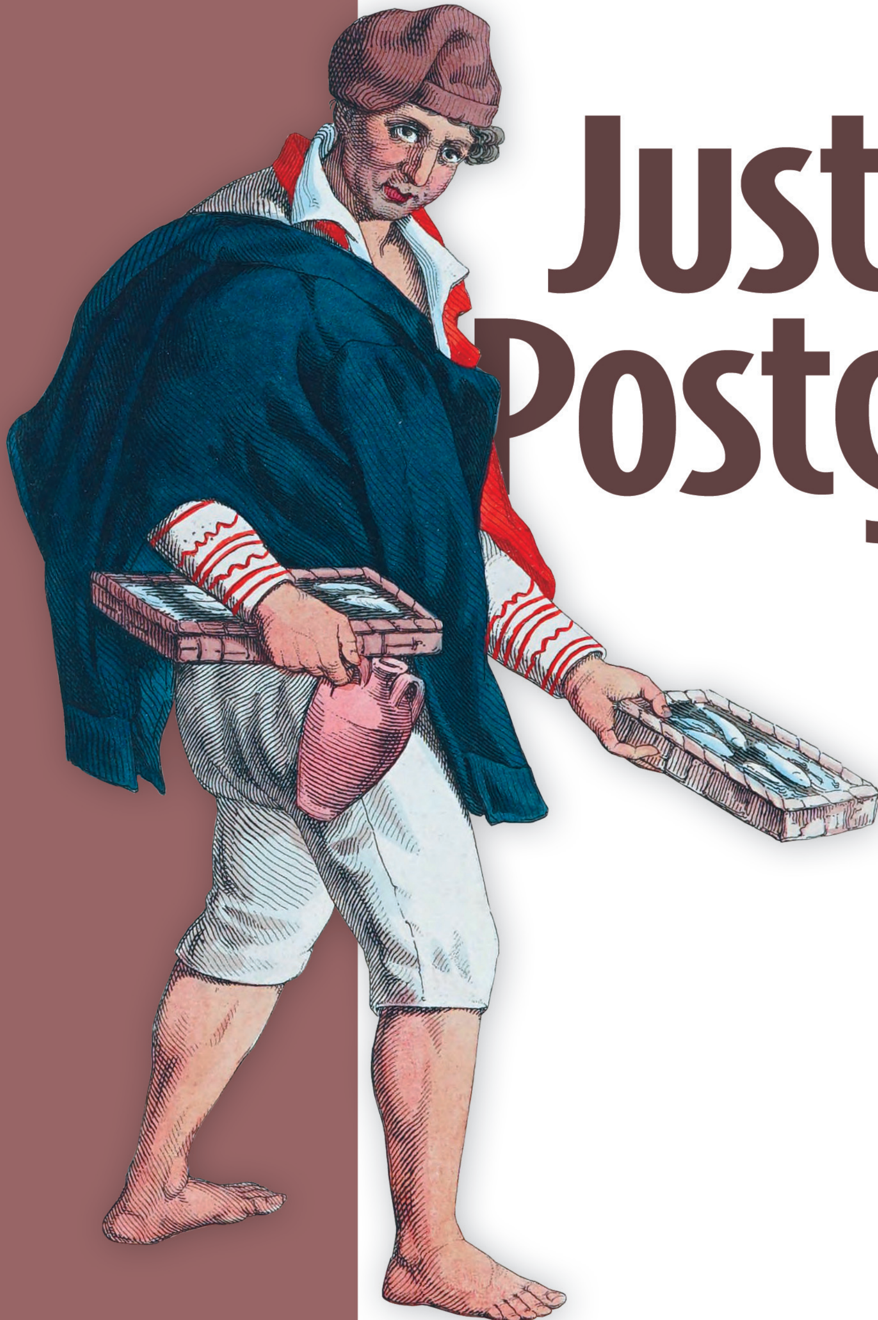
All the database you need

Just Use Postgres!



Denis Magda

Foreword by Josh Long
Afterword by Vlad Mihalcea

MANNING

Some essential Postgres extensions for developers

Category	Extensions	Description
Postgres Beyond Relational (Extensions that let you use the database far beyond the traditional relational workloads)	`pgvector`	The extension that turns Postgres into a vector database by storing vector embeddings, performing similarity searches, and providing specialized indexes for vectorized data.
	`pgai`	Lets you generate embeddings, call LLMs, and run full RAG workflows directly inside Postgres without separate services.
	`pgvectorscale`	Complements the `pgvector` extension by enabling cost-efficient and high-performance vector search at scale through StreamingDiskANN index, statistical binary quantization, and other optimizations.
	`TimescaleDB`	Enables Postgres for storing and analyzing large volumes of time-series and event data.
	`PostGIS`	Transforms Postgres into a database capable of storing, indexing, and querying geospatial data.
	`pgmq`	Allows to use Postgres as a lightweight message queue.
	`pg_duckdb`	Enables high-performance analytical workloads for Postgres by integrating with DuckDB's columnar-vectorized storage engine.
Programming and Procedural Languages (Extensions that let you create database functions in your preferred programming language)	`PLV8`	Embeds V8 JavaScript engine into Postgres, allowing to write JavaScript-based database logic and call it from SQL.

	`PL/Java`
	`PL/Python`
	`PL/Rust`
	`PL/dotnet`
	`PL/PHP`
	`PL/Perl`
	`PL/R`
	`PL/Haskel`
		Extensions that let you implement database logic using various major programming languages, including Java, C#, Python, Rust, and more.

(continues on the inside back cover)

Just use Postgres!

All the database you need

Denis Magda

Foreword by Josh Long
Afterword by Vlad Mihalcea

MANNING
Shelter Island

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

© 2026 Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.

Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Rebecca Senninger
Technical editors: Josephine Bush, Stefan Papp
Review editor: Dunja Nikitović
Production editor: Aleksandar Dragosavljević
Copy editor: Tiffany Taylor
Proofreader: Mike Beady
Typesetter: Tamara Švelić Sabljić
Cover designer: Marija Tudor

ISBN 9781633435698
Printed in the United States of America

To Postgres and its passionate users who just use it

contents

foreword *ix*
preface *xi*
acknowledgments *xiii*
about this book *xv*
about the author *xix*
about the cover illustration *xx*

PART 1 POSTGRES AS A RELATIONAL DATABASE 1

1 *Meeting Postgres* *3*

1.1 Why is Postgres so popular? 4
1.2 “Just use Postgres!” explained 6
1.3 Starting Postgres in Docker 6
1.4 Connecting with psql 10
1.5 Generating mock data 11
1.6 Running basic queries 15

2 *Standard RDBMS capabilities* *18*

2.1 Creating the database structure 19
Creating databases *20* ▪ *Creating schemas* *22*
Creating tables *25*
2.2 Querying and manipulating data 27

2.3 Data integrity 29
Constraints 29 ▪ Foreign keys 32

2.4 Transactions 36
*Implicit transactions 36 ▪ Explicit transactions 37
Multiversion concurrence control 39*

2.5 Joins 42

2.6 Functions and triggers 44
Functions: A practical example 45 ▪ Triggers: A practical example 51

2.7 Views 55

2.8 Roles and access control 59

3 Modern SQL 65

3.1 What is modern SQL? 66

3.2 Loading the music service dataset 67

3.3 Common table expressions 68
Selecting data with CTEs 69 ▪ Using multiple CTEs in a query 70 ▪ Modifying data with CTEs 73

3.4 Recursive queries 76
Querying hierarchical data 78 ▪ Using arguments in recursion 80

3.5 Window functions 82

4 Indexes 89

4.1 Why are indexes so popular? 90

4.2 Overview of Postgres index types 92

4.3 Loading the multiplayer game dataset 93

4.4 Learning to use the EXPLAIN statement 95

4.5 Single-column indexes 99
Single-column B-tree indexes 99 ▪ Single-column hash indexes 103

4.6 Composite indexes 107
*Considering an additional single-column index 108
Creating a composite index 111 ▪ Caveats of composite indexes 113*

4.7 Covering indexes 117
4.8 Partial indexes 119
4.9 Functional and expression indexes 122

PART 2 CORE POSTGRES BEYOND RELATIONAL 127

5 *Postgres and JSON 129*
5.1 Storing JSON data 130
5.2 Loading the pizza order dataset 131
5.3 JSON in Postgres: Striking the balance 134
5.4 Querying JSON data 136
Extracting fields with the -> and ->> operators 137 ▪ Using the ? operator to check for the presence of a key 140 ▪ Comparing objects with the @> operator 141 ▪ Using JSON path expressions 143
5.5 Modifying JSON data 147
5.6 Indexing JSON data 150
Using an expression index with a B-tree 151 ▪ Using GIN indexes 154

6 *Postgres for full-text search 164*
6.1 Basics of full-text search in Postgres 165
Tokenization and normalization 166 ▪ Full-text search configurations 167
6.2 Preparing data for text search 171
Generating lexemes with the to_tsvector function 171
Storing tsvector lexemes in the database 173
6.3 Performing full-text search 177
Using plainto_tsquery for simple queries 177 ▪ Using to_tsquery for advanced filtering 179
6.4 Ranking search results 182
6.5 Highlighting search results 188
6.6 Indexing lexemes 191
Using GIN indexes 192 ▪ Using GiST indexes 194

PART 3 EXTENSIONS AND THE BROADER ECOSYSTEM .. 199

7 *Postgres extensions 201*

7.1 The roots of Postgres extensibility 201

7.2 Getting started with extensions 202

Exploring available extensions 203 ▪ Installing and using extensions 205

7.3 Essential extensions for developers 209

7.4 Postgres-compatible solutions 211

8 *Postgres for generative AI 213*

8.1 How to use Postgres with gen AI 214

Postgres and LLMs 214 ▪ Postgres and embedding models 216

8.2 Starting Postgres with pgvector 218

8.3 Generating embeddings 220

Generating embeddings for movies 221 ▪ Loading the final dataset into Postgres 223

8.4 Performing vector similarity search 226

Using cosine distance for similarity search 227
Changing the search phrase for better results 230

8.5 Indexing embeddings 232

Using IVFFlat indexes 234 ▪ Using the HNSW index 240

8.6 Implementing RAG 245

Preparing the environment for the prototype 246 ▪ Interacting with the LLM 247 ▪ Retrieving context for LLM 248
Using RAG to answer questions 249

9 *Postgres for time series 253*

9.1 How Postgres works with time-series data 254

9.2 Starting Postgres with TimescaleDB 257

9.3 Loading the time-series data 259

9.4 Exploring TimescaleDB hypertables 262

9.5 Analyzing time-series data 266

Using the time_bucket function 267 ▪ Using the time_bucket_gapfill function 271

9.6 Using continuous aggregates 274

Creating and using aggregates 274 ▪ Refreshing aggregates 277

9.7 Indexing time-series data 279
Using a B-tree index 281 ▪ Using BRIN indexes 284

10 Postgres for geospatial data 292

10.1 How Postgres works with geodata 293
Built-in Postgres capabilities 295 ▪ Extended capabilities with PostGIS 296

10.2 Starting Postgres with PostGIS 297

10.3 Loading the OpenStreetMap dataset 300
Exploring tables with points 302 ▪ Exploring tables with ways 303 ▪ Exploring table with polygons 304

10.4 Visualizing geospatial data 305

10.5 Querying geospatial data 307
Working with points 309 ▪ Working with polygons 314 Working with line segments 318

10.6 Indexing geospatial data 320
Understanding GiST structure 321 ▪ Using a GiST index 325

11 Postgres as a message queue 332

11.1 When to use Postgres as a message queue 333

11.2 Building a custom message queue 335

11.3 Using a custom queue 339

11.4 Using LISTEN and NOTIFY 343

11.5 Queue implementation considerations 346
LISTEN and NOTIFY considerations 346 ▪ Indexing considerations 347 ▪ Partitioning considerations 348 Messages processing failover considerations 350

11.6 Starting Postgres with pgmq 350

11.7 Using pgmq 352
Creating and using a visitors queue 353 ▪ Using message visibility timeouts 354

appendix A Five optimization tips 358

appendix B When not to use Postgres 369

afterword 371

index 373

foreword

The only way to have a friend is to be one.

—Ralph Waldo Emerson

This may seem like an unusual foreword for me to write, especially considering my reputation as a member of the Spring team. Spring is an application development framework utilized by millions to create robust systems and services on the Java Virtual Machine. Although I appreciate the nuances of Postgres, I am an application developer, not a database administrator or database engineer who would typically be invited to write a foreword for a database-related book. But this book is written by a developer for developers.

