analysis, the next step is to run queries to identify key indicators and, if possible, trends over time.

Your goal is to unearth data that you can summarize in a sentence or present as a slide in a presentation. An example of a finding would be something like this: "After five years of declines, the number of people enrolling in Widget University has increased by 5 percent for two consecutive semesters."

To identify this type of trend, you'll follow a two-step process:

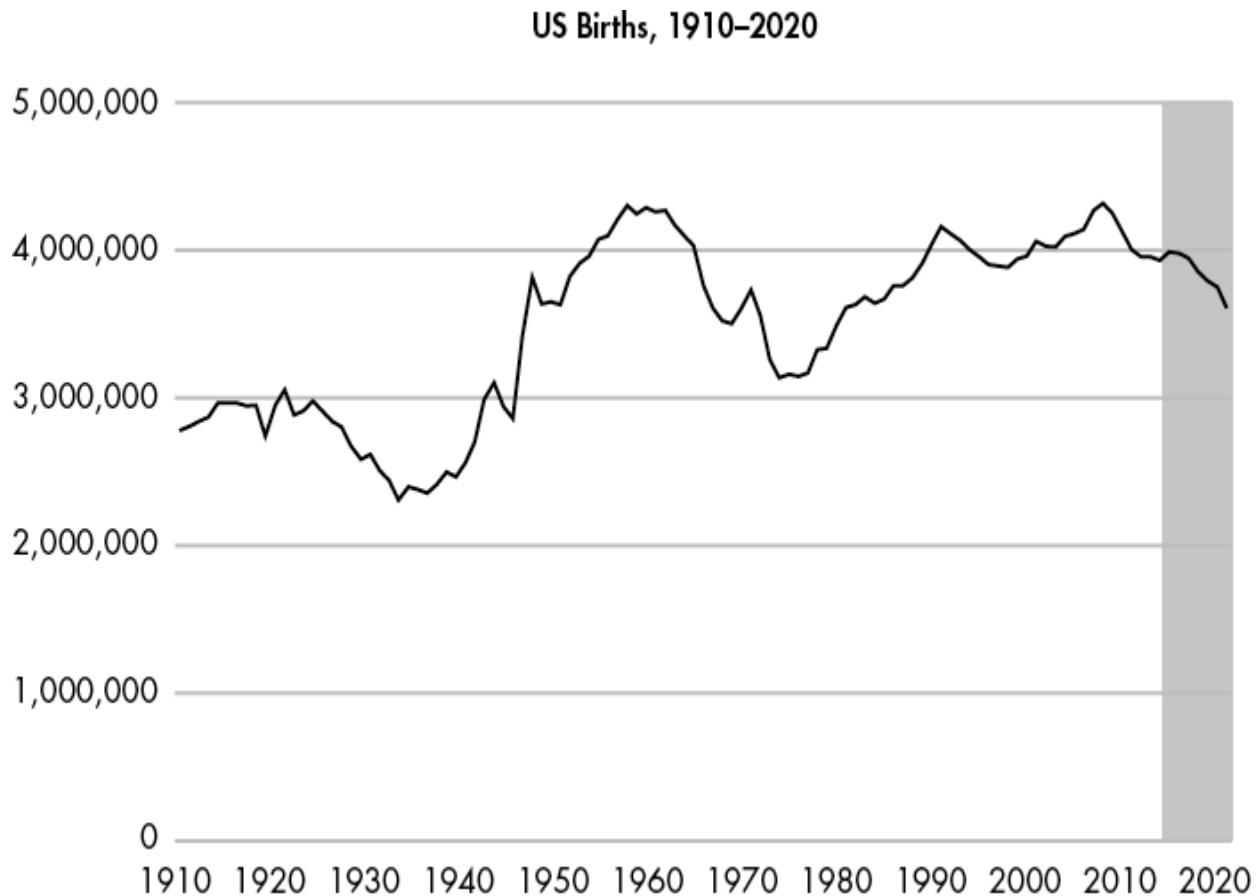
Choose an indicator to track. In census data, it might be the percentage of the population that is over age 60. Or in the New York City taxi data, it could be the median number of weekday trips over the span of one year.

Track that indicator over multiple years to see how it has changed, if at all.

In fact, these are the steps we used in Chapter 7 to apply percent change calculations to multiple years of census data contained in joined tables. In that case, we looked at the change in population in counties between 2010 and 2019. The population estimate was the key indicator, and the percent change showed the trend over the nine-year span for each county.

One caveat about measuring change over time: even when you see a dramatic change between any two years, it's worth digging into as many years' worth of data as possible to understand the shorter-term change in the context of a long-term trend. Any year-to-year change might seem dramatic, but seeing it in context of multiyear activity can help you assess its true significance.

For example, the US National Center for Health Statistics releases data on the number of babies born each year. As a data nerd, this is one of the indicators I like to keep tabs on, because births often reflect broader trends in culture or the economy. [Figure 20-1](#) shows the annual number of births from 1910 to 2020.



[Figure 20-1](#): US births from 1910 to 2020. Source: US National Center for Health Statistics

Looking at only the last five years of this graph (shaded in gray), we see that the number of births has declined steadily, to 3.61 million in 2020 from 3.95 million in 2016. The recent drops are indeed noteworthy (reflecting continuing decreases in birth rates and an aging population). But in the long-term context, we see that the nation has experienced several baby booms and busts in the past 100 years. One example you can see in [Figure 20-1](#) is the major rise in the mid-1940s following World War II, which signaled the start of the Baby Boom generation.

By identifying key indicators and looking at change over time, both short term and long term, you might uncover one or more findings worth presenting to

others or acting on.

NOTE

Any time you work with data from a survey, poll, or other sample, it's important to test for statistical significance. Are the results actually a trend or just the result of chance? Significance testing is a statistical concept beyond the scope of this book but one that data analysts should know. See the appendix for PostgreSQL resources for advanced statistics.

Ask Why

Data analysis can tell you what happened, but it doesn't always indicate why something happened. To learn the *why*, it's worth revisiting the data with experts in the topic or the owners of the data. In the US births data, it's easy to calculate year-to-year percent change from those numbers. But the data doesn't tell us why births steadily increased from the early 1980s to 1990. For that information, you could consult a demographer who would most likely explain that the rise in births during those years coincided with more Baby Boomers entering their child-bearing years.

As you share your findings and methodology with experts, ask them to note anything that seems unlikely or worthy of further examination. For the findings that they can corroborate, ask them to help you understand the forces behind those findings. If they're willing to be cited, you can use their comments to supplement your report or presentation. Quoting experts' insights about trends in this way is a standard approach journalists use.

Communicate Your Findings

How you share the results of your analysis depends on your role. A student might present their results in a paper or dissertation. A person who works in a corporate setting might present their findings using PowerPoint, Keynote, or Google Slides. A journalist might write a story or produce a data visualization. Regardless of the end product, here are tips for presenting the information well, using a fictional home sales analysis as an example:

Identify an overarching theme based on your findings. Make the theme the title of your presentation, paper, or visualization. For example, you might title a presentation on real estate “Home Sales Rise in Suburban Neighborhoods, Fall in Cities.”

Present overall numbers to show the general trend. Highlight the key findings from your analysis. For example, “All suburban neighborhoods saw sales rise 5 percent each of the last two years, reversing three years of declines. Meanwhile, city neighborhoods saw a 2 percent decline.”

Highlight specific examples that support the trend. Describe one or two relevant cases. For example, “In Smithtown, home sales increased 15 percent following the relocation of XYZ Corporation’s headquarters last year.”

Acknowledge examples counter to the overall trend. Use one or two relevant cases here as well. For example, “Two city neighborhoods did show growth in home sales: Arvis (up 4.5%) and Zuma (up 3%).”

Stick to the facts. Never distort or exaggerate any findings.

Provide expert insights. Use quotes or citations.

Visualize numbers using bar charts, line charts, or maps. Tables are helpful for giving your audience specific numbers, but it’s easier to understand trends from a visualization.