I fell in love with Postgres long ago because of all the amazing tools (and, let's be honest, awe-inspiring toys) it offered me as an application developer and programmer. I loved it from very early on, but sometimes it felt like I was alone.

Today, Postgres is a de facto standard in the industry. As the saying *should* go, "Nobody ever got fired for deploying Postgres." But let me tell you, being a Postgres enthusiast hasn't always been easy.

I began my professional journey as a software engineer in 2002, a time when MySQL roamed the earth. It was MySQL, not Postgres, that was the go-to database of choice. MySQL was the emerging standard. It was the *M* in the *LAMP Stack*, after all! Its popularity was undeniable.

And as appealing as Postgres was for me, an application developer, application developers aren't the only ones who get to choose a database. There were several reasons MySQL was in vogue:

- *MySQL was easy to get up and running for development.* It was preinstalled on every Linux distribution I could lay my hands on, and it was the go-to option for open source projects. It ran on everything, even Windows. Postgres, on the other hand, didn't even have a Windows build until 2005, almost eight years after MySQL did.

- *MySQL was easy to run in production.* MySQL was easy to configure, it had solid administration tools, and it had relatively straightforward built-in asynchronous replication. Postgres didn't even have native replication until version 9.0!
- *MySQL was battle-tested by large-scale web applications.* Did you know that Google used MySQL as the engine for AdWords (now called Google Ads)? That's right, the Google "cash register" was powered by MySQL! Amazing! And Facebook used MySQL as a fundamental part of its core infrastructure. Who wouldn't want to use such a database? It's the best! It's "we scale"! Or so everyone kept reminding me. And Postgres? Only the brave dared to use it for large-scale workloads.

MySQL was all the rage, it was the only one folks felt like they could run in production, and it was the only one developers could readily get access to. (Remember, 2002 was long before Docker and long before developers ran virtual machines locally.) It wasn't easy being Postgres's friend!

But all of that is in the past now, and here we are: in 2025, surfing the tidal wave of Postgres's popularity. It's finally happened: Postgres is the friend we always knew it could be, not just for application developers, but for operations, too. It's easy to get started using it, and it's easy to deploy and scale. For the person looking at Postgres today, the questions that should be top of mind are those same things that mesmerized me decades ago: What can you build with Postgres? And how?

I'll answer the first question: What can you build with Postgres? Just about anything you want!

And the second question: How can you build with Postgres? You'll get no better or more pragmatic an answer than by reading the book in your hands right now.

Denis Magda has written the developer's guide to Postgres for *today*. The one where we can take for granted how easy it is to set up, deploy, and build on Postgres. The one where Postgres is as good a friend to me as I was to it. The Postgres that enjoys the largest ecosystem of extensions, plugins, and integrations of any modern database, by far. The Postgres that has been hacked, sliced, diced, and improved in a million different ways by countless contributors in the space.

Denis knows Postgres inside and out, having spent a lot of time working at different layers of Postgres's evolution. Denis knows how to build anything on Postgres, and this is his recipe book. I have been following this book's evolution for more than a year, waiting eagerly for the updates to the manuscript, because the chapters always teach me something new and remind me of the exciting reasons I fell in love with Postgres in the first place as an application developer.

It's easy to be a Postgres friend today, and this book will help you quickly strike up a friendship that lasts.

Thanks, Denis.

—Josh Long
Application developer and big Postgres fan
Spring Developer Advocate, Java Champion, Microsoft MVP,
Kotlin GDE Alumnus, etc.

preface

It was yet another hot sunny day in Florida when I met Alex, a close friend of mine, for lunch. Although we lived in the same city, it had been a while since we last met. With more kids and mounting responsibilities, meetings like this had become rare. We had less than an hour to share news about the summer with our kids, the quiet hurricane season, and work projects.

We are both in tech, so it's not surprising that most of our time was spent discussing work-related matters. And Alex told me about one challenge he had been dealing with, which eventually convinced me that this book had to be written:

> *Alex:* I have to migrate that application to a public cloud by the end of the month, and the problem is the database.
>
> *Denis:* What's wrong with the database?
>
> *Alex:* Our database administration team doesn't manage this specific database in the cloud. So, if I migrate the database there, it's going to be me who needs to take care of all the administration and maintenance tasks.
>
> *Denis:* Which databases do they provide support for in the cloud?
>
> *Alex:* Postgres is one option.
>
> *Denis:* Why can't you use Postgres then? Just rewrite the data layer of the app. You've done that more than once.
>
> *Alex:* Well, the application's data model is built around documents. It stores and exchanges data in the JSON format. The current database fits exceptionally well for this use case. Switching to Postgres would mean

transitioning to the relational model, which would require giving up the flexibility of the JSON-based document model.

Denis: Postgres supports JSON. You can continue benefiting from the documents-based design.

Alex: Silence. He looked at me, puzzled.

Denis: I'm not joking. JSON is a first-class citizen in Postgres. The database stores it efficiently, provides special functions and operators for querying and modifying document structure, and comes with specialized index types that make searches through the documents fast.

Alex: This is great! Looks like a lot has changed since the last time I used Postgres …

That exact conversation convinced me this book needed to be written. Alex was not new to Postgres. We both started using the database more than 15 years ago while working together on high-load web applications. We knew how to use it—at least, its relational capabilities.

After finishing lunch and giving my friend a hug, I asked myself the following question: "If Alex, a senior-level technical guru, didn't know what Postgres is capable of these days, then how many software engineers who are entering the tech space or have worked in it for years don't know what Postgres is truly capable of?" So, I began thinking about the book idea. I took notes and jotted down ideas. Most importantly, I kept meeting people who were both surprised and happy to discover what Postgres offers now. That continued until one day I jumped on a video call with Jonathan Gennick, an acquisition editor at Manning … and now you're reading the result of that conversation.

acknowledgments

This book lists me as the author, but that doesn't mean it is solely my book. Only through the implicit and direct contributions of dozens of people did the book come into existence. Without their support, inspiration, and collaboration, I would not have started or finished this long project. Writing a book is hard to begin and even harder to finish, which makes the help and involvement of others crucial.

First, I would like to thank all *those who inspired me* by showing the power of teaching and education. My dear Grandma Lubov Krivobok, a world-class chemistry teacher who continues to work with high school students in her late 80s, preparing them for high-ranked medical universities and academies; and my dear aunt, Oksana Semochkina, and uncle, Alexander Semochkin, both PhDs in mathematics, who dedicated themselves to raising the next generation of mathematicians and software engineers: you three showed me that true teaching is a craft and that if someone decides to share their knowledge, they must do so with complete dedication.

Second, I would like to thank *those who supported me* throughout this project: my lovely wife, Elena, and our two adorable but mischievous boys, Lev and Chris. This book was a sacrifice for all of us. We didn't have a proper vacation. We could have spent much more time together. But we'll catch up. The book helped me realize that family is even more important to me than I thought. I love you.

Third, I want to thank those *who worked with me* on the book and who will continue to do so even after it's published. To the exceptional Manning team, thank you! I'm grateful to Jonathan Gennick, who listened to my book proposal, believed in it, and gave me the chance. I owe a great deal to Rebecca Senninger, Tiffany Taylor, Mike Beady, and many others who worked with me on the chapters and helped make them much better. I want to thank Josephine Bush, a Microsoft Data Platform MVP, and Stefan Papp, an

author and university educator, who worked as technical editors on the book. Finally, I am thankful to the many independent experts and professionals who volunteered to read the book drafts and provided invaluable feedback: Ajinkya Kadam, Akshay Phadke, Al Pezewski, Alessandro Puzielli, Alex Elistratov, Anil Kumar Moka, Annamalai Muruganathan, Benjamin Lis, Brandon Friar, Charly Batista, Deep Bodra, Diego Alonso, Etienne de Maricourt, Frits Hoogland, Garry Offord, Gayathri Vijayan, Gregorio Piccoli, Jeff Marcus, Johannes Lochmann, John Harrop, Josh Kelley, Lukas Vileikis, Nadir Doctor, Nate Clark, Philip Patterson, Ravi Kiran Magham, Ron Lease, Ronald Goodwin, Ronald Harmsen, Saurabh Aggarwal, Sophia Willows, Vlad Mihalcea, and William LeBorgne. Your suggestions helped make this a better book.

Finally, thank you, *dear reader*, for choosing this book and dedicating your time to learning Postgres. I hope our teamwork will meet your expectations!

about this book

In 2009, I graduated from university and landed my first full-time job as a Java developer. Our team chose Postgres for a high-load web application I had yet to build. In those days, Postgres lacked many features, and few people considered using it. Still, we placed our trust in it, and since then it has never let me down.

Over the years, I changed jobs and companies, but Postgres remained my default database. I continued learning it gradually while using it on various projects, although I mostly saw it as a relational database for transactional workloads.

In early 2022, I joined the Yugabyte team, which builds a distributed version of Postgres. That's when I discovered a different Postgres: one that supports full-text search, powers geospatial applications, and enables generative AI. I realized Postgres could do much more than I had expected from it as a relational database.

I rediscovered Postgres by reading countless blog posts, watching endless videos, and having many conversations with community members. At some point, I asked myself why there wasn't a book to introduce the breadth and depth of Postgres capabilities. It felt strange that such a book didn't exist, given the database's growing popularity. Eventually, I decided to contribute to Postgres by writing that book myself.

This book introduces you to the core and extended Postgres capabilities in a practical manner. Think of every chapter as a dedicated hands-on developer guide. Each chapter is full of practical examples that let you study specific capabilities in action. You're expected to start Postgres and run code samples as you read. This matters, because once you finish a chapter, I want you to have this feeling: "Excellent! Postgres can really do *X*, and I've just experienced it myself! I did that on my own machine!" where *X* is the capability the chapter focuses on.

Finally, I'd like to set clear expectations about what this book does and doesn't cover:

- The book doesn't cover all existing Postgres capabilities. Instead, it walks you through some of the most prominent features that are widely used. And the book does this at a fast pace, because the goal is not to learn all the ins and outs of a particular capability, but to introduce you to it and make sure you know how to start using it in your applications.
- This book is not an in-depth guide on deployment, optimization, scalability, security, or other best practices. Each chapter could be a book on its own, but I've kept them to 30–40 pages so you can finish them in a few evenings. More importantly, I've purposefully avoided going too deep into details so you can enjoy the process, run the code, see results, feel successful, and trust Postgres for the use case. Once you've chosen Postgres for the use case, you can always dive deeper by referring to other resources.
- This book doesn't compare Postgres to other databases or data platforms. That's deliberate: the focus is on exploring Postgres, regardless of how it stacks up against other technologies.

Who should read this book

The primary audience for this book is application developers, software engineers, and architects—those who are responsible for designing and building applications. Database administrators may find it less valuable, because the book doesn't focus on Postgres deployment, configuration, and maintenance best practices and aspects.

The book suits both newcomers to Postgres and experienced users. If you're new to Postgres, you should know the basics of SQL and relational databases. At a minimum, you should know how to create tables, populate them with data, and modify or query records using SQL. If you're an experienced Postgres user, you can broaden and deepen your understanding of the database by reading the chapters that interest you most.

How this book is organized: A roadmap

This book is divided into 3 parts, 11 chapters, and 2 appendices, each highlighting distinct Postgres capabilities. The first part comprises four chapters that cover core features common to most relational database management systems. The second part examines core Postgres capabilities that go beyond relational workloads: JSON support and full-text search. The third part focuses on Postgres extensions that enable the database to handle generative AI, time series, and other scenarios.

Here is a brief overview of each chapter and appendix:

- Chapter 1 provides a quick introduction to Postgres and explains why it became so popular. The chapter also demonstrates how to start Postgres in Docker, connect to the database, generate mock data, and run a few basic SQL queries.
- Chapter 2 explores widely used Postgres capabilities commonly supported by relational databases. You learn how to design database structure; perform queries

and updates; use transactions, triggers, and views; maintain data integrity; and secure data access.

- Chapter 3 covers modern SQL capabilities of Postgres, including common table expressions, recursive queries, and window functions.
- Chapter 4 discusses various Postgres index types and shows how to optimize performance with single-column, composite, partial, covering, and functional indexes.
- Chapter 5 examines Postgres's JSON capabilities by showing how to store JSON objects in the database, query and modify JSON structures with built-in operators and functions, and optimize searches with GIN and B-tree indexes.
- Chapter 6 demonstrates how to use Postgres's full-text search capabilities by converting textual data into lexemes, performing the search with specialized functions and operators, and optimizing performance with GIN and GiST indexes.
- Chapter 7 introduces the Postgres extension ecosystem, discusses different types of extensions, and shows how to use one extension that comes preinstalled with the standard Postgres distribution.
- Chapter 8 explores Postgres's capabilities for generative AI, demonstrates how to use the `pgvector` extension to store and query vector embeddings, and shows how to implement retrieval-augmented generation and optimize performance with HNSW and IVFFlat indexes.
- Chapter 9 highlights Postgres's capabilities for time-series workloads, shows how to use the TimescaleDB extension to manage and analyze time-series data, and demonstrates how to optimize queries with B-tree and BRIN indexes.
- Chapter 10 covers the PostGIS extension that enables Postgres for geospatial workloads by allowing storage and querying of location-based data, visualizing geographic information with specialized tools, and optimizing searches with spatial indexes backed by GiST, SP-GiST, or BRIN.
- Chapter 11 explains when, why, and how to use Postgres as a message queue. It demonstrates how to implement a custom queue using Postgres's built-in capabilities or use the `pgmq` extension, which provides a ready-to-use message queue implementation.
- Appendix A provides a quick overview of my top five query optimization tips to help you write more efficient code and design better solutions with Postgres.
- Appendix B gives you some guidance on when you should refrain from using Postgres.

If you're new to Postgres, I suggest starting with the first four chapters, which cover capabilities common to most relational databases. After that, feel free to jump to any chapter that interests you.

If you're an experienced Postgres user, you can pick any chapter you like. If you plan to explore chapters 2–7, briefly review the beginning of chapter 1, which shows how to start a Postgres instance in Docker that is reused in those chapters.

About the code

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (➥).

You can get executable snippets of code from the liveBook (online) version of this book at https://livebook.manning.com/book/just-use-postgres. The complete code for the examples in the book is available for download from the Manning website at https://www.manning.com/books/just-use-postgres and from GitHub at https://github.com/dmagda/just-use-postgres-book.

liveBook discussion forum

Purchase of *Just use Postgres!* includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to https://livebook.manning.com/book/just-use-postgres/discussion.

Manning's commitment to our readers is to provide a venue where meaningful dialogue between individual readers and between readers and authors can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest that you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible on the publisher's website for as long as the book is in print.

Author online

Denis Magda can be found online at the following platforms:

- *Twitter:* https://x.com/denismagda
- *LinkedIn:* https://www.linkedin.com/in/dmagda/
- *YouTube:* https://www.youtube.com/@DevMastersDb
- *Blog:* https://medium.com/@magda7817
- *GitHub:* https://github.com/dmagda

about the author

Denis Magda is a software engineer who started his career at Sun Microsystems and Oracle, working on the Java platform and leading one of the Java development teams. After mastering Java from the inside, he ventured into the world of Postgres and other databases, where he has stayed ever since.

about the cover illustration

The figure on the cover of *Just use Postgres!* is "Pescivendolo," or "A fish vendor," taken from *Usi e costumi di Napoli e contorni descritti e dipinti* by Francesco de Bourcard, published in 1853.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

Part 1

Postgres as a relational database

In this part of the book, we explore core Postgres capabilities that define it as a relational database management system. These are the capabilities that most people are aware of and that have been used for decades.

We start with an introduction to Postgres and learn how to run it in Docker on our machine. From there, we explore its core relational capabilities by designing a database structure and manipulating data. Finally, we dive into modern SQL features and learn how to optimize query performance using different Postgres index types. After completing the first part of the book, we'll be ready to use Postgres as a relational database for designing and building applications for transactional and other types of workloads.

Meeting Postgres

This chapter covers

- Understanding what "just use Postgres" means
- Starting Postgres and connecting from a command-line tool
- Generating mock data using built-in database capabilities and running a few basic queries

PostgreSQL (or simply Postgres, as most people call it) is one of the fastest-growing relational and general-purpose databases. As a *relational database*, it natively supports Structured Query Language (SQL) and is a popular choice for online transaction processing (OLTP) workloads that process user requests with low latency while guaranteeing data consistency and integrity even if a database server crashes in the middle of an operation execution. You encounter OLTP systems daily when you, for example, book a flight to your next vacation destination, pay for groceries at a local supermarket, or share a new post on social media.

Over the years, Postgres has evolved into a *general-purpose database* that supports use cases beyond traditional transactional workloads. Postgres has a broad ecosystem of extensions and other derived solutions that make it capable of handling full-text

search, time-series, generative AI, geospatial, analytical, and other workloads. Chances are high that you've already interacted with Postgres while analyzing stock market fluctuations over different time periods in a financial application, finding the nearest coffee shop through a geolocation service, or pair-programming with an AI coding assistant that updates and retrieves relevant context from the database. With that in mind, let's start exploring the depth and breadth of Postgres capabilities so you're ready to "just use Postgres" for the applications you build.

1.1 Why is Postgres so popular?

There are hundreds of databases nowadays. The DB-Engines website (https://db-engines.com/en/ranking) ranks and categorizes more than 420 database management systems, with Postgres ranked as the fourth most popular option for many years in a row.

DB-Engines uses various metrics to calculate the scores of ranked databases. It measures the number of mentions of a database on websites across the internet and social networks, the number of discussions on Stack Overflow and DBA Stack Exchange, and more to compute the most accurate ranking possible.

With scores in place for each listed database, the website allows you to build trend charts comparing the popularity of databases you're interested in. Figure 1.1 shows a chart comparing the ranks of Oracle, MySQL, Microsoft SQL Server, Postgres, and MongoDB since 2013. The trend shows that Postgres has been gaining momentum, while the popularity of the top three leaders has either stagnated or declined.

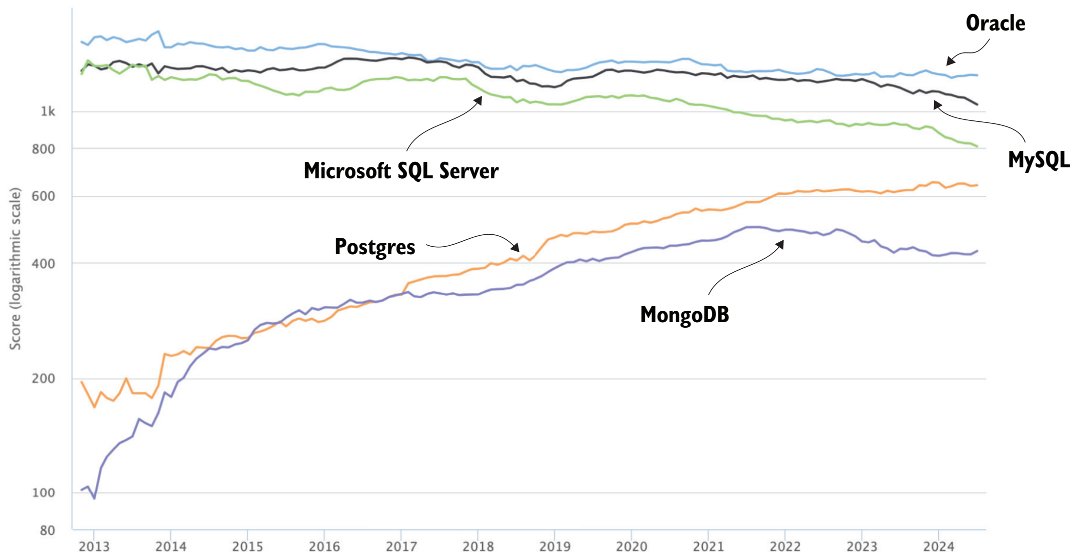


Figure 1.1 DB-Engines ranking showing Postgres trending up in popularity (Photo courtesy of DB-Engines, https://db-engines.com)

Apart from DB-Engines, Stack Overflow surveys thousands of developers annually to measure the popularity of various programming languages, databases, tools, and other technologies. The developers voted for Postgres as the most popular, admired, and desired database three years in a row—2023, 2024, and 2025 StackOverflow Developer Surveys—a trend that's likely to continue well after this book is published.

What can explain the growing popularity of Postgres among developers and other technical experts? The following three factors seem to contribute the most:

- *Postgres is open source and governed by the community.* Postgres started as a research project at the University of California, Berkeley, led by Michael Stonebraker, a computer scientist and Turing Award winner specializing in database systems. The Berkeley team released the database to open source in 1994 under the MIT license. A few years later, in 1996, the PostgreSQL Global Development Group was established to oversee and lead the project development. Since then, Postgres has remained fully open and governed by the global community. No single vendor has full control of the project. Postgres is built and led by the open source community.
- *Postgres is enterprise-ready.* Postgres has gained a reputation as one of the most reliable and robust databases throughout its 35 years of development. A new major Postgres version is released annually, but the community avoids drastic changes to the database engine and focuses on incremental improvements instead.

  This development approach has benefits, resulting in fewer regressions and instabilities between major database releases. In addition to the quality of the database, a significant number of vendors, cloud providers, and consultants offer services and paid solutions, which is an important factor for companies seeking enterprise-grade support.
- *Postgres is extensible by design.* Although the community takes a relatively conservative approach to the Postgres core development, a lot of innovation happens in the ecosystem of Postgres extensions and other derived solutions.

  Michael Stonebraker defined *extensibility* as one of the main design goals in his original "The Design of Postgres" paper, which was published during the early days of Postgres. In that paper, he referred to the capability using the term *extendibility*, which was his preferred wording at the time:

  > *Provide user extendibility for data types, operators and access methods.*

  Since then, the rich ecosystem of extensions has served Postgres well. Hundreds of extensions simplify its operations, bring in new capabilities, and let us use Postgres as a general-purpose database that can efficiently query JSON documents, handle time-series data, perform full-text and vector similarity searches, and even tackle analytical workloads.

Considering the growing popularity of Postgres, which relies on its robustness, extensibility, and open source nature, does this mean that Postgres can tackle all types of

workloads and use cases? Let's understand why the book is titled *Just Use Postgres!* and what that phrase implies.

1.2 "Just use Postgres!" explained

As Postgres kept gaining popularity as a general-purpose database, more and more experts could be heard saying "Just use Postgres!" in online discussions and at technical events. At that time, many people still perceived Postgres as a traditional relational database suited only for OLTP workloads, but Postgres experts kept narrowing the knowledge gap by advising the use of the database for workloads that went far beyond transactional. Over the years, "Just use Postgres" has effectively become the motto of the Postgres community, highlighting the breadth and depth of use cases the database can now handle.

Does this mean Postgres has become a Swiss Army knife and the only database every developer needs? Certainly not.

Treat "Just use Postgres" as a reasonable hint from those who have been building or using Postgres for decades. If you or your team already use Postgres (or are planning to do so), and another use case needs to be supported (such as geospatial, time series, or generative AI), then before adding another database to your technology stack, check whether Postgres can solve the use case for you.

If you discover that Postgres works fine for the new use case, you will avoid needing to learn and run two or more database systems in production. If you conclude otherwise, no problem; bring in another database system that solves the use case better. The choice is always yours, and the goal of this book is to gradually guide you through various Postgres capabilities, helping you make better decisions and see the full potential of Postgres!

Now that you have a good understanding of Postgres and the reasons behind its increasing growth, let's see it in action as developers. We'll start by deploying, connecting to, and working with a Postgres instance on our laptops in under 10 minutes.

1.3 Starting Postgres in Docker

According to the stats for the Postgres repo on GitHub, more than 85% of the database code is written in the good old C programming language. The Postgres development group chose to support both Unix-based and Windows operating systems, producing database binaries for a broad range of systems and CPU architectures. Whether your laptop is powered by Windows, Linux, macOS, or another operating system, you can go to https://www.postgresql.org/download/ and pick an installation option that works best for you.

In this book, you'll learn how to deploy Postgres in a cross-platform way using Docker. With Docker, you can pass a single command that will spin up a database container in under a minute. This installation option is ideal if you want an easy-to-use development environment without making Postgres a permanent part of your developer laptop. Docker lets you start, stop, and remove the database container, bypassing OS-specific installers, package managers, and other services.

NOTE If you already have a preferred Postgres deployment option and want to continue using it while reading the book, feel free to do so, but keep in mind that the output of some commands and connection strings may differ on your end.