Cite the source of the data and what your analysis includes or omits. Provide dates covered, the name of the provider, and any distinctions that affect the analysis, for example, “Based on Walton County tax filings in 2022 and 2023. Excludes commercial properties.”

Share your data. Post data online for download, including a description of the steps you took to analyze it. Nothing says transparency more than sharing your data with others so they can perform their own analysis and corroborate your findings.

Generally, a short presentation that communicates your findings clearly and succinctly, and then invites dialogue from your audience, works best. Of course, you can follow your own preferred pattern for working with data and presenting your conclusions. But over the years, these steps have helped me avoid data errors and mistaken assumptions.

Wrapping Up

At last, you've reached the end of our practical exploration of SQL! Thank you for reading this book, and I welcome your suggestions and feedback via email at practicalsqlbook@gmail.com. At the end of this book is an appendix that lists additional PostgreSQL-related tools you might want to try.

I hope you've come away with data analysis skills you can start using immediately on the everyday data you encounter. More importantly, I hope you've seen that each dataset has a story, or several stories, to tell. Identifying and telling these stories is what makes working with data worthwhile; it's more than just combing through a collection of rows and columns. I look forward to hearing about what you discover!

TRY IT YOURSELF

It's your turn to find and tell a story using the SQL techniques we've covered. Using the process outlined in this chapter, consider a local or national topic and search for available data. Assess its quality, the questions it might answer, and its timeliness. Consult with an expert who knows the data and the topic well. Load the data into PostgreSQL and interview it using aggregate queries and filters. What trends can you discover? Summarize your findings in a short presentation.

APPENDIX

ADDITIONAL POSTGRESQL RESOURCES



This appendix contains resources to help you stay informed about PostgreSQL developments, find additional software, and get help. Because software resources are likely to change, I'll maintain a copy of this appendix at the GitHub repository that contains all the book's resources. You can find a link to GitHub via <https://nostarch.com/practical-sql-2nd-edition/>.

PostgreSQL Development Environments

Throughout the book, we've used the graphical user interface pgAdmin to connect to PostgreSQL, run queries, and view database objects. Although pgAdmin is free, open source, and popular, it's not your only choice for working with PostgreSQL. The wiki entry "PostgreSQL Clients" at https://wiki.postgresql.org/wiki/PostgreSQL_Clients catalogs many alternatives.

The following list shows several tools I've tried, including free and paid options. The free tools work well for general analysis work. If you wade deeper into database development, you might want to upgrade to the paid options, which typically offer advanced features and support.

Beekeeper Studio Free and open source GUI for PostgreSQL, MySQL, Microsoft SQL Server, SQLite, and other platforms. Beekeeper works on Windows, macOS, and Linux and features one of the more refined app designs among database GUIs (see <https://www.bekeeperstudio.io/>).

DBeaver Described as a “universal database tool” that works with PostgreSQL, MySQL, and many other databases, DBeaver includes a visual query builder, code completion, and other advanced features. There are paid and free versions for Windows, macOS, and Linux (see <https://dbeaver.com/>).

DataGrip A SQL development environment that offers code completion, bug detection, and suggestions for streamlining code, among many other features. It’s a paid product, but the company, JetBrains, offers discounts and free versions for students, educators, and nonprofits (see <https://www.jetbrains.com/datagrip/>).

Navicat A richly featured SQL development environment with versions that support PostgreSQL as well as other databases, including MySQL, Oracle, MongoDB, and Microsoft SQL Server. There is no free version of Navicat, but the company offers a 14-day free trial (see <https://www.navicat.com/>).

Postbird A simple cross-platform PostgreSQL GUI for writing queries and viewing objects. Free and open source (see <https://github.com/Paxal/postbird/>).

Postico A macOS-only client from the maker of Postgres.app that takes its cues from Apple design. The full version is paid, but a restricted-feature version is available with no time limit (see <https://eggerapps.at/postico/>).

A trial version can help you decide whether the product is right for you.

PostgreSQL Utilities, Tools, and Extensions

You can expand the capabilities of PostgreSQL via numerous third-party utilities, tools, and extensions. These range from additional backup and import/export options to improved formatting for the command line to powerful statistics packages. You’ll find a curated list online at <https://github.com/dhamaniasad/awesome-postgres/>, but here are several to highlight:

Devar Excel Add-in for PostgreSQL An Excel add-in that lets you load and edit data from PostgreSQL directly in Excel workbooks (see <https://www.devart.com/excel-addins/postgresql.html>).

MADlib A machine learning and analytics library for large data sets that integrates with PostgreSQL (see <https://madlib.apache.org/>).

pgAgent A job manager that lets you run queries at scheduled times, among other tasks (see <https://www.pgadmin.org/docs/pgadmin4/latest/pgagent.html>).

pgBackRest An advanced database backup and restore management tool (see <https://pgbackrest.org/>).

pgcli A substitute command-line interface for psql that includes autocompletion and syntax highlighting (see <https://github.com/dbcli/pgcli>).

pgRouting Enables a PostGIS-enabled PostgreSQL database to perform network analysis tasks, such as finding driving distance along roadways (see <https://pgrouting.org/>).

PL/R A loadable procedural language that provides the ability to use the R statistical programming language within PostgreSQL functions and triggers (see <https://www.joeconway.com/plr.html>).

pspg Formats the output of psql into sortable, scrollable tables with support for several color themes (see <https://github.com/okbob/pspg>).

PostgreSQL News and Community

Now that you're a bona fide PostgreSQL user, it's wise to stay on top of community news. The PostgreSQL development team updates the software on a regular basis, and changes might affect code you've written or tools you're using. You might even find new opportunities for analysis.

Here's a collection of online resources to help you stay informed:

Crunchy Data blog Posts from the team at Crunchy Data, which provides enterprise PostgreSQL support and solutions (see <https://blog.crunchydata.com/blog/>).

The EDB Blog Posts from the team at EDB, a PostgreSQL services company that provides the Windows installer referenced in this book and also leads development of pgAdmin (see <https://www.enterprisedb.com/blog/>).

Planet PostgreSQL Aggregates blog posts and announcements from the database community (see <https://planet.postgresql.org/>).

Postgres Weekly An email newsletter that rounds up announcements, blog posts, and product announcements (see <https://postgresweekly.com/>).

PostgreSQL mailing lists These lists are useful for asking questions of community experts. The pgsql-novice and pgsql-general lists are particularly good for beginners, although note that email volume can be heavy (see <https://www.postgresql.org/list/>).

PostgreSQL news archive Official news from the PostgreSQL team (see <https://www.postgresql.org/about/newsarchive/>).

PostgreSQL nonprofits PostgreSQL-related charitable organizations include the United States PostgreSQL Association and PostgreSQL Europe. Both provide education, events, and advocacy around the product (see <https://postgresql.us/> and <https://www.postgresql.eu/>).

PostgreSQL user groups A list of community groups that offer meetups and other activities (see <https://www.postgresql.org/community/user-groups/>).

PostGIS blog Announcements and updates about the PostGIS extension (see <https://postgis.net/blog/>).

Additionally, I recommend paying attention to developer notes for any of the PostgreSQL-related software you use, such as pgAdmin.

Documentation

Throughout this book, I've made frequent reference to pages in the official PostgreSQL documentation. You can find documentation for each version of the software along with an FAQ and wiki on the main page at <https://www.postgresql.org/docs/>. It's worth reading sections of the manual as you learn more about a topic, such as indexes, or search for all the options that come with functions. In particular, the "Preface," "Tutorial," and "SQL Language" sections cover much of the material presented in the book's chapters.

Other good resources for documentation are the Postgres Guide at <http://postgresguide.com/> and Stack Overflow, where you can find questions and answers posted by developers at <https://stackoverflow.com/questions/tagged/postgresql>. You can also check out the Q&A site for PostGIS at <https://gis.stackexchange.com/questions/tagged/postgis>.

Index

Please note that index links to approximate location of each term.