Figure 1.2 shows a high-level representation of how Postgres will be deployed in Docker on your machine and the way you'll work with the database while reading the book:

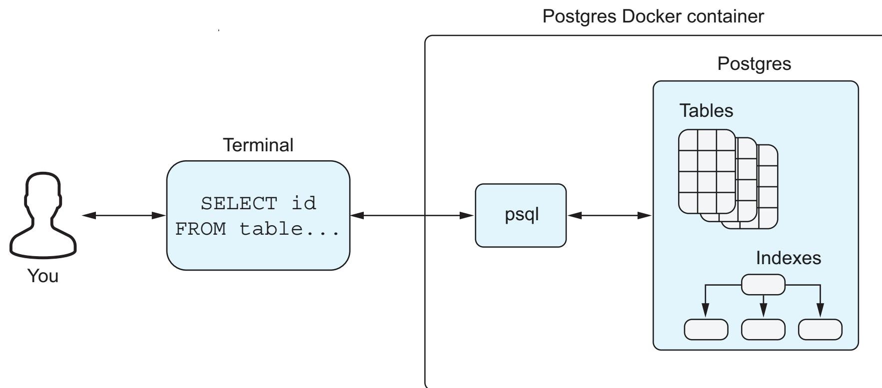


Figure 1.2 How Postgres is deployed and used throughout the book

- *Postgres Docker container*—Throughout the book, you'll run Postgres in Docker using different images. In the first seven chapters, you'll use the official Postgres image with all the capabilities found in the standard/core database distribution. From chapter 8 onward, you'll use images with additional extensions not included in the standard deployment. For example, you'll have a dedicated Postgres container with the `pgvector` extension to explore the database's generative AI capabilities, and another container with the TimescaleDB extension for time-series workloads.
- *Postgres*—This is the actual Postgres instance running inside the container. It stores application data in tables and optimizes searches using various database indexes that you'll explore throughout the book. Note that the instance consists of several processes, not shown in the figure for simplicity. Depending on their role, these processes can listen for incoming client connections, handle client requests, or perform other database-specific tasks and routines.
- *Psql*—This powerful and lightweight command-line tool is included in the standard Postgres distribution and, therefore, available within the Postgres Docker container. This means you don't need to install it or any other tool separately on your host operating system to work with Postgres. You'll use psql from within the container to connect to the database and run various SQL commands and statements against Postgres.

- *Terminal*—This application runs on your host OS, which lets you interact with the OS and other external services or systems via a command-line interface. Examples include PowerShell on Windows, iTerm on macOS, and Terminal on Linux. You'll use the terminal to spin up the Postgres container and then connect to the database using the psql tool. Once connected, you'll type and run SQL queries, which psql will execute against Postgres.

With this Postgres deployment model in mind, the only requirement is to have Docker installed on your machine. If the following command fails in your terminal, then you need to install Docker:

```
docker version
```

If the command fails and you're new to Docker, consider installing Docker Desktop, which is available for all major OSs and provides a straightforward way to manage containers: https://docs.docker.com/desktop/.

Once Docker is installed and ready, use the command in the following listing to start a Postgres database container on a Unix-based OS such as Linux or macOS.

Listing 1.1 Starting a Postgres container on Unix

```
docker volume create postgres-volume

docker run --name postgres \
    -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password \
    -p 5432:5432 \
    -v postgres-volume:/var/lib/postgresql/data \
    -d postgres:17.2
```

If you're a Windows user, run the following command in PowerShell. If you use Command Prompt (CMD), replace the backtick (`) with a caret (^) at the end of each line.

Listing 1.2 Starting a Postgres container on Windows

```
docker volume create postgres-volume

docker run --name postgres `
    -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password `
    -p 5432:5432 `
    -v postgres-volume:/var/lib/postgresql/data `
    -d postgres:17.2
```

These commands create a Docker-managed volume named `postgres-volume` and then start a new database container named `postgres` with the following settings:

- `POSTGRES_USER` and `POSTGRES_PASSWORD` define the default username and password, set to `postgres` and `password`, respectively.

- The `-p` parameter maps the container's Postgres port (5432) to the host system, allowing incoming connections.
- The `-v` parameter mounts `postgres-volume` into the container in the /var/lib/postgresql/data directory. This directory is where Postgres stores its database files. Because the volume is managed by Docker and stored on the host OS, your data persists even if you stop, remove, or re-create the container.
- The `-d` flag starts the container as a background process that doesn't occupy a terminal window. The container uses the Postgres 17.2 image from the official Docker Hub (`postgres:17.2`). This was the latest version of the database at the time the book was written, and you're encouraged to use it while reading the book to ensure that the output of all commands and code snippets matches on your end.

NOTE The way we deploy Postgres in Docker is suitable for development and for exploring the database's capabilities. However, if you plan to use this deployment option in production, be sure to review security and other best practices.

Usually, it can take up to a minute to pull the image from Docker Hub and perform the initial configuration of the database container. You can always check the status by running the following command:

```
docker container ls -f name=postgres
```

If the `STATUS` field is similar to this one, it means the container has started successfully:

```
IMAGE            STATUS              PORTS                       NAMES
postgres:17.2    Up About a minute   0.0.0.0:5432->5432/tcp      postgres
```

Finally, you can do one last check by viewing the database logs using the following Docker command:

```
docker logs postgres
```

If you see this message in the logs, then you've successfully initialized Postgres and it's ready to serve your requests:

```
listening on IPv4 address "0.0.0.0", port 5432
listening on IPv6 address "::", port 5432
...
database system is ready to accept connections
```

TIP If Docker fails to start Postgres on your machine for any reason, try searching for the specific error online. Docker is so widely used that you should be able to find a solution quickly.

1.4 Connecting with psql

With the Postgres container running, you can roll up your sleeves and start experimenting. You can always connect to the database using your preferred SQL tool or create a code snippet in your favorite programming language. But what if you don't have any SQL tools on your laptop and you're not ready to build a sample application using Postgres? In this case, the psql tool comes to the rescue.

The psql tool is a lightweight command-line tool that is included in the standard database distribution. The source code for psql is located in the main Postgres repository and is maintained by the PostgreSQL Global Development Group. This means the Postgres Docker container also includes psql, allowing you to connect to the database without needing to install any additional tools on your host OS. To connect to Postgres with psql from within the container, use the command in the next listing.

Listing 1.3 Connecting with psql

```
docker exec -it postgres psql -U postgres
```

The `docker exec -it` command connects to the `postgres` container in interactive mode. Once connected, it runs the `psql -U postgres` command, which uses the psql tool to connect to the Postgres instance running inside the container. The `-U postgres` flag tells psql to use `postgres` as the username.

Note that we don't specify the password in the psql connection string, and you won't be prompted for one when connecting. This is because, once inside the container (`docker exec -it postgres`), psql connects via a *local* Unix socket. The Postgres Docker image is configured to treat such local connections as *trusted*, not requiring the input of the password.

Upon successful connection, psql will welcome you with a prompt similar to the following:

```
psql (17.2 (Debian 17.2-1.pgdg120+1))
Type "help" for help.
postgres=#
```

TIP Use the `\q` command to terminate the psql connection to Postgres.

In addition to supporting standard SQL statements, psql also comes with various meta-commands that let you see information about the current connection, list existing database objects, copy data from external files, and much more. All these commands start with the backslash symbol (`\`).

TIP Run the `\?` command to view a complete list of meta-commands supported by psql. To close the generated list and return control to the psql prompt, press q on the keyboard.

As an example, the `\conninfo` command displays information about the current database connection:

```
\conninfo
You are connected to database "postgres" as user "postgres" via socket in
"/var/run/postgresql" at port "5432".
```

The message generated by the command indicates that you're connected to a database named `postgres`. In this context, the word *database* implies a *named collection* of database objects such as tables, views, and indexes, not the Postgres database container/server itself. The `postgres` database (a named collection) is created by default during initialization, and you connect to it unless another database is specified. You can create additional databases using the `CREATE DATABASE` statement, as discussed in greater detail in chapter 2.

The `\d` command is another frequently used meta-command that lists tables, views, and sequences belonging to the database you're connected to (which is `postgres`, in our case):

```
\d
Did not find any relations.
```

Because you've just started with a fresh Postgres database container, the command reports that you haven't created any tables or other relations yet. This is easy to address. You just need to load a dataset, which we do in the next section by generating mock data using built-in Postgres capabilities.

1.5 Generating mock data

Once you have the database running and know how to connect to it, the next logical step is to find a sample dataset, load it into the database, and run a few experiments to help you learn the database's capabilities. In the following chapters, you'll learn how to preload an existing dataset into Postgres. In the meantime, let's learn how to generate mock data using built-in Postgres capabilities.

> **NOTE** The data generation approach we're going to explore isn't considered the best practice in Postgres, but it's a handy technique you can use in certain situations without leaving the boundaries of your command-line tool.

First, let's use the `CREATE TABLE` command to create a simplified version of a table storing information about stocks traded on a stock exchange.

Listing 1.4 Creating a sample table

```
CREATE TABLE trades(
    id bigint,
    buyer_id integer,
    symbol text,
```

```
    order_quantity integer,
    bid_price numeric(5,2),
    order_time timestamp
);
```

Now, if you execute the `\d` meta-command, Postgres shows the table in the list of existing relations:

```
\d
        List of relations
 Schema | Name   | Type  |  Owner
--------+--------+-------+----------
 public | trades | table | postgres
```

However, if you execute the following request

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

you'll see that there are no trades in the table:

```
  Count
-------
     0
```

To fill the table with sample data, we'll use the Postgres `generate_series` function. As the name suggests, the function generates a series of values within a predefined range. It can produce a series of numeric values or timestamps. This version of the function creates a set of integers in the range from `start` to `stop` (inclusive). The `step` argument is optional and set to 1 by default:

```
generate_series (start integer, stop integer [, step integer ])
➥ setof integer
```

As an example, if you'd like to generate a series of integers from 1 to 5, make a call to the function as follows:

```
SELECT generate_series(1,5);
```

The function produces five rows and assigns the values to the column under a similar name: `generate_series`:

```
 generate_series
-----------------
               1
               2
               3
               4
               5
(5 rows)
```

The name of the column can be easily changed. Let's say you want the function to generate unique IDs for the trades. You can ask the function to assign the generated values to the `id` column:

```
SELECT generate_series(1,5) as id;
```

The function generates the same five integer values, but this time stores them in the `id` column:

```
 id
----
  1
  2
  3
  4
  5
(5 rows)
```

Even though `generate_series` produces integer values, it's more than enough for mock data generation of varying complexity. For instance, the `trades` table has a `buyer_id` column. Assuming that you want to generate five trades with a random `buyer_id`, you can make the following call to the function.

Listing 1.5 Generating random buyers

```
SELECT id, random(1,10) AS buyer_id
FROM generate_series(1,5) AS id;
```

The query returns the values for the `id` and `buyer_id` columns:

```
 id | buyer_id
----+----------
  1 |        5
  2 |        6
  3 |        1
  4 |        6
  5 |       10
(5 rows)
```

The `id` value is generated by the `generate_series` function, and `buyer_id` is set to a random value in the range from 1 to 10 (inclusive).

> **NOTE** The `random(min_value, max_value)` function used in listing 1.5 was added to Postgres in version 17. If you are using an earlier version of the database, use `floor(random()*(max_value - min_value + 1) + min_value)` instead to generate a random integer in the `[min_value, max_value]` range.

What if we want to generate values for the `stock` symbol column of the `text` type? In this case, the symbol is the registered name of a stock. Let's say the stock exchange

allows you to trade the stocks of Apple, Ford, and DoorDash. Those stocks have the following symbols: AAPL, F, and DASH. Using the SQL statement in the next listing, you can generate five random trades for these stocks.

Listing 1.6 Generating random stock trades

```
SELECT id,
(array['AAPL','F','DASH'])[random(1,3)] AS symbol
FROM generate_series(1,5) AS id;
```

The output might be as follows:

```
 id | symbol
----+--------
  1 | AAPL
  2 | DASH
  3 | F
  4 | DASH
  5 | F
(5 rows)
```

The `generate_series` function produces five rows (trades), using generated integer values as IDs for the trades. The value of the `symbol` column is randomly selected from the array of text values: `['AAPL', 'F', 'DASH']`.

If we follow the same principle, we can generate mock data for the remaining columns of the `trades` table. The following listing shows the final query that both generates and inserts 1,000 sample trades.

Listing 1.7 Inserting 1,000 sample trades

```
INSERT INTO trades (id, buyer_id, symbol,
    order_quantity, bid_price, order_time)
SELECT
    id,
    random(1,10) as buyer_id,
    (array['AAPL','F','DASH'])[random(1,3)] as symbol,
    random(1,20) as order_quantity,
    round(random(10.00,20.00), 2) as bid_price,
    now() as order_time
FROM generate_series(1,1000) AS id;
```

The `SELECT` statement uses the `generate_series(1, 1000)` function to populate 1,000 rows with the following mock data:

- The `id` column is set to the next numeric identifier produced by the `generate_series` function.
- The `buyer_id` is set to a random identifier in the range from 1 to 10.
- The `symbol` column stores a stock randomly selected from the array `['AAPL', 'F', 'DASH']`.

- The `order_quantity` column is initialized with a random value from 1 to 20.
- The `bid_price` column is set to a random number between 10 and 20.
- The `order_time` column is initialized with the current timestamp.

This is the power of SQL. Using a simple SQL query, you can generate mock data using Postgres' built-in capabilities.

As a final check, take a look at the first five generated rows:

```
SELECT id, buyer_id, symbol, order_quantity as quantity,
    bid_price, order_time
FROM trades
LIMIT 5;
```

This query retrieves values for all the columns listed in the `SELECT` statement and shows how to assign a custom column name (alias) by returning `order_quantity` as `quantity`:

```
 id | buyer_id | symbol | quantity | bid_price |         order_time
----+----------+--------+----------+-----------+---------------------------
  1 |        3 | F      |       18 |     12.92 | 2024-08-02 02:13:36.266052
  2 |        9 | AAPL   |       19 |     19.91 | 2024-08-02 02:13:36.266052
  3 |        7 | F      |        3 |     17.61 | 2024-08-02 02:13:36.266052
  4 |       10 | F      |        1 |     19.55 | 2024-08-02 02:13:36.266052
  5 |        9 | AAPL   |       15 |     12.25 | 2024-08-02 02:13:36.266052
(5 rows)
```

NOTE Because we used the `random` and `now` functions to generate the table data, the output in the book will differ from what you see when executing the `SELECT` queries in this chapter.

1.6 Running basic queries

SQL is the language that Postgres "speaks" natively. Throughout this book, we'll continue exploring the database's essential and extended capabilities by "talking" to Postgres in its native language.

Even if you decide to use an object-relational mapping (ORM) framework on top of Postgres, the ORM will still generate and execute raw SQL against the database. Therefore, it's always beneficial for you, as a developer, to know how to read and write SQL queries, as there may be times when you need to debug or optimize SQL requests generated by the ORM.

To conclude this chapter, let's run a few basic SQL queries that many developers frequently use in their application logic. One of the most common queries is to get the number of records that satisfy a specific condition. A combination of the `count` function with a filter condition passed into the `where` clause usually does the job. For example, you can get the total number of trades for Apple stock as follows:

```
SELECT count(*) FROM trades WHERE symbol = 'AAPL';
```

The query returns the number of trades generated by the random function:

```
count
-------
   305
(1 row)
```

Note the * symbol that is passed to the `count` function. It instructs the database to return the data for all the columns of all the selected rows. Generally, you should avoid using the * operator unless it's truly necessary to return all the columns to your application logic. The best practice is to specify the names of the columns that your application really needs. This habit will help you use resources like memory, CPU, and network bandwidth more prudently.

Does it mean that by running the `count(*)`, we're not following best practices? In fact, we're taking advantage of a special Postgres optimization by making that call. The `count(*)` function is treated specially in Postgres, as it simply counts the number of rows without retrieving the actual data stored in the columns.

What else can we do with our sample dataset? As developers, we're often tasked with creating logic to find the most popular items—those in high demand. For instance, this simple SQL query lets you find the most-traded stocks by volume.

Listing 1.8 Finding most-traded stocks by volume

```
SELECT symbol, count(*) AS total_volume
FROM trades
GROUP BY symbol
ORDER BY total_volume DESC;
```

The query uses the `GROUP BY` clause to group rows with the same stock `symbol` and then calculates the total number of trades in each group. The result is ordered by the `total_volume` column in descending order. The output might be as follows:

```
 symbol | total_volume
--------+--------------
 DASH   |          351
 F      |          344
 AAPL   |          305
(3 rows)
```

Another common task is to find the most valuable clients (or frequent buyers) who spend the most on our products and services. In the context of our stock exchange, let's find the top three buyers who have spent the most. This query is as simple and elegant as the previous one.

Listing 1.9 Finding the top three buyers

```
SELECT buyer_id, sum(bid_price * order_quantity) AS total_value
FROM trades
```

```
GROUP BY buyer_id
ORDER BY total_value DESC
LIMIT 3;
```

This time, the query groups rows by `buyer_id` and calculates the total dollar amount of proceeds for each buyer using the `sum(bid_price * order_quantity)` function. The result set is then ordered in descending order, and the `LIMIT 3` clause is used to return only the top three buyers. Postgres returns results that might be as follows:

```
 buyer_id | total_value
----------+-------------
        8 |    18260.73
        9 |    17295.22
        2 |    17133.51
(3 rows)
```

You can continue by calculating the average price per stock, finding the most recent trades, identifying the highest price trade for each stock, and much more. You can do a lot by crafting simple SQL queries and running them against the dataset you generated using built-in Postgres capabilities just a few minutes ago. By now, you've had a taste of how to get started with Postgres, and we're ready to begin exploring its capabilities in more depth. Let's go.

Summary

- Postgres is one of the most popular and fastest-growing databases.
- Postgres's open source nature, enterprise readiness, and extensibility are key factors contributing to its popularity and growth.
- The phrase "Just use Postgres" implies that Postgres offers a wide range of capabilities, allowing it to handle use cases far beyond traditional transactional workloads.
- Postgres is written in C and can be installed on Windows and a wide range of Unix-based operating systems.
- The database can be started as a container in under a minute on any operating system that supports Docker.
- Postgres comes with the `generate_series` function, which can be used to generate mock data of varying complexity.
- Postgres "speaks" SQL natively, allowing you to solve various business tasks by crafting simple and elegant SQL queries.

2 Standard RDBMS capabilities

This chapter covers

- Creating database objects
- Querying and manipulating data
- Using JOINs and transactions
- Executing application logic within a database
- Enforcing data integrity
- Restricting access to data

This book assumes you're familiar with the basics of relational database management systems (RDBMSs) and Structured Query Language (SQL). At a minimum, you should understand what relational databases are used for and how to run basic SQL queries. If you meet this bar, this chapter offers an overview of standard capabilities typically supported across all SQL databases, including Postgres. If these concepts still sound new to you, no problem—enjoy reading this chapter to see the standard features in action, and refer to other resources whenever you'd like a deeper understanding of any topic.

Let's take a quick tour of the widely used RDBMS capabilities that Postgres provides while designing a multitenant eCommerce platform used by thousands of

merchants worldwide. First, we'll learn how to create a database structure that stores and isolates each merchant's (tenant's) data within a single Postgres instance. Then we'll explore how to query, modify, analyze, and secure access to that data using the abundant features the database offers.

2.1 Creating the database structure

Postgres stores application data in tables, which are named collections of rows. Each row has the same set of columns with similar data types. At a minimum, we need to think through the tables our application needs and create them using the `CREATE TABLE` command. However, in addition to tables, many applications can benefit from higher-level abstractions—such as schemas and databases—that help better define the overall database structure.

The *schema* is a named collection of database objects. You can think of schemas as similar to namespaces or packages used in many programming languages. We can use schemas whenever we want to group tables, views, or other objects under one logical name. By default, all the database objects you create are placed in the `public` schema, which is created by Postgres during the bootstrap process.

NOTE In Postgres, a *database object* is a named SQL object that is used to store or manipulate data. Tables, views, functions, and triggers are all examples of database objects.

A *database* is the topmost named collection of objects that includes schemas, which in turn contain tables and other objects. Databases make it easy to isolate different datasets and application workloads. Tables belonging to different schemas but stored within a single database can still be queried together; you just connect to the database and run queries that pull data from several tables. However, if these tables belong to different databases, it won't be possible to run a SQL query spanning multiple databases unless you use a third-party solution. When Postgres is initialized for the first time, it creates a default database named `postgres`, and that's the one users typically connect to and use unless a different database is specified.

NOTE Throughout the book, we often refer to Postgres as a "database," which can make the term a bit ambiguous. Sometimes we talk about Postgres as an instance of a database server running on your laptop or in the cloud. Other times, we mean a logical database within that server instance—such as the default `postgres` database created during initialization or other database objects created by your application.

If you've never used schemas or databases in practice, this definition might not fully clarify their practical purpose. To make things clearer, let's review a use case of an eCommerce platform where merchants can launch their online stores within days and start selling goods online. As a software-as-a-service (SaaS) platform, our eCommerce platform would benefit from using all three foundational objects—databases, schemas, and tables—which will allow us to see their practical implications in action.

2.1.1 Creating databases

Meet Jake, the owner of a small coffee chain on the East Coast of the United States. Customers have fallen in love with his brand, which is known for its high-quality coffee, exceptional service, and experience. The coffee shops have even become popular spots for tourists seeking to explore local gems. At some point, Jake decides to start selling coffee beans and other branded products online. The problem is, he has no idea how to start.

Fortunately, he has a friend who lives across the country on the West Coast. Susannah owns a super-popular local brewery, and her loyal customers encouraged her to start selling beer online. Like Jake, Susannah had no experience in high-tech but managed to launch an online store within a week. So Jake decides to talk to her and learn her secret.

It turns out that Susannah uses our eCommerce platform, which allowed her to launch a fully functional online store in just a few days. She simply registered the brewery with the service, created a product catalog, and launched the store! Features like payments, order delivery with tracking, and customer account management were provided out of the box. Inspired by her success, Jake follows his friend's path and decides to launch an online shop for the coffee chain on the platform as well.

Now, let's get back to Postgres and software engineering. Our eCommerce platform is an example of a *multitenant* SaaS application. With a multitenant architecture, it's possible to build scalable and cost-effective services by sharing resources like CPUs and storage across multiple customers (*tenants*). This architecture ensures data isolation and fine-grained security controls to keep each tenant's data private and accessible only under specific conditions.

In our platform, a merchant is considered a tenant. Even though the brewery and coffee chain owners know each other and both use our platform for their online stores, they don't need to share their business data. Therefore, at the database level, the eCommerce platform fully isolates each tenant from the others.

There are several ways to implement multitenancy in Postgres, with the following being the most popular options:

- *Table-level multitenancy*—Data for all tenants is stored in shared tables, with a column such as `tenant_id` used to distinguish one tenant's data from another's.
- *Schema-level multitenancy*—Each tenant has a dedicated schema with its own tables, while all tenant schemas reside within a single logical database.
- *Database-level multitenancy*—Each tenant has its own database.

We use database-level multitenancy to ensure that each merchant has a dedicated database object storing all data related to their business. Access to each database will be controlled through roles and permissions (created at the end of chapter 2), ensuring that merchants can connect only to their own database and work with their business's data. Additionally, with this multitenancy approach, we can easily distribute tenants across multiple Postgres servers by assigning their databases to specific Postgres

instances. This becomes especially beneficial if our SaaS platform grows in popularity and requires additional compute and storage capacity.

As shown in figure 2.1, a single Postgres instance hosts data for two different merchants: the brewery from the West Coast and the coffee chain from the East Coast. This approach allows Jake and Susannah to pay lower fees because Postgres compute and storage resources are shared among them. At the same time, it's an excellent practical example for using various database objects in Postgres.