Symbols

- + (addition operator), [79](#)
- & (ampersand operator), [270](#)
- := (assignment operator), [359](#)
- * (asterisk)
 - as multiplication operator, [79](#)
 - as wildcard in SELECT statement, [30](#)
- \ (backslash), [249](#)
 - escaping characters with, [255](#)
- ,
- // (cube root operator), [81](#)
- { } (curly brackets), [249](#)
 - denoting an array in query output, [90](#)
- <-> (distance operator), [271](#)
- @@ (double at sign match operator), [266](#)
- :: (double-colon CAST operator), [56](#)
- \$\$ (double-dollar quoting), [351](#)
- || (double-pipe concatenation operator), [172](#), [261](#), [329](#)
- " (double quote), [61](#), [118](#)

- ! (exclamation point)
 - as factorial operator, [81](#)
 - as negation, [250](#), [270](#)
- ^ (exponentiation operator), [80](#)
- / (forward slash)
 - as division operator, [80](#)
 - in macOS file paths, [44](#)
- > (greater-than comparison operator), [35](#)
- >= (greater-than or equals comparison operator), [35](#)
- (hyphen subtraction operator), [79](#), [331](#)
- @> and <@ (JSON containment operators), [315](#)
- ? and ?| (JSON existence operators), [315](#)
- > and - (JSON field and element extraction operators), [310](#), [310–311](#)
- #- (JSON path deletion operator), [331](#)
- #> and ##> (JSON path extraction operators), [310](#), [314](#)
- < (less-than comparison operator), [35](#)
- <= (less-than or equals comparison operator), [35](#)
- <> (not-equal comparison operator), [35](#)
- != (not-equal comparison operator), [35](#)
- () (parentheses)
 - to designate order of operations, [38](#)
 - to specify columns for importing, [70](#)
- % (percent sign)
 - as modulo operator, [80](#)
 - wildcard for pattern matching, [37](#)
- | (pipe character)
 - as delimiter, [44](#), [63](#), [72](#)

to redirect output, [295](#)
; (semicolon), [19](#)
| / (square root operator), [81](#)
~* (tilde-asterisk case-insensitive matching operator), [250](#)
~ (tilde case-sensitive matching operator), [250](#)
_ (underscore wildcard for pattern matching), [37](#)

A

adding numbers, [79](#)
 across columns, [82](#)
addition operator (+), [79](#)
aggregate functions, [86](#), [142](#)
 avg(), [86](#), [200](#)
 binary (two-input), [186](#)
 count(), [143](#), [159](#)
 filtering with HAVING, [153](#)
 GROUP BY requirement, [146](#)
 interviewing data, [159](#)
 max(), [145](#)
 min(), [145](#)
 PostgreSQL documentation, [143](#)
 sum(), [86](#), [149](#)
aliases for table names, [107](#), [151](#)
ALTER COLUMN statement, [133](#)
ALTER SYSTEM command, [394](#)
ALTER TABLE statement, [165](#)
 ADD COLUMN, [165](#), [169](#), [286](#)

ADD CONSTRAINT, [133](#)
ALTER COLUMN, [165](#)
DROP COLUMN, [165](#), [176](#)
RENAME TO, [180](#)
to restart sequence, [127](#)
table constraints, adding and removing, [132](#)
American National Standards Institute (ANSI), [305](#)
SQL standard, [xxiv](#)
ampersand operator (&), [270](#)
Amtrak trip data, [218](#)
ANALYZE keyword
with EXPLAIN statement, [135](#)
with VACUUM, [391](#)
AND logical operator combining comparison operators, [38](#), [114](#)
antimeridian, [66](#)
API (application programming interface), [318](#)
array, [20](#), [256](#)
 array_length() function, [252](#)
 constructor example, [20](#)
 denoted by curly brackets in query output, [20](#), [256](#)
 extracting element with JSON operators, [312](#)
 functions, [91](#)
 index positions, [313](#)
 in JSON, [306](#), [332](#)
 notation in query, [260](#)
 passing into ST_MakePoint(), [284](#)
 returned from regexp_match(), [255](#)

unnest() function, [91](#)
array_length() function, [252](#)
ASC keyword in ORDER BY clause, [32](#)
AS keyword
 declaring table aliases with, [107](#), [113](#)
 renaming columns in query results with, [82](#), [106](#)
assignment operator (:=), [359](#)
asterisk (*)
 as multiplication operator, [79](#)
 as wildcard in SELECT statement, [30](#)
attribute, [22](#)
auto-incrementing integers, [23](#), [31](#), [46](#)
 gaps in sequence, [47](#), [126](#)
 identity column SQL standard, [46](#)
 overriding values, [127](#)
 restarting sequence, [127](#)
 as surrogate primary key, [125](#)
 using IDENTITY, [46](#), [122](#), [125](#)
autovacuum, [390](#)
 editing server setting, [393](#)
 time of last vacuum, [391](#)
avg() function, [86](#), [200](#), [226](#)

B

backslash (\), [249](#)
escaping characters with, [255](#)
backups

column, [168](#)
improving performance when updating tables, [179](#)
restoring values from another table, [170](#)
tables, [167](#)

bell curve. *See* normal distribution

BETWEEN comparison operator, [35](#), [229](#)
inclusive property, [36](#)

BINARY file format, [63](#)

birth data, US, [404](#)

Boolean expression
in constraint, [131](#)
in table join, [94](#)

Boolean value, [94](#)
used in table join, [94](#)

B-tree index, [134](#)

Bureau of Labor Statistics, [401](#)

C

CALL command, [352](#)

camel case, [27](#), [118](#)

carriage return in text files, [63](#)

Cartesian product as result of CROSS JOIN, [103](#)

CASCADE keyword, [129](#)

case sensitivity
with ILIKE operator, [37](#)
with LIKE operator, [37](#)

CASE statement, [241](#)

in common table expression, [242](#)
ELSE clause, [242](#)
syntax, [241](#)
with trigger, [358](#)
in UPDATE statement, [262](#)
WHEN clause, [241](#), [359](#)
CAST () function, [55](#)
 shortcut notation, [56](#)
categorizing data, [241](#)
char, [42](#)
character string types, [42](#)
 char, [42](#)
 functional difference from number types, [45](#)
 performance in PostgreSQL, [43](#)
 text, [42](#)
 varchar, [42](#)
char_length() function, [246](#)
CHECK constraint, [130](#)
classify_max_temp() user function, [358](#)
clock_timestamp() function, [208](#)
Codd, Edgar F., [xxiv](#), [93](#)
codes, distinguishing from numbers, [65](#)
column, [22](#)
 adding numbers in, [86](#)
 alias, [82](#)
 alter data type, [165](#)
 avoiding spaces in name, [119](#)

deleting, [176](#)
indexes, [136](#)
populating new during backup, [179](#)
retrieving in queries, [31](#)
updating values, [166](#)

comma (,), [60](#)

command line, [363](#)
advantages of using, [364](#)
createdb command, [383](#)
psql, [370](#)
setup, [364](#)
macOS, [368](#)
PATH environment variable, [364](#)
Windows, [364](#)

shell programs, [368](#)

COMMIT statement in transaction block, [177](#)

common table expression (CTE), [234](#)
with CASE statement, [242](#)
defining, [234](#)

comparison operators, [35](#)
combining with AND and OR, [38](#)

composite primary key, [121](#)

concatenation, [172](#)

conditional expression, [241](#)

constraints, [23](#), [120](#)
adding and removing, [132](#)
CHECK, [130](#), [185](#)

column vs. table, [121](#), [123](#)
CONSTRAINT keyword, [95](#), [123](#), [130](#)
foreign key, [128](#)
NOT NULL, [132](#)
primary key, [95](#), [121](#), [123](#)
UNIQUE, [96](#), [131](#)
violations when altering table, [166](#)

constructor, [90](#)

Coordinated Universal Time (UTC), [52](#), [206](#)
UTC offset, [52](#), [206](#), [219](#)