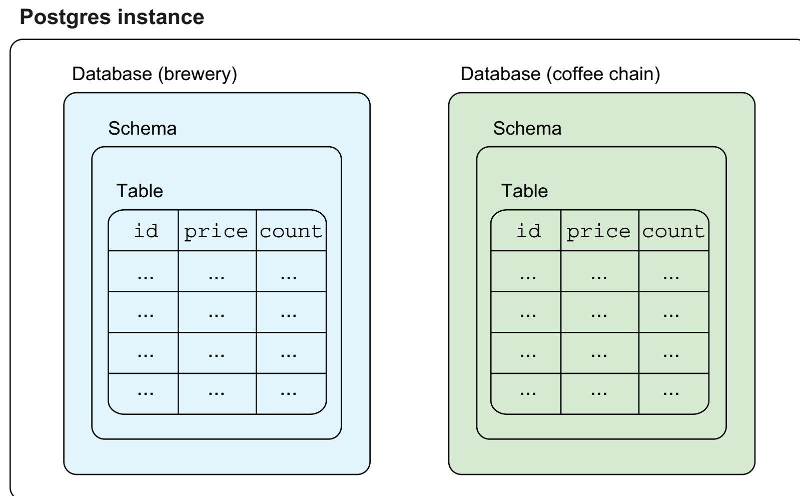


Figure 2.1 A Postgres instance with two databases for different tenants

Let's now connect to the Postgres instance in Docker and see how to create isolated databases for each tenant:

1. Connect to the Postgres instance started in chapter 1 using psql:

   ```
   docker exec -it postgres psql -U postgres
   ```

   This command connects you to the default `postgres` database that is created during the Postgres bootstrap process.
2. Create database objects for the brewery and the coffee chain:

   ```
   CREATE DATABASE coffee_chain;
   CREATE DATABASE brewery;
   ```

3. Run this SQL query against the `pg_database` system catalog to see a full list of existing databases:

   ```
   SELECT datname FROM pg_database;
   ```

The output is as follows:

```
 datname
--------------
 postgres
 coffee_chain
 template1
 template0
 brewery
(5 rows)
```

The databases for Susannah and Jake are ready. From now on, as we explore other standard RDBMS capabilities of Postgres, we'll use one tenant as an example: Jake, the owner of the coffee chain. Susannah and the other merchants on our eCommerce platform will have their data organized and accessed in the same way, with the only difference being that their data will reside in their own databases.

Let's move on by connecting to the `coffee_chain` database using the `\c` meta-command of psql:

```
\c coffee_chain
```

On successful connection, psql greets you with the following message:

```
You are now connected to database "coffee_chain" as user "postgres".
coffee_chain=#
```

TIP When you connect to a specific database, include the database name in the connection string. For psql, you can also connect directly to `coffee_chain` by passing the database name via the `-d` parameter, like this: `docker exec -it postgres psql -U postgres -d coffee_chain`.

Next, let's explore how to take advantage of database schemas while creating them for the coffee chain.

2.1.2 Creating schemas

A *schema* is a named collection of various database objects. It's the next-level abstraction after databases for grouping tables, views, stored procedures, and other objects at a more granular level.

After connecting to the database of Jake, the coffee chain owner, you can execute the psql `\dn` meta-command to see a list of existing schemas:

```
\dn
      List of schemas
  Name  |       Owner
--------+-------------------
 public | pg_database_owner
(1 row)
```

The command returns only one schema named `public`, which Postgres creates for every new database. Unless you create and use a different schema, various database objects, including tables, functions, and views, will be stored under the `public` schema by default. This applies to `SELECT`, `INSERT`, and other data manipulation commands as well—if you don't specify a schema for the tables in your query, Postgres defaults to the current schema, which is `public` unless you create and use another one.

Having just the `public` schema might be sufficient for your use case. There is no need to create additional schemas unless they improve the design and manageability of an application architecture. However, our eCommerce platform would benefit from having custom schemas.

Every company that launches an online store on the platform will need to manage a product catalog, handle customer accounts, and fulfill orders. It's likely that these capabilities will be organized as standalone modules or microservices on the application layer. Similarly, each functional module on the application layer can be mapped to a dedicated schema on the database layer.

As shown in figure 2.2, the SaaS platform creates three additional custom schemas for each tenant. The `products` schema stores all tables related to the product catalog and pricing. The `customers` schema contains data for customer accounts and customer product reviews. Finally, the `sales` schema stores tables related to order placement, management, and fulfillment.

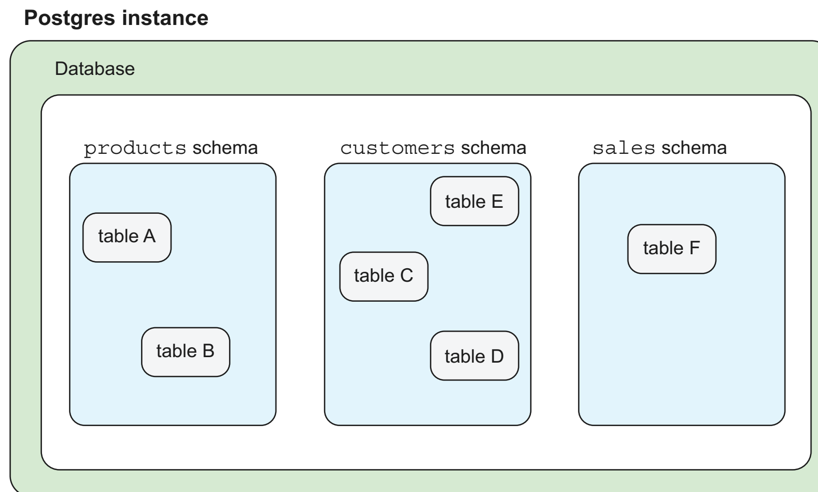


Figure 2.2 Create a custom schema mapped to application modules.

In this use case, schemas help us better align the database structure with the application layer's architecture. Each application module or microservice has its own schema

containing all the related data. This setup also avoids name conflicts if microservices have tables with similar names and allows running queries that access tables from different schemas. For example, as you'll see later, you can run a single `SELECT` statement accessing table data from multiple schemas. This wouldn't be possible if each module or microservice had its data stored in separate database objects.

Now that we understand the role of schemas in the eCommerce platform, let's add custom schemas to the coffee chain's database:

```
CREATE SCHEMA products;
CREATE SCHEMA customers;
CREATE SCHEMA sales;
```

Then execute the `\dn` command to see an updated list of existing schemas:

```
\dn

        List of schemas
   Name    |       Owner
-----------+-------------------
 customers | postgres
 products  | postgres
 public    | pg_database_owner
 sales     | postgres
(4 rows)
```

TIP If the same set of databases, schemas, and tables needs to be created for every tenant, consider automating this process. For example, in our eCommerce platform, a new database with all required schemas and other database objects can be automatically created when a new business owner signs up and starts configuring various modules.

Once you have more than one schema in the database, it's important to understand the concept of the current schema. The *current schema* is the one that Postgres uses implicitly if no specific schema is mentioned in a command that requires it. To check which schema is currently set, you can run the `SHOW search_path` command:

```
SHOW search_path;

   search_path
-----------------
 "$user", public
(1 row)
```

The current schema can always be changed with the `SET search_path TO schema` command. For example, the following command sets the current schema to `products`:

```
SET search_path TO products;
```

After that, all queries and commands that don't explicitly specify a schema will be executed against the `products` schema. Now, with the custom schemas in place, we can add the first tables to the coffee chain's database.

2.1.3 Creating tables

Whether you're working on a pet project or building an enterprise application, you create and use tables regardless of the complexity of a use case. A *table* is the actual database object that stores the application data. If you decide to put some data into Postgres, it needs to be added to one of the existing tables. As discussed earlier, Postgres creates the `postgres` database and the `public` schema by default, making the database and schema creation steps optional. However, the design and creation of tables are fully on you—there is no shortcut.

Imagine that Jake, the owner of the coffee chain, is getting ready for the launch of his online store, and he needs to populate the product catalog. Create the `catalog` table where the eCommerce platform will store Jake's products.

Listing 2.1 Creating a product catalog table

```
CREATE TABLE products.catalog (
    id SERIAL PRIMARY KEY,
    name VARCHAR (100) NOT NULL,
    description TEXT NOT NULL,
    category TEXT CHECK (category IN ('coffee', 'mug', 't-shirt')),
    price NUMERIC(10, 2),
    stock_quantity INT CHECK (stock_quantity >= 0)
);
```

This statement places the `catalog` table under the `products` schema. The schema name is optional in the `CREATE TABLE` statement and is used only if a table needs to be added to a specific schema.

The `id` column is of the `SERIAL` type, meaning that Postgres will use an underlying sequence object to generate IDs automatically for all the products. The sequence generates 32-bit integer numbers with a maximum value of 2,147,483,647, which equals $2^{31} - 1$.

The table uses several `NOT NULL` and `CHECK` constraints to enforce data integrity at the database level. These constraints are discussed in greater detail in section 2.3.

In addition to the `catalog` table, create the `reviews` table where the SaaS platform will keep customer reviews.

Listing 2.2 Creating a product reviews table

```
CREATE TABLE products.reviews (
    id BIGSERIAL PRIMARY KEY,
    product_id INT,
    customer_id INT,
    review TEXT,
    rank SMALLINT
);
```

This statement creates the `products.reviews` table, where `products` is the schema name and `reviews` is the table name. The IDs for the customer reviews are also generated automatically by Postgres. However, this time, we use the `BIGSERIAL` datatype: it generates 64-bit integer numbers with a maximum value of 9,223,372,036,854,775,807, which equals $2^{63} - 1$. The assumption is that the number of product reviews will significantly surpass the number of products, so a larger data type might be necessary.

NOTE Postgres has a rich set of data types you can choose from. The Postgres documentation is an excellent resource on this and many other topics discussed in the book. Refer to the Postgres docs when you need to pick data types for columns that best match the data types used on the application layer: https://www.postgresql.org/docs/current/datatype.html.

The definition of the `products.reviews` table is simpler than that of `products.catalog` because we haven't added any constraints. This is done intentionally for two reasons. First, although creating constraints is highly recommended, it remains optional. It's up to you to decide whether to add them at the database level or rely solely on application-level checks to ensure data integrity. Second, if you're unsure about adding a specific constraint, you can always do it later using the `ALTER TABLE` command, as discussed in section 2.3. This flexibility shows that both SQL and Postgres can easily adapt to your needs.

Finally, let's use the psql's `\d` meta-command to see a list of created database objects:

```
\d

                List of relations
  Schema  |       Name       |   Type   |  Owner
----------+-----------------+----------+----------
 products | catalog         | table    | postgres
 products | catalog_id_seq  | sequence | postgres
 products | reviews         | table    | postgres
 products | reviews_id_seq  | sequence | postgres
(4 rows)
```

As you can see, the product's `catalog` and `reviews` tables are ready, along with two sequences that Postgres will use to generate unique IDs for products and reviews. If the list is empty on your end, either set the current schema to `products` using the `SET search_path TO products;` statement and run `\d` one more time, or use the `\dts products.*` command.

TIP Sequences are widely used for generating unique numeric identifiers in Postgres, but they aren't the only option. Another common technique is to use UUID as the data type for ID columns and let Postgres or the application layer generate the UUID-based identifiers. The UUID generation approach is reviewed in section 2.4.

With the product-specific tables ready, let's go over the basic SQL queries for data querying and manipulation as we populate the catalog for the coffee chain's online store.

2.2 Querying and manipulating data

The SQL standard defines four key commands for data querying and manipulation in relational databases: `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. These commands fall under the data manipulation language (DML) category of SQL. Let's put these commands into practice as we work with the coffee chain's product catalog.

Make sure you're still connected to the coffee chain's database to follow the steps in this section. As a reminder, this is how you can connect to the database with psql:

```
docker exec -it postgres psql -U postgres -d coffee_chain
```

Suppose Jake has fully set up his online store using our eCommerce platform and now needs to add the first products to his catalog. He goes to the store's user interface (UI) and adds the first five items to the list. Our platform's backend receives the request from the frontend (Jake's UI) and executes the following `INSERT` statement against the coffee chain's database in Postgres.

Listing 2.3 Inserting products

```
INSERT INTO products.catalog
  (name, description, category, price, stock_quantity) VALUES
  ('Sunrise Blend',
   'A smooth and balanced blend with notes of caramel and citrus.',
   'coffee', 14.99, 50),
  ('Midnight Roast',
   'A dark roast with rich flavors of chocolate and toasted nuts.',
   'coffee', 16.99, 40),
  ('Morning Glory',
   'A light roast with bright acidity and floral notes.',
   'coffee', 13.99, 30),
  ('Sunrise Brew Co. Mug',
   'A ceramic mug with the Sunrise Brew Co. logo.',
   'mug', 9.99, 100),
  ('Sunrise Brew Co. T-Shirt',
   'A soft cotton t-shirt with the Sunrise Brew Co. logo.',
   't-shirt', 19.99, 25);
```

The query adds five products: three types of bags of coffee in different flavors, one branded coffee mug, and one branded t-shirt. The application provided values for all the product catalog's columns except `catalog.id`, which was automatically generated and assigned by Postgres. You can run this query to view the generated identifiers:

```
SELECT id, name FROM products.catalog;

 id |           name
----+--------------------------
  1 | Sunrise Blend
  2 | Midnight Roast
  3 | Morning Glory
```

```
  4 | Sunrise Brew Co. Mug
  5 | Sunrise Brew Co. T-Shirt
(5 rows)
```

Next, Jake navigates to the products page of the website—the same page that customers will see and interact with. He wants to check that filtering by product category works correctly, so he selects the Coffee category from the drop-down list. The platform's backend queries Postgres and executes the following `SELECT` statement to retrieve the latest data matching the search criteria:

```
SELECT id, name, price FROM products.catalog
WHERE category = 'coffee';
```

The query uses the `WHERE` clause to filter data at the database level, returning only the products in the `coffee` category. The web page on Jake's end reloads and displays three bags of coffee that are on sale:

```
 id |      name      | price
----+----------------+-------
  1 | Sunrise Blend  | 14.99
  2 | Midnight Roast | 16.99
  3 | Morning Glory  | 13.99
(3 rows)
```

TIP If you'd like to query the `catalog` table without specifying its schema name, first use the `SET search_path TO products;` to set the current schema to `products`, and then query the table like this: `SELECT id FROM catalog`.

Next, Jake notices that the price for the Sunrise Blend doesn't look right, so he decides to update it. He clicks the Edit button next to the product's name and changes the price from $14.99 to $16.54. The platform's backend again queries the database and runs the following `UPDATE` statement, changing the price of the existing product:

```
UPDATE products.catalog SET price = 16.54 WHERE id = 1;
```

The `UPDATE` statement locates the record with `id = 1`, which corresponds to the Sunrise Blend, and changes the price to $16.54.

Finally, Jake remembers that due to a shortage of specific coffee beans, the Midnight Roast flavor is currently out of stock. He decides to remove the product from the list by clicking the Delete button next to the product name on the website UI. The eCommerce platform issues the following `DELETE` statement against the database:

```
DELETE FROM products.catalog WHERE id = 2;
```

The `DELETE` command locates the record with `id = 2`, which is the Midnight Roast, and removes it from the database. The database confirms the successful completion of the

operation, and the online store's product page reloads by executing the following query:

```
SELECT id, name, price, stock_quantity FROM products.catalog;
```

The query returns the four remaining products in the catalog:

```
 id |          name           | price | stock_quantity
----+-------------------------+-------+----------------
  3 | Morning Glory           | 13.99 |             30
  4 | Sunrise Brew Co. Mug    |  9.99 |            100
  5 | Sunrise Brew Co. T-Shirt | 19.99 |             25
  1 | Sunrise Blend           | 16.54 |             50
(4 rows)
```

Jake goes ahead and launches the online store in anticipation of the first sales. And we've just seen how easy it is to query and manipulate data in Postgres using `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements.

2.3 Data integrity

Relational databases are designed for transactional applications that require data consistency and integrity at all times. As a relational database, Postgres supports transactions to prevent data inconsistencies and provides various constraints to enforce data integrity. However, even though these capabilities are built into the database, it's up to us, developers, to use them effectively and in a timely manner.

NOTE *Data integrity* refers to data accuracy, consistency, and completeness over its entire lifecycle.

2.3.1 Constraints

The data types you choose for columns are the first line of defense in Postgres when it comes to data integrity. For instance, if a product price column is of the `NUMERIC` type, the database won't let you set the price to a `text` or `boolean` value. Another line of defense against inconsistencies is constraints, which are rules that are checked against the data an application tries to write to a column or set of columns. Postgres supports the following constraints:

- *Not-null constraint*—Ensures that a column cannot be set to a null value.
- *Unique constraint*—Guarantees that a column or group of columns doesn't hold duplicate values.
- *Primary key*—Defines a column or group of columns that serve as unique identifiers for rows in the table. It works as a combination of the not-null and unique constraints.
- *Foreign key*—Specifies that the values in a column or group of columns must match values in a row from another table.

- *Check constraint*—Verifies that the value of a column satisfies a custom condition.
- *Exclusion constraint*—Ensures that for any two rows being compared on specified columns or expressions with the given operators, at least one of these comparisons must return false or null.

Now, let's see how some of these constraints can be used in practice. Imagine that we, the creators of the eCommerce SaaS platform, are reviewing the existing constraints set on the tables. Initially, we assumed that the application backend would perform all the required verification before writing data to the database. But because we build quickly, we've already accidentally broken a few existing checks at the application layer, which has led to data integrity problems in the database. To prevent this from happening in the future, we decide to add additional constraints at the database level.

NOTE If you know in advance which table constraints are necessary, define them right away in the `CREATE TABLE` statement. We didn't do this earlier for educational purposes, so you can see how to add them later in Postgres.

First, make sure you're still connected to the coffee chain's database with psql:

```
docker exec -it postgres psql -U postgres -d coffee_chain
```

Let's look at the existing constraints of the `products.catalog` table by executing the `\d products.catalog` command:

```
\d products.catalog

               Table "products.catalog"
     Column     |          Type          | Nullable |
----------------+------------------------+----------+
 id             | integer                | not null |
 name           | character varying(100) | not null |
 description    | text                   | not null |
 category       | text                   |          |
 price          | numeric(10,2)          |          |
 stock_quantity | integer                |          |
Indexes:
    "catalog_pkey" PRIMARY KEY, btree (id)
Check constraints:
    "catalog_category_check" CHECK (category = ANY (ARRAY['coffee'::text,
        'mug'::text, 't-shirt'::text]))
    "catalog_stock_quantity_check" CHECK (stock_quantity >= 0)
```

Let's take a closer look at some parts of the output:

- The `id` column is the table's primary key, which ensures that every row has a unique identifier and that it can't be set to `null`. The primary key relies on the `catalog_pkey` index, which allows Postgres to verify the uniqueness of the `id` column by quickly traversing the index structure. The search through the index has

logarithmic complexity, which is much faster than performing a linear scan (full scan) of all the table's records.

- In addition to the `id` column, the `NOT NULL` constraint is also set for the `name` and `description`, requiring the application layer to pass values for these fields.
- The value of the `category` column is validated by the `catalog_category_check`, a check constraint that requires the application to pick from `coffee`, `mug`, or `t-shirt` as a value.
- `catalog_stock_quantity_check` (another check constraint) ensures that the `stock_quantity` cannot be less than 0.

Overall, from a data integrity standpoint, the `products.catalog` table is already in good shape. However, there's still one obvious improvement we can make. Take a look at the `price` column, which currently has no constraints. What if a merchant accidentally enters a negative price? We can rely on the application layer to perform price verification, but if the application code changes frequently, the verification logic might be broken at some point. Therefore, adding an additional check at the database level would do no harm.

Postgres supports the `ALTER TABLE` command, which allows you to modify an existing table even if it already stores data. Let's use the following SQL statement to add a constraint that ensures the price is higher than 0.

Listing 2.4 Adding a price constraint

```
ALTER TABLE products.catalog
    ADD CONSTRAINT catalog_price_check CHECK (price > 0);
```

Now, suppose Jake accidentally sets the price of one of his bags of coffee to 0 or a negative value, and the application backend misses this mistake:

```
UPDATE products.catalog SET price = -16 WHERE name = 'Morning Glory';
```

Postgres detects this oversight and raises the following error:

```
ERROR:  new row for relation "catalog" violates
check constraint "catalog_price_check"
DETAIL:  Failing row contains (3, Morning Glory, A light roast with
  bright acidity and floral notes., coffee, -16.00, 30).
```

WARNING The `ALTER TABLE` command that adds a new constraint can fail if at least one row contains a value that violates the new constraint. For example, if any row in the `products.catalog` table had a negative price value, the command from listing 2.4 would fail with an exception like `ERROR: check constraint "catalog_price_check" of relation "catalog" is violated by some row`. To mitigate this problem, you would first need to update the rows that violate the constraint and only then enforce the new check.

After reviewing the set of constraints on the `products.catalog` table of our platform, we proceed to the `products.reviews` table, where the database stores customer feedback about products. We run the `\d products.reviews` command to examine the existing constraints, and the truncated output looks as follows:

```
\d products.reviews

        Table "products.reviews"
   Column     |   Type    | Nullable |
-------------+----------+----------+
 id          | bigint   | not null |
 product_id  | integer  |          |
 customer_id | integer  |          |
 review      | text     |          |
 rank        | smallint |          |
Indexes:
    "review_pkey" PRIMARY KEY, btree (id)
```

We notice that, aside from the primary key on the `id` column, no other constraints are associated with the table. We immediately decide to add additional checks to ensure that the `review` column is never set to null and that `rank` is within the range of 1 to 5.

Listing 2.5 Adding constraints for the `reviews` table

```
ALTER TABLE products.reviews
    ALTER COLUMN review SET NOT NULL,
    ADD CONSTRAINT review_rank_check CHECK (rank BETWEEN 1 AND 5);
```

2.3.2 Foreign keys

The `product_id` and `customer_id` columns are supposed to store the IDs of existing products and customers. But what if `product_id` is set to an identifier of a nonexistent product, or `customer_id` stores the ID of a customer who has deleted their account? To prevent these data-integrity-related oversights, we can use the foreign key constraint.

Using the following `ALTER TABLE` command, we add a foreign key to the `product_id` column of the `products.reviews` table. This key references the `id` column of the `products.catalog` table.

Listing 2.6 Creating a foreign key on `product_id`

```
ALTER TABLE products.reviews
    ADD CONSTRAINT products_review_product_id_fk
    FOREIGN KEY (product_id) REFERENCES products.catalog(id);
```

A column that is referenced by a foreign key must have an associated unique constraint. In our case, the `id` column of the `products.catalog` table is a primary key, which is

unique by definition. However, if this were not the case, listing 2.6 would fail with an error similar to this: `ERROR: there is no unique constraint matching given keys for referenced table "catalog"`. Foreign keys have this requirement for a good reason. Postgres needs to verify that a `product_id` value inserted into the `products.reviews` table actually exists in the referenced `products.catalog` table. The fastest way to perform this verification is by traversing an index, which is always created for primary keys or unique columns.

With this new constraint in place, it is no longer possible to leave reviews for nonexistent products. However, before validating this, let's create one more foreign key on the `customer_id` column of the `products.reviews` table.

The next SQL statement creates an `accounts` table under the `customers` schema.

Listing 2.7 Creating a customer account table

```
CREATE TABLE customers.accounts (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT NOT NULL,
    passwd_hash TEXT NOT NULL
);
```

Next, we create a foreign key on the `customer_id` column of the `products.reviews` table that references the `id` column (primary key) of the `customers.accounts` table.

Listing 2.8 Creating a foreign key on `customer_id`

```
ALTER TABLE products.reviews
    ADD CONSTRAINT products_review_customer_id_fk
    FOREIGN KEY (customer_id) REFERENCES customers.accounts(id);
```

Now we're ready to see how the foreign keys help improve the integrity of the data that merchants store on our eCommerce platform. Imagine that three loyal customers of the coffee chain discover that Jake has launched an online store where they can buy their favorites and have them delivered to their doors. The customers visit the website and sign up. We can emulate this by inserting three new records into the database.

Listing 2.9 Adding new customers

```
INSERT INTO customers.accounts (name, email, passwd_hash)
VALUES
('Alice Johnson', 'alice.johnson@example.com',
    '5f4dcc3b5aa765d61d8327deb882cf99'),
('Bob Smith', 'bob.smith@example.com', 'd8578edf8458ce06fbc5bb76a58c5ca4'),
('Charlie Brown', 'charlie.brown@example.com',
     '5f4dcc3b5aa765d61d8327deb882cf99');
```

Because the `id` column of the `accounts` table is of the `SERIAL` type, Postgres uses a sequence to generate unique identifiers for the customers:

```
SELECT id, name FROM customers.accounts;

 id |     name
----+---------------
  1 | Alice Johnson
  2 | Bob Smith
  3 | Charlie Brown
(3 rows)
```

Suppose Alice browses through the product catalog and comes across a branded coffee mug she already owns. She loves that mug so much that she can't resist leaving a positive review. The `id` of the mug is 4:

```
SELECT id, name FROM products.catalog WHERE name = 'Sunrise Brew Co. Mug';

 id |         name
----+----------------------
  4 | Sunrise Brew Co. Mug
(1 row)
```

She types the review and clicks the Send button, but an exception pops up, stating that the review can't be posted at this time. On investigation, we notice that many buyers across different merchants have failed to leave reviews within the last five minutes. We quickly connect to the application's backend and check the logs. We see that customers can't share reviews due to constraint violation errors reported by Postgres. Among those errors, we find an error message generated for Alice's review:

```
ERROR:  insert or update on table "reviews" violates
foreign key constraint "products_review_product_id_fk"
DETAIL:  Key (product_id)=(1004) is not present in table "catalog".
```

The error is generated when the application executes the following query against the database:

```
INSERT INTO products.reviews (product_id, customer_id, review, rank)
VALUES (1004, 1, 'This mug is perfect — sturdy, stylish, and keeps my coffee
warm for a good while.', 5);
```

The error suggests that something is wrong with the product IDs being passed to the database. The `product_id` for the coffee mug should have been 4, not 1004. It turns out we introduced a bug in our application logic that occasionally passes incorrect identifiers. This problem is easy to address, and we roll out the fix within minutes. After that, Alice can post her review, and the SQL statement with `product_id` set to 4 is successfully executed in Postgres.