COPY statement
 DELIMITER option, [63](#)
 description of, [59](#)
 exporting data, [43](#), [72](#)
 FORMAT option, [63](#)
 FROM keyword, [62](#)
 HEADER option, [63](#)
 importing data, [62](#)
 JSON import, [310](#)
 naming file paths, [44](#)
 QUOTE option, [63](#)
 specifying columns to import, [68](#)
 specifying file formats, [63](#)
 specifying rows to import, [70](#)
 TO keyword, [72](#), [216](#)
 WHERE clause, [70](#)
 WITH keyword, [62](#)

`corr()` function, [186](#)

correlation does not imply causality, [189](#), [192](#)

interpreting correlation coefficients, [186](#)

correlated subquery, [224](#)

`count()` function, [143](#), [159](#), [227](#)

distinct values, [144](#)

with GROUP BY, [147](#)

on multiple columns, [148](#)

values present in a column, [144](#)

counting

distinct values, [144](#)

missing values displayed, [161](#)

rows, [143](#)

using pgAdmin, [144](#)

`CREATE DATABASE` statement, [19](#)

`createdb` utility, [383](#)

`CREATE EXTENSION` statement, [237](#), [276](#), [352](#)

`CREATE FUNCTION` statement, [347](#)

`CREATE INDEX` statement, [134](#), [136](#)

`CREATE TABLE` statement, [22](#)

backing up a table with, [167](#)

declaring data types, [41](#)

temporary table, [71](#)

`CREATE TRIGGER` statement, [356](#)

`CREATE VIEW` statement, [339](#)

`CROSS JOIN` keyword, [103](#), [236](#)

`crosstab()` function, [240](#)

syntax example, [239](#)
cross tabulations, [236](#)
 installing the `tablefunc` module, [237](#)
CSV (comma-separated values), [60](#)
 export using `COPY`, [72](#)
 header row, [61](#)
 opening with text editor, [2](#)
cube root operator (`||/`), [81](#)
curly brackets (`{ }`), [249](#)
 denoting an array in query output, [90](#)
`current_date` function, [208](#)
`current_time` function, [208](#)
`current_timestamp` function, [208](#)
cut points, [88](#)

D

data
 structured and unstructured, [245](#)
 telling its story, [399](#)
database
 backup and restore, [395](#)
 connecting to, [20](#), [22](#)
 create from command line, [383](#)
 creating, [17](#), [19](#)
 maintenance, [387](#)
database management system, [12](#), [19](#)
database server, [12](#), [19](#)

data dictionary, [41](#)
data types, [41](#)
 bigint, [45](#)
 bigserial, [23](#), [45](#), [46](#)
 boolean, [55](#)
 char, [42](#)
 character string types, [42](#)
 date, [22](#), [51](#)
 date and time types, [51](#)
 decimal, [47](#)
 declaring with CREATE TABLE, [41](#)
 double precision, [48](#)
 full-text search, [265](#)
 geography, [280](#)
 geometry, [281](#)
 importance of using appropriate type, [41](#)
 integer, [45](#)
 interval, [51](#)
 json, [54](#)
 jsonb, [54](#)
 modifying with ALTER COLUMN, [165](#)
 number types, [44](#)
 numeric, [23](#), [47](#)
 real, [48](#)
 returned by math operations, [78](#)
 serial, [31](#), [45](#), [46](#)
 serial types, [46](#)

smallint, [45](#)
smallserial, [45](#), [46](#)
text, [42](#)
time, [51](#)
timestamp, [51](#), [204](#)
transforming values with CAST(), [55](#)
tsquery, [266](#)
tsvector, [265](#)
varchar, [23](#), [42](#)

date data types

date, [51](#)
interval, [51](#)
date_part() function, [205](#), [215](#), [241](#)

dates

calculations with, [212](#)
input format, [22](#), [25](#), [36](#), [52](#), [204](#), [205](#)
matching with regular expressions, [253](#)
setting default style, [394](#)

Davis, Robert, [401](#)

daylight saving time, [210](#)

deciles, [90](#)

decimal data types, [47](#)

decimal degrees, [66](#)

DELETE CASCADE statement with foreign key constraint, [129](#)

DELETE statement, [70](#)

removing rows matching criteria, [175](#)

with subquery, [225](#)

delimited text files, [59](#)
text qualifiers, [61](#)

delimiter, comma as most common, [60](#)

DELIMITER keyword with COPY statement, [63](#)

dense_rank() function, [193](#)

derived table, [225](#)
joining, [226](#)

DESC keyword in ORDER BY clause, [32](#)

dirty data, [29](#), [157](#)
cleaning, [157](#)

foreign keys help to avoid, [129](#)

when to discard, [165](#)

distance operator (<->), [271](#)

DISTINCT keyword, [33](#)
with count(), [144](#)
in IS DISTINCT FROM clause, [169](#)
on multiple columns, [34](#)

division, [79](#)
finding the remainder, [80](#)
integer vs. decimal, [79](#), [80](#)

documenting code, [41](#)

double at sign match operator (@@), [266](#)

double-colon CAST operator (::), [56](#)

double-dollar quoting (\$\$), [351](#)

double-pipe concatenation operator (||), [172](#), [261](#), [329](#)

double quote ("), [61](#), [118](#)

DROP statement

COLUMN, [176](#)

INDEX, [137](#)

TABLE, [71](#), [176](#)

duplicate data created by spelling variations, [160](#)

E

Eastern Standard Time (EST), [52](#)

eliminating duplicate values in query, [34](#)

ELSE clause in CASE statement, [242](#)

entity, [19](#)

environment variables, [364](#)

epoch, [206](#), [320](#)

equals comparison operator (=), [35](#)

error messages

CSV import failure, [67](#), [69](#)

foreign key violation, [129](#)

out of range, [46](#)

primary key violation, [124](#)

relation already exists, [119](#)

UNIQUE constraint violation, [132](#)

when using CAST(), [56](#)

escaping characters, [255](#)

EST (Eastern Standard Time), [52](#)

exclamation point (!)

as factorial operator, [81](#)

as negation, [250](#), [270](#)

EXISTS operator, [229](#)

with subquery, [230](#)
in WHERE clause, [167](#)
EXPLAIN statement, [135](#)
 output of command, [135](#)
exponentiation operator (^), [80](#)
exporting data
 all data in table, [72](#)
 header row, including, [63](#)
 limiting columns, [73](#)
 from query results, [73, 216](#)
 using COPY statement, [59, 72, 216](#)
 using pgAdmin wizard, [74](#)
 using the command line, [380](#)
expressions, [53, 224](#)
 conditional, [241](#)
 subquery, [229](#)
extract() datetime function, [206](#)

F

factorial() function, [81](#)
factorials, [80](#)
false as Boolean value, [94](#)
Federal Information Processing Standards (FIPS), [65, 295](#)
field, [22](#)
file paths
 import and export file locations, [62](#)
 naming conventions for operating systems, [44, 62](#)

filtering rows

HAVING clause, [153](#), [160](#)

with subquery, [224](#)

WHERE clause, [35](#), [224](#)

findstr Windows command, [162](#)

FIPS (Federal Information Processing Standards), [65](#), [295](#)

fixed-point numbers, [47](#)

floating-point numbers, [48](#)

inexact math calculations, [49](#)

Food Safety Inspection Service, [158](#)

foreign key

creating with REFERENCES keyword, [128](#)

definition, [96](#), [128](#)

formatting SQL for readability, [27](#)

forward slash (/)

as division operator, [80](#)

in macOS file paths, [44](#)

FROM keyword with COPY, [62](#)

FULL OUTER JOIN keywords, [102](#)

full-text search, [265](#)

adjacent words, locating, [271](#)

data types, [265](#)

highlighting terms, [269](#)

language configurations, [266](#)

lexemes, [265](#)

multiple terms in query, [270](#)

querying, [268](#)

ranking results, [271](#)
setting default language, [394](#)
table and column setup, [267](#)
`to_tsquery()` function, [266](#)
`to_tsvector()` function, [265](#)
`ts_headline()` function, [269](#)
`ts_rank()` function, [272](#)
`ts_rank_cd()` function, [272](#)
using GIN index with, [268](#)
functions, [337](#)
difference with procedures, [349](#)
creating, [346](#)
full-text search, [265](#)
`IMMUTABLE` keyword, [348](#)
`RAISE NOTICE` keywords, [351](#)
`RETURNS` keyword, [348](#)
specifying language, [348](#)
string, [246](#)
structure of, [348](#)

G

generalized inverted index (GIN), [134](#), [268](#), [310](#)
generalized search tree (GiST), [134](#), [286](#)
`generate_series()` function, [209](#), [241](#), [389](#)
GeoJSON, [276](#), [319](#)
getting help when code goes bad, [26](#)
GIN (generalized inverted index), [134](#), [268](#), [310](#)

GIS (geographic information systems), [276](#)

decimal degrees, [66](#)

GiST (generalized search tree), [134](#), [286](#)

GitHub, downloading code resources from, [3](#)

graphical user interface (GUI), [292](#)

list of tools, [407](#)

greater-than comparison operator (>), [35](#)

greater-than or equals comparison operator (>=), [35](#)

grep Linux command, [162](#)

GROUP BY clause

with aggregate functions, [146](#)

eliminating duplicate values, [146](#)

on multiple columns, [147](#)

H

HAVING clause, [153](#), [160](#)

HEADER keyword, [63](#)

header row

in CSV file, [61](#), [64](#)

ignoring during import, [61](#)

hyphen subtraction operator (-), [79](#), [331](#)

I

identifiers

avoiding reserved keywords, [119](#)

enabling mixed case, [118](#)

naming, [27](#), [118](#)

quoting, [119](#)
IDENTITY keyword, [46](#), [125](#)
ILIKE comparison operator, [35](#), [37](#)
 case-insensitive search, [37](#)
importing data, [59](#), [62](#)
 adding default column value, [71](#)
 choosing a subset of columns, [68](#)
 choosing a subset of rows, [70](#)
 ignoring header row in text files, [61](#), [63](#)
 from non-text sources, [60](#)
 from text file format, [62](#)
 using COPY statement, [59](#)
 using pgAdmin wizard, [74](#)
 using the command line, [380](#)
IN comparison operator, [35](#), [172](#), [229](#)
 with subquery, [230](#)
indexes, [133](#)
 and ANSI SQL standard, [133](#)
 B-tree, [134](#)
 considerations before adding, [137](#)
 creating on columns, [136](#)
 dropping, [137](#)
 effect on performance, [135](#)
 GIN, [134](#)
 GiST, [134](#), [286](#)
 not included with table backups, [168](#)
 syntax for creating, [134](#)

initcap() function, [246](#)
inserting rows into a table, [25](#)
INSERT statement, [25](#)
Institute of Museum and Library Services, [140](#)
integer data types, [45](#)
 auto-incrementing, [46](#)
 basic math operations, [79](#)
 bigint, [45](#)
 bigserial, [46](#)
 difference in integer type capacities, [45](#)
 integer, [45](#)
 serial, [46](#)
 smallint, [45](#)
 smallserial , [46](#)
International Date Line, [66](#)
International Organization for Standardization (ISO), [52](#), [277](#)
 SQL standard, [xxiv](#)
 time format, [204](#)
interval data type
 calculations with, [53](#), [219](#)
 cumulative, [220](#)
 value options, [53](#)
interviewing data, [29](#), [159](#)
 across joined tables, [151](#)
 artificial values as indicators, [146](#), [150](#)
 checking for missing values, [32](#), [160](#)
 correlations, [185](#)

counting rows and values, [143](#)
determining correct format, [32](#)
finding inconsistent values, [162](#)
malformed values, [162](#)
maximum and minimum values, [145](#)
rankings, [193](#)
rates calculations, [196](#)
statistics, [183](#)
summing grouped values, [149](#)
unique combinations of values, [34](#)
ISO (International Organization for Standardization), [52](#), [277](#)
SQL standard, [xxiv](#)
time format, [204](#)

J

JOIN keyword, [24](#)
example of using, [100](#)
in FROM clause, [24](#)
with USING clause, [100](#)
join types
CROSS JOIN, [103](#), [236](#)
FULL OUTER JOIN, [102](#)
JOIN (INNER JOIN), [100](#), [151](#)
LEFT JOIN, [101](#)
list of, [98](#)
RIGHT JOIN, [101](#)
joining tables, [23](#)

derived tables, [226](#)
multiple-table joins, [107](#)
naming tables in column list, [106](#), [151](#)
performing calculations across tables, [112](#)
with set operators, [109](#)
spatial joins, [300](#)
specifying columns to query, [106](#)
using JOIN keyword, [24](#), [27](#)

`json_agg()` function, [329](#)
`jsonb_array_elements()` function, [332](#)
`jsonb_array_elements_text()` function, [332](#)
`jsonb_array_length()` function, [331](#)
`jsonb_build_object()` function, [329](#)
`jsonb_set()` function, [330](#)

JSON (JavaScript Object Notation), [54](#), [305](#)
array, [306](#)
considerations for using in database, [307](#)
containment operators (`@>` and `<@`), [315](#)
data types, [308](#)
`json`, [309](#)
`jsonb`, [309](#)
existence operators (`?` and `?|`), [315](#), [317](#)
extraction operators, [310](#)
field and element extraction operators (`->` and `-`), [310](#), [310–311](#)
functions
`json_agg()`, [329](#)
`jsonb_array_elements()`, [332](#)

`jsonb_array_elements_text()`, [332](#)
`jsonb_array_length()`, [331](#)
`jsonb_build_object()`, [329](#)
`jsonb_set()`, [330](#)
`to_json()`, [327](#)
generating, [327](#)
GeoJSON format, [319](#)
importing, [309](#)
indexing, [310](#), [319](#)
json and jsonb data types, [54](#)
key/value pairs, [54](#), [306](#)
manipulating, [327](#)
modifying key/value pairs, [329](#)
object, [306](#)
path extraction operators (#> and ##>), [310](#), [314](#)
processing functions, [331](#)
schema flexibility, [306](#)
structure of, [54](#), [306](#)
`justify_interval()` function, [221](#)

K

key columns
foreign key, [26](#)
primary key, [95](#)
relating tables with, [94](#)
key/value pairs, [306](#)
extracting from JSON, [311](#)

Korea standard time, [212](#)

L

LATERAL subquery, [231](#)

with FROM keyword, [231](#)

with JOIN keyword, [232](#)

latitude

in US Census data, [66](#)

in well-known text, [278](#)

least squares regression line, [190](#)

left() function, [247](#)

LEFT JOIN keywords, [101](#)

length() string function, [163](#), [246](#)

less-than comparison operator (<), [35](#)

less-than or equals comparison operator (<=), [35](#)

lexemes, [265](#)

LIKE comparison operator, [35](#)

case-sensitive search, [37](#)

LIKE expression in UPDATE statement, [171](#)

LIMIT clause, [67](#)

limiting number of rows query returns, [67](#)

linear regression, [189](#)

least squares regression line, [190](#)

linear relationship, [186](#)

Linux

file path declaration, [44](#), [62](#)

grep command, [162](#)

system permissions, [44](#)
Terminal setup, [370](#)
literals, [25](#)
localhost, [13](#), [20](#)
localtime function, [208](#)
logical operators, [38](#)
longitude
 positive and negative values, [68](#)
 in US Census data, [66](#)
 in well-known text, [278](#)
lower() string function, [246](#)

M

macOS
 file path declaration, [44](#), [62](#)
 Terminal, [368](#)
 bash shell, [368](#)
 entering instructions, [369](#)
 setup, [368](#)
 useful commands, [369](#)
 make_date() function, [207](#)
 make_time() function, [207](#)
 make_timestamptz() function, [207](#)
 many-to-many table relationship, [105](#)
 map projection, [279](#)
 matching operators, [37](#)
 math

across joined table columns, [112](#)

across table columns, [82](#)

order of operations, [81](#)

math operators, [78](#)

addition (+), [79](#)

division (/), [79](#)

exponentiation (^), [80](#)

factorial (!), [80](#)

modulo (%), [79](#)

multiplication (*), [79](#)

square root (| /), [80](#)

subtraction (-), [79](#)

`max()` function, [145](#)

`median`, [87](#)

compared to average, [87](#), [217](#), [226](#)

with `percentile_cont()` function, [89](#)

Microsoft Access, [xxv](#)

Microsoft Excel, [xxv](#)

Microsoft SQL Server, [118](#), [237](#)

`BULK INSERT` command, [60](#)

Transact-SQL, [xxiv](#)

Microsoft Windows

Command Prompt

entering instructions, [366](#)

setup, [364](#), [366](#)

useful commands, [367](#)

file path declaration, [43](#), [62](#)

findstr command, [162](#)
folder permissions, [3](#)
min() function, [145](#)
mode, [91](#)
mode() function, [91](#)
modifying data, [164](#)
 for consistency, [170](#)
 updating column values, [169](#)
modulo, [79](#)
 testing for even numbers, [80](#)
multiplying numbers, [79](#)
MySQL, [xxv](#)
 LOAD DATA INFILE statement, [60](#)

N

NAICS (North American Industry Classification System), [197](#)
naming conventions
 camel case, [118](#)
 Pascal case, [118](#)
 snake case, [118](#), [120](#)
National Center for Education Statistics, [401](#)
National Center for Health Statistics, [404](#)
natural primary key, [121](#), [159](#)
New York City taxi data, [213](#)
 creating table, [213](#)
 exporting results, [216](#)
 finding busiest hour of day, [215](#)

importing, [214](#)
longest trip duration, [217](#)
normal distribution, [192](#), [226](#)
North American Industry Classification System (NAICS), [197](#)
NoSQL databases, [307](#)
NOT comparison operator, [35](#)
not-equal comparison operator
 `<>` syntax, [35](#)
 `!=` syntax, [35](#)
NOT NULL keywords, [132](#)
 adding to column, [165](#)
 removing from column, [133](#), [165](#)
now() function, [52](#), [208](#)
NULL keyword
 comparisons using IS DISTINCT FROM, [169](#)
 definition, [104](#)
 display in query results, [105](#)
 ordering with FIRST and LAST, [161](#)
 using in table joins, [104](#)
 using in WHERE clause, [104](#)
number data types, [44](#)
 decimal types, [47](#)
 decimal, [47](#)
 double precision, [48](#)
 fixed-point type, [47](#)
 fixed-point vs. floating-point types, [48](#)
 floating-point types, [48](#)

numeric, [47](#)
real, [48](#)
integer types, [45](#)
 bigint, [45](#)
 bigserial, [45](#)
 integer, [45](#)
 serial, [45](#)
 smallint, [45](#)
 smallserial, [45](#)
usage considerations, [50](#)

O

OGC (Open Geospatial Consortium), [277](#)
ON clause used with JOIN, [94](#)
one-to-many table relationship, [105](#), [128](#)
one-to-one table relationship, [105](#)
ON keyword used with DELETE CASCADE, [129](#)
Open Geospatial Consortium (OGC), [277](#)
operators
 addition (+), [79](#)
 comparisons with, [35](#)
 division (/), [79](#)
 exponentiation (^), [80](#)
 factorial (!), [80](#)
 JSON, [310](#)
 modulo (%), [79](#)
 multiplication (*), [79](#)

precedence, [81](#)
prefix, [81](#)
square root (`|/`), [80](#)
subtraction (`-`), [79](#)
suffix, [81](#)

Oracle, [xxiv](#)

ORDER BY clause, [32](#)

 ASC, DESC options, [32](#)
 with `count()` function, [148](#)
 on multiple columns, [33](#)
 specifying columns to sort, [32](#)
 specifying NULLS first or last, [161](#)
OR logical operator combining comparison operators, [38](#)
OVER clause with window functions, [193](#)

P

Pacific time zone, [52](#), [211](#)
padding character columns with spaces, [42](#), [44](#)
parentheses ()
 to designate order of operations, [38](#)
 to specify columns for importing, [70](#)
Pascal case, [118](#)
path deletion operator, JSON (#-), [331](#)
pattern matching
 using LIKE and ILIKE, [37](#), [171](#)
 with wildcards, [37](#)
Pearson correlation coefficient (r), [186](#)

percentage

percent change, [85](#)

creating user function, [347](#)

formula, [85](#), [113](#), [347](#)

of the whole, [84](#)

`percent_change()` user function, [347](#)

using with census data, [348](#)

percentile, [88](#), [224](#)

continuous vs. discrete values, [88](#)

definition of, [88](#)

`percentile_cont()` function, [88](#)

finding median with, [217](#)

in subquery, [224](#)

using an array to enter multiple values, [20](#)

`percentile_disc()` function, [88](#)

percent sign (%)

as modulo operator, [80](#)

wildcard for pattern matching, [37](#)

pgAdmin, [11](#)

alternatives to, [407](#)

connecting to database, [20](#), [22](#)

connecting to server, [12](#)

drag and drop objects, [32](#)

executing SQL, [20](#), [21](#)

geometry viewer, [324](#)

importing and exporting data, [74](#)

installation

Linux, [9](#)
macOS, [8](#)
Windows, [4](#)

keyword highlighting, [119](#)
launching, [11](#)
localhost, [13](#), [20](#)
master password, [11](#)
object browser, [12](#), [21](#), [24](#)
preferences, [15](#)
Query Tool, [14](#), [20](#)
.sql files, [3](#)
viewing data, [26](#), [31](#), [95](#), [144](#)
viewing tables, [65](#)
viewing table SQL statements, [24](#)
viewing text in results grid, [254](#)
views, [339](#)

pg_ctl utility, [395](#)
pg_dump utility, [395](#)
pg_reload_conf() function, [394](#)
pg_restore utility, [396](#)
pg_size_pretty() function, [389](#)
pg_total_relation_size() function, [389](#)
pipe character (|)
 as delimiter, [44](#), [63](#), [72](#)
 to redirect output, [295](#)

PL/pgSQL, [347](#), [350](#)

BEGIN ... END block, [356](#)

IF ... THEN statement, [356](#)

point, [66](#)

position() string function, [246](#)

PostGIS

creating spatial objects, [281](#)

data types, [280](#), [281](#)

displaying version, [276](#)

functions

ST_AsText(), [295](#)

ST_DFullyWithin(), [288](#)

ST_Distance(), [289](#)

ST_DWithin(), [287](#), [326](#)

ST_GeogFromText(), [283](#), [288](#), [326](#)

ST_GeometryType(), [300](#)

ST_GeomFromText(), [281](#)

ST_Intersection(), [301](#)

ST_Intersects(), [301](#)

ST_LineFromText(), [284](#)

ST_MakeLine(), [284](#)

ST_MakePoint(), [283](#), [324](#)

ST_MakePolygon(), [284](#)

ST_MpolyFromText(), [284](#)

ST_PointFromText(), [283](#)

ST_PolygonFromText(), [284](#)

ST_SetSRID(), [286](#), [324](#)

installation, [276](#)

Linux, [11](#)

macOS, [8](#)
Windows, [5](#)

loading extension, [276](#), [324](#)
shapefile, [292](#), [295](#), [383](#)
spatial joins, [300](#)

Postgres.app, [20](#)

PostgreSQL

- backup and restore, [395](#)
 - pg_dump, [395](#)
 - pg_restore, [396](#)
- command line usage, [363](#)
- comparison operators, [36](#)
- configuration, [387](#)
- creating functions, [346](#)
- default postgres database, [19](#)
- description of, [12](#), [19](#)
- documentation, [410](#)
- functions, [337](#)
- GUI tools, [407](#)
- importing from other database managers, [60](#)
- installation, [1](#), [3](#)
 - Linux, [9](#)
 - macOS, [8](#)
 - Windows, [4](#)
- JSON support, [308](#)
- maintenance, [387](#)
- modules, [237](#)

news and community, [409](#)
pg_operator table, [79](#)
postgresql.conf settings file, [393](#)
procedure, [349](#)
recovering unused space, [388](#)
settings, [392](#)
spatial data analysis, [275](#), [287](#), [289](#)
starting and stopping, [395](#)
statistics collector, [391](#)
table size, [388](#)
triggers, [337](#), [354](#)
utilities, tools, and extensions, [408](#)
version() function, [14](#)
views, [337](#)
postgresql.conf settings file, [210](#), [393](#)
editing, [393](#)
location of, [393](#)
reloading settings, [395](#)
precision input for numeric and decimal types, [47](#)
primary key, [18](#), [31](#)
composite, [121](#), [124](#)
definition of, [95](#), [121](#)
natural, [96](#), [121](#), [159](#)
surrogate, [122](#), [126](#)
 auto-incrementing, [125](#)
syntax, [99](#), [123](#), [124](#)
uniqueness, [96](#)

using auto-incrementing serial type, [46](#), [122](#)
violation, [124](#)

Prime Meridian, [66](#), [280](#)

procedural language, [347](#)

procedures

- CALL command, [352](#)
- difference with functions, [349](#)
- updating data with, [349](#)

projection (map), [279](#)

- Albers, [279](#)

psql command line application, [364](#)

- connecting to database, [370](#), [372](#)
- displaying table info, [379](#)
- editing queries, [375](#)
- executing queries from a file, [382](#)
- formatting results, [375](#)
- help commands, [372](#)
- importing and exporting files, [380](#)
- meta-commands, [379](#), [389](#)
- multiline queries, [374](#)
- NULL value display, [105](#)
- paging results, [376](#)
- parentheses in queries, [374](#)
- passing in SQL commands, [381](#)
- password file, [373](#)
- running queries, [374](#)
- saving query output, [381](#)

setup

Linux, [370](#)

macOS, [368](#)

Microsoft Windows, [364](#)

Public Libraries Survey, [140](#)

creating tables, [140–141](#)

Python programming language, [xxv](#)

creating PL/Python extension, [352](#)

in PostgreSQL function, [347](#), [352](#)

installation

macOS, [2](#)

Windows, [5](#)

Q

quantiles, [88](#)

quartiles, [90](#)

query

choosing order of columns, [31](#)

definition of, [17](#)

eliminating duplicate values, [34](#)

execution time, [135](#)

exporting results of, [73](#)

limiting number of rows returned, [67](#)

measuring performance with EXPLAIN, [135](#)

order of clauses, [39](#)

retrieving a subset of columns, [31](#)

selecting all rows and columns, [30](#)

quintiles, [90](#)

quotes, single vs. double, [25](#)

R

r (Pearson correlation coefficient), [186](#)

`rank()` function, [193](#)

ranking data, [193](#)

`rank()` and `dense_rank()` functions, [193](#)

 by subgroup, [195](#)

rates calculations, [196](#), [227](#), [235](#)

`record_if_grade_changed()` user function, [356](#)

REFERENCES keyword declaring foreign key, [128](#)

referential integrity

 cascading deletes, [129](#)

 foreign keys, [128](#)

 primary key, [124](#)

`regexp_match()` function, [255](#)

 extracting text from result, [260](#)

`regexp_matches()` function, [256](#)

`regexp_replace()` function, [252](#)

`regexp_split_to_array()` function, [252](#)

`regexp_split_to_table()` function, [252](#)

regular expressions, [247](#)

 capture group, [249](#), [257](#)

 escaping characters, [255](#)

 examples, [250](#)

 notation, [248](#)

parsing unstructured data, [253](#), [258](#)
regexp_match() function, [255](#)
regexp_matches() function, [256](#)
regexp_replace() function, [252](#)
regexp_split_to_array() function, [252](#)
regexp_split_to_table() function, [252](#)
with substring() function, [250](#)
in WHERE clause, [250](#)
relational database, [93](#)
 related tables, [18](#)
relational model, [93](#)
 reducing redundant data, [97](#)
 table relationships, [105](#)
replace() string function, [247](#)
reserved keywords, [119](#)
RETURNING clause, [170](#)
 with UPDATE statement, [167](#)
right() function, [247](#)
RIGHT JOIN keyword, [101](#)
ROLLBACK statement in transaction block, [177](#)
roots, square and cube, [80](#)
round() function, [86](#), [188](#)
row
 constructor, [328](#)
 counting, [143](#)
 in CSV file, [60](#)
 definition, [93](#)

- deleting, [175](#)
- generating from JSON array, [332](#)
- inserting, [25](#)
- recovering unused, [388](#)
- specifying in window function, [200](#)
- updating specific, [169](#)

`row()` function, [328](#)

R programming language, [xxv](#)

r-squared (coefficient of determination), [191](#)

S

- scalar subquery, [224](#)
- scale input for numeric and decimal types, [47](#)
- scatterplot, [186](#), [188](#)
- selecting all rows and columns, [30](#)
- SELECT statement
 - definition, [30](#)
 - order of clauses, [39](#)
- semicolon (;), [19](#)
- server
 - localhost, [20](#)
 - postgresql.conf* file, [210](#)
 - setting time zone, [210](#)
- SET keyword
 - clause in UPDATE statement, [166](#)
 - TIME ZONE, [211](#)
- set operators, [109](#)

EXCEPT, [111](#)

INTERSECT, [111](#)

UNION, [109](#)

UNION ALL, [109](#)

setting up your coding environment, [1](#)

shapefile, [291](#)

contents of, [292](#)

loading into database, [292](#)

shp2pgsql command line utility, [383](#)

US Census TIGER/Line, [292](#), [300](#)

SHOW command

config_file, [393](#)

data_directory, [395](#)

timezone, [209](#)

view all server settings with, [210](#)

shp2pgsql command line utility, [383](#)

simple feature standard, [277](#)

slope-intercept formula, [190](#)

snake case, [118](#)

sorting data, [32](#)

on aggregate results, [149](#)

by multiple columns, [33](#)

spatial data

area analysis, [296](#)

building blocks of, [276](#)

converting JSON to, [323](#)

data types, [280](#)

distance analysis, [287](#), [289](#)
finding location, [297](#)
geographic coordinate system, [276](#), [279](#)
geometries, [277](#), [281](#), [283](#), [284](#), [319](#)
intersection analysis, [301](#)
joins, [300](#)
projection, [279](#)
shapefile, [291](#)
simple feature standard, [277](#)
spatial reference system identifier (SRID), [277](#), [279](#)
well-known text (WKT), [278](#)
WGS 84 coordinate system, [280](#)
spatial reference system identifier (SRID), [277](#), [279](#)
 setting with `ST_SetSRID()`, [286](#)

SQL
 history of, [xxiv](#)
 indenting code, [27](#)
 math operators, [78](#)
 relational model, [93](#)
 reserved keywords, [119](#)
 style conventions, [23](#), [27](#), [56](#)
 using with external programming languages, [xxvi](#)
 value of using, [xxiv](#), [xxv](#)
 variations among databases, [xxiv](#)

`sqrt()` function, [81](#)
square root operator (`|/`), [81](#)
SRID (spatial reference system identifier), [277](#), [279](#)

setting with `ST_SetSRID()`, [286](#)
START TRANSACTION statement, [177](#)
statistical functions, [183](#)
 correlation with `corr()`, [186](#)
 dependent and independent variables, [186](#)
 linear regression, [189–190](#)
 `regr_intercept()` function, [190](#)
 `regr_slope()` function, [190](#)
 rates calculations, [196](#)
 rolling average, [198](#)
 standard deviation, [192](#)
 variance, [192](#)
string functions, [246](#)
 case formatting, [246](#)
 character information, [246](#)
 `char_length()`, [246](#)
 extracting and replacing characters, [247](#)
 `initcap()`, [246](#)
 `left()`, [247](#)
 `length()`, [246](#)
 `lower()`, [246](#)
 `position()`, [246](#)
 removing characters, [247](#)
 `replace()`, [247](#)
 `right()`, [247](#)
 `to_char()`, [219](#)
 `trim()`, [247](#)

`upper()`, [246](#)

subquery

correlated, [224](#)

with `crosstab()` function, [239](#)

definition, [223](#)

in `DELETE` statement, [225](#)

expressions, [229](#)

generating column with, [228](#)

`IN` operator expression, [230](#)

with `LATERAL`, [231](#)

scalar, [224](#)

uncorrelated, [224](#)

in `WHERE` clause, [224](#)

`substring()` function, [250](#)

subtracting numbers, [79](#)

across columns, [82](#)

`sum()` function, [86](#)

example on joined tables, [149](#)

grouping by column value, [152](#)

summarizing data, [139](#)

surrogate primary key, [122](#)

creating, [126](#)

T

tab character as delimiter, [63](#)

table

add column, [165](#), [168](#)

aliases, [107](#), [226](#)
alter column, [165](#)
autovacuum, [390](#)
constraints, [23](#)
creating, [22](#)
dead tuples, [388](#)
definition of, [17](#)
deleting column, [165](#), [176](#)
deleting data, [175](#)
derived table, [225](#)
design best practices, [117](#)
dropping, [176](#)
holds data on one entity, [23](#)
indexes, [133](#)
inserting rows, [25](#)
key columns, [24](#)
modifying with ALTER statement, [164](#)
naming, [120](#)
querying multiple tables using joins, [27](#)
relationships, [17](#)
size, [388](#)
temporary tables, [71](#)
viewing data, [26](#)
`tablefunc` module, installing for `crosstab()`, [237](#)
table relationships
 many to many, [105](#)
 one to many, [105](#), [128](#)

one to one, [105](#)

TABLE statement, [31](#)

telling your data's story, [399](#)

asking why, [405](#)

assessing the data's origins, [402](#)

building your own database, [401](#)

communicating your findings, [405](#)

consulting the data's owner, [403](#)

documenting your process, [400](#)

gathering your data, [400](#)

identifying trends over time, [403](#)

interviewing the data with queries, [402](#)

starting with a question, [400](#)

temporary table

declaring, [71](#)

removing with `DROP TABLE`, [71](#)

text

case formatting, [246](#)

concatenation, [172](#)

editors, [2](#)

escaping characters, [255](#)

extracting and replacing characters, [247](#)

formatting with functions, [246](#)

matching patterns with regular expressions, [247](#)

removing characters, [247](#)

text data type, [42](#)

text editors, [2](#)

TEXT file format, [63](#)

text qualifier

ignoring delimiters with, [61](#)

specifying with QUOTE option in COPY, [63](#)

tilde-asterisk case-insensitive matching operator (~*), [250](#)

tilde case-sensitive matching operator (~), [250](#)

time, matching with regular expression, [249](#)

time data types

interval, [51](#)

time, [51](#)

timestamp, [51](#), [204](#)

timestamp, [51](#), [204](#)

calculations with, [212](#)

creating from components, [207](#), [261](#)

extracting components from, [205](#)

formatting display, [219](#)

retrieving current date and time, [208](#)

subtracting to find interval, [219](#)

with time zone, [51](#)

within transaction, [208](#)

time zones

AT TIME ZONE keywords, [212](#)

automatic conversion of, [205](#)

including in timestamp, [51](#), [204](#), [262](#)

setting, [210](#)

setting server default, [394](#)

standard name database, [52](#)

viewing names of, [210](#)

viewing server setting, [209](#)

working with, [209](#)

`to_char()` function, [219](#)

`to_json()` function, [327](#)

`to_timestamp()` function, [320](#)

`to_tsquery()` function, [266](#)

`to_tsvector()` function, [265](#)

transaction block

`COMMIT`, [177](#)

 definition of, [177](#)

`ROLLBACK`, [177](#)

`START TRANSACTION`, [177](#)

 with time functions, [208](#)

 visibility to other users, [179](#)

triggers, [337](#), [354](#)

`BEFORE INSERT` statement, [359](#)

`CREATE TRIGGER` statement, [356](#)

`EXECUTE PROCEDURE` statement, [357](#)

`FOR EACH ROW` statement, [356](#)

`FOR EACH STATEMENT` statement, [356](#)

`NEW` and `OLD` variables, [356](#)

 testing, [357](#), [360](#)

`trim_county()` user function, [353](#)

`trim()` function, [247](#)

true as Boolean value, [94](#)

`TRUNCATE` statement, [176](#)

restarting identity sequence with, [176](#)

`ts_headline()` function, [269](#)

`tsquery` data type, [266](#)

`ts_rank()` function, [272](#)

`ts_rank_cd()` function, [272](#)

`tsvector` data type, [265](#)

U

uncorrelated subquery, [224](#)

underscore wildcard for pattern matching (`_`), [37](#)

`UNIQUE` constraint, [96](#), [131](#)

universally unique identifier (UUID), [55](#), [122](#)

as primary key, [122](#)

`unnest()` function, [91](#)

unstructured data, [245](#)

parsing with regular expressions, [253](#), [258](#)

`update_personal_days()` user function, [350](#)

UPDATE statement

with CASE, [262](#)

definition, [166](#)

PostgreSQL-specific syntax, [167](#)

with RETURNING clause, [167](#), [170](#), [262](#)

with SET clause, [166](#)

using across tables, [166](#), [173](#)

`upper()` function, [246](#)

USA Today, [xxiii](#)

US Census

American Community Survey, 2014–2018, [184](#)
description of columns, [184](#)
importing data, [184](#)

County Business Patterns, [197](#)

county population estimates, [63](#), [295](#)
adding and subtracting columns, [82](#)
description of columns, [65](#)
finding average county population, [87](#)
finding median county population, [89](#)
importing data, [64](#)
percent calculations with, [84](#)
performing calculations across tables, [112](#)

methodologies compared, [185](#), [402](#)

trade data, [200](#)

US Department of Agriculture, [158](#), [402](#)
farmers' market data, [285](#)

US Geological Survey, [318](#)

UTC (Coordinated Universal Time), [52](#), [206](#)
UTC offset, [52](#), [206](#), [219](#)

UUID (universally unique identifier), [55](#), [122](#)
as primary key, [122](#)

V

VACUUM command, [388](#)
ANALYZE option, [391](#)

autovacuum process, [390](#)
editing server setting, [393](#)

FULL option, [392](#)
monitoring table size, [388](#)
purpose of, [388](#)
running manually, [392](#)
time of last vacuum, [391](#)
VERBOSE option, [392](#)
VALUES clause with INSERT, [25](#)
varchar, [23](#), [42](#)
 also specified as character varying, [42](#)
version () function, [14](#)
views, [337](#)
 advantage of using, [338](#)
 creating, [338](#)
 deleting data with, [346](#)
 dropping, [339](#)
 inserting data with, [342](#), [344](#)
 LOCAL CHECK OPTION, [344](#)
 materialized, [338](#)
 queries in, [339](#)
 retrieving specific columns, [341](#)
 security_barrier option, [343](#)
 updating data with, [342](#), [345](#)

W

Wall Street Journal, [xxiv](#)
well-known text (WKT), [278](#)
 extended, [283](#)

order of coordinates, [278](#)
WHEN clause in CASE statement, [241](#)
WHERE clause, [35](#)
 in COPY statement, [70](#)
 with DELETE statement, [175](#)
 with EXISTS operator, [167](#)
 with IS NULL keywords, [161](#)
 with JSON operators, [316](#)
 with LIKE expression, [171](#)
 with regular expressions, [250](#)
 in UPDATE statement, [166](#)
whole numbers, [45](#)
wildcard
 percent sign (%), [37](#)
 underscore (_), [37](#)
window functions
 definition of, [193](#)
 OVER clause, [193](#), [220](#)
 PARTITION BY clause, [195](#)
 specifying rows, [200](#)
WITH keyword
 defining common table expression, [234](#)
 options with COPY, [44](#), [62](#)
WKT (well-known text), [278](#)
 extended, [283](#)
 order of coordinates, [278](#)
working tables, [176](#)

X

XML, [55](#)

Z

ZIP codes, [163](#)

loss of leading zeros, [163](#)