Listing 2.10 Posting a product review

```
INSERT INTO products.reviews (product_id, customer_id, review, rank)
VALUES (4, 1, 'This mug is perfect — sturdy, stylish, and keeps
   my coffee warm for a good while.', 5);
```

Even though the bug at the application layer caused some disruption across several tenants, the effect could have been much greater if we didn't have foreign key constraints on the `reviews` table. Without the foreign keys, we would have spent much more time cleaning up reviews posted for incorrect or nonexistent products.

Another benefit of the foreign key constraint is that it works in both directions. As we've seen, it controls what's written to the table that owns the key (preventing reviews for nonexistent products), but it also keeps an eye on the referenced table. For instance, if Alice decides to remove her account from the coffee chain's store and the application logic requests Postgres to delete her record from the database, such an operation won't be permitted:

```
DELETE FROM customers.accounts WHERE id = 1;
```

The database produces the following error message:

```
ERROR:  update or delete on table "accounts" violates foreign key
   constraint "products_review_customer_id_fk" on table "reviews"
DETAIL:  Key (id)=(1) is still referenced from table "reviews".
```

Postgres warns us that there is still a review posted by Alice that would remain dangling if her customer record were deleted. We definitely don't want this to happen. So, we update our application logic to use soft deletion by adding the `deleted` column to the `customers.accounts` table and setting it to `false` by default.

Listing 2.11 Adding a `deleted` column to the `accounts` table

```
ALTER TABLE customers.accounts
    ADD COLUMN deleted boolean DEFAULT false;
```

Now, if any customer decides to remove their account, instead of physically deleting their record from the database, we can flip the `deleted` flag to `true`, allowing us to keep the reviews left for the products:

```
UPDATE customers.accounts SET deleted = true WHERE id = 1;
```

TIP If you prefer to delete customer accounts along with their product reviews, add the `ON DELETE CASCADE` clause to the foreign key definition. After that, whenever you delete a customer from the `customers.accounts` table, Postgres will also delete all the associated reviews from the `products.reviews` table.

With that, our eCommerce platform is in a good state from a data integrity standpoint, and we've learned how to take advantage of various Postgres constraints in practice.

2.4 Transactions

Whether you update a single record or multiple rows across several tables, you can trust that Postgres will handle the update transactionally in accordance with the ACID principles (atomicity, consistency, isolation, and durability). In short, this means changes will be either fully committed, ensuring that the database remains in a consistent state, or rolled back entirely if something goes wrong during the update.

2.4.1 Implicit transactions

Let's explore the transactional capabilities of Postgres in practice by continuing to use Jake's coffee chain as an example. First, make sure you're still connected to the coffee chain's database with psql:

```
docker exec -it postgres psql -U postgres -d coffee_chain
```

Suppose that Jake received a package of bags of coffee at one of his warehouses and now needs to update the product quantities on his online store. He goes to the store's admin panel and starts updating the catalog. He counts 100 bags of the Sunrise Blend flavor (`id = 1`), and increases the total quantity by that number, and the application backend executes the following query.

Listing 2.12 Updating the quantity for a single product

```
UPDATE products.catalog
SET stock_quantity = stock_quantity + 100
WHERE id = 1;
```

He continues counting the delivered products and discovers that there are 50 more bags of Sunrise Blend (`id = 1`) and the same number of bags of Morning Glory (`id = 3`). He goes back to the admin panel to update the quantity for both flavors, and our eCommerce platform sends an `UPDATE` statement updating two rows at once.

Listing 2.13 Updating the quantity for two products

```
UPDATE products.catalog
SET stock_quantity = stock_quantity + 50
WHERE id = 1 or id = 3;
```

The queries from listings 2.12 and 2.13 are examples of *implicit transactions* in Postgres. First, the database assigns internal transaction IDs to both updates. The queries then read the current quantities of the products from the `stock_quantity` column and increment them by the specified numbers. Finally, the updated quantities are written

back to the database, and Postgres commits the transactions. If anything goes wrong during the updates, Postgres rolls back the changes, returning the database to the state it was in before the implicit transactions began.

The same rule applies to `DELETE` and `INSERT` operations, which Postgres also executes within implicit transactions. So, even if you're not using transactions directly, the database is taking care of data consistency behind the scenes for you.

2.4.2 Explicit transactions

Explicit transactions are useful when you need to execute multiple SQL statements atomically, treating them as a single operation that either completes successfully or rolls back gracefully, leaving no trace. Explicit transactions start with the `BEGIN` statement and must be explicitly committed with the `COMMIT` operation. Let's explore this type of transaction while handling a customer order for the coffee chain. First, let's create the `orders` and `order_items` tables under the `sales` schema.

Listing 2.14 Creating `orders` and `order_items` tables

```
CREATE TABLE sales.orders (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    customer_id INT REFERENCES customers.accounts(id),
    order_date TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10, 2)
);

CREATE TABLE sales.order_items (
    order_id UUID REFERENCES sales.orders(id),
    product_id INT REFERENCES products.catalog(id),
    quantity INT CHECK (quantity > 0),
    price DECIMAL (10, 2),
    PRIMARY KEY (order_id, product_id)
);
```

The `id` column of the `sales.orders` table is a primary key of the `UUID` type, which is a common alternative to numeric primary keys backed by sequences. Postgres's `gen_random_uuid()` function generates a random UUIDv4 value if the application layer doesn't provide one. UUID-based primary keys usually scale better because there is no dependency on sequences, which are frequently used for the automatic generation of numeric IDs.

> **NOTE** Starting with Postgres 18, which was in development at the time of writing, the database introduced support for UUIDv7. This version includes a Unix timestamp in its value, making it more index-friendly because UUIDs generated close in time are stored nearer to each other in the index and can be accessed faster. UUIDv7 values can be generated using the `uuidv7()` function. For compatibility, the `get_random_uuid()` function serves as an alias for `uuidv4()`, which generates UUIDv4 values.

The `price` column of the `sales.order_items` table stores the price of an item at the time of sale. This is needed because the product price can change after the order is placed.

The `sales.order_items` table defines a compound primary key that uses `order_id` and `product_id` to ensure the uniqueness of a record. With such a key, customers can buy different products in a single order, and each product can appear only once in the purchase list.

Having the `orders` and `order_items` tables in place, let's see how multistatement explicit transactions can help us handle customer orders. Imagine that a customer lands on the coffee chain website and decides to buy a bag of Sunrise Blend and a coffee mug. She finishes the checkout process, and our application backend successfully records the order in the system by executing an explicit multistatement transaction.

Listing 2.15 Customer buying two products

```
BEGIN;

INSERT INTO sales.orders (id, customer_id, total_amount)
VALUES ('19a0cffc-8757-453c-a4d2-b554fdc08954', 1, 26.53);

INSERT INTO sales.order_items (order_id, product_id, quantity, price)
VALUES ('19a0cffc-8757-453c-a4d2-b554fdc08954', 1, 1, 16.54),
 ('19a0cffc-8757-453c-a4d2-b554fdc08954', 4, 1, 9.99);

UPDATE products.catalog
SET stock_quantity = stock_quantity - 1
WHERE id IN (1, 4);

COMMIT;
```

The `BEGIN` statement starts an explicit transaction. The first `INSERT` statement adds a new order with the id `19a0cffc-8757-453c-a4d2-b554fdc08954` to the `orders` table. The customer's ID is `1`, and the order total is $26.53 ($16.54 + $9.99).

> **NOTE** We define the UUID value explicitly here just to show what it might look like in practice if we let the `gen_random_uuid()` function generate it automatically.

The next `INSERT` adds two products to the `order_items` table. The first item is a Sunrise Blend bag with `id = 1` and `price = 16.54`. The second item is a branded coffee mug with `id = 4` and a sales price of $9.99. Both items have the same `order_id` value as long as they belong to the same order.

The `UPDATE` statement updates the product quantities in the `products.catalog` table by subtracting 1 from the `stock_quantity` for both products. The `COMMIT` statement completes the transaction.

The transaction from listing 2.15 modifies data across three tables, and Postgres guarantees that this transaction will either be committed successfully as a whole or

rolled back if any of the statements fail. For example, if the transaction has already inserted a new order and its items into the `sales.orders` and `sales.order_items` tables, but the transaction fails to execute the final `UPDATE` because another customer has just purchased the last coffee mug, the entire transaction will be rolled back, leaving no visible trace in the `sales.orders` and `sales.order_items` tables.

Changes that are successfully applied but then rolled back by Postgres don't disappear from the database immediately. The Postgres multiversion concurrency control (MVCC) engine ensures these changes are no longer visible to subsequent application requests and transactions. However, even though the rolled-back changes are no longer visible, they will remain in the database for some time until Postgres performs garbage collection (called a *vacuum*).

2.4.3 Multiversion concurrence control

While some customers are completing their orders with multistatement transactions, others will still be exploring Jake's online store, resulting in many `SELECT` queries over the product catalog. Postgres implements MVCC and uses the read committed isolation level by default to execute these `SELECT` queries concurrently. This means if one customer loads the full product catalog using a `SELECT * FROM products.catalog` statement while another completes the final `UPDATE` from listing 2.15, both customers can proceed without blocking each other.

The goal of the read committed isolation level (the default in Postgres) is to avoid the *dirty read* phenomenon: no single-statement or multistatement transaction should see uncommitted changes from another implicit or explicit transaction. At the same time, if two read-committed transactions attempt to update the same row concurrently, Postgres will block one of them until the other commits to prevent dirty writes—a situation where one transaction would overwrite a value written by another transaction that hasn't committed yet. Let's break this down by using the following transaction, which we are going to execute concurrently from two separate psql connections (don't run this query just yet).

Listing 2.16 Reading and changing the quantity for a product

```
BEGIN;

SELECT stock_quantity FROM products.catalog
WHERE id = 1;

UPDATE products.catalog
SET stock_quantity = stock_quantity - 1
WHERE id = 1;

COMMIT;
```

First, open one more psql connection to Jake's database in another terminal window:

```
docker exec -it postgres psql -U postgres -d coffee_chain
```

And then follow the steps from table 2.1.

Table 2.1 Executing two transactions concurrently

First psql session	Second psql session

Start the first transaction and check the stock quantity for `product` = 1:
 	
`BEGIN;`
 	
`SELECT stock_quantity FROM products.catalog`
`WHERE id = 1;`
 	
The output should be as shown here if you have followed all the previous steps in the chapter:
 	
`stock_quantity`
`----------------`
`199`
`(1 row)`	

Update the product quantity:
 	
`UPDATE products.catalog`
`SET stock_quantity = stock_quantity - 1`
`WHERE id = 1;`	

	Start the second transaction and check the stock quantity of `product` = 1:
	
	`BEGIN;`
	
	`SELECT stock_quantity FROM products.catalog`
	`WHERE id = 1;`
	
	The `SELECT` is not blocked by the first transaction and returns the following result:
	
	`stock_quantity`
	`----------------`
	`199`
	`(1 row)`
	
	Even though the first transaction has already set the stock quantity to 198 (by subtracting 1 from 199), that change has not been committed yet. Therefore, the second transaction still sees 199.

	Update the product quantity:
	
	`UPDATE products.catalog`
	`SET stock_quantity = stock_quantity - 1`
	`WHERE id = 1;`
	
	The operation is blocked because the record of the product with `id` = 1 is locked by the `UPDATE` statement of the first transaction.

Table 2.1 Executing two transactions concurrently (*continued*)

First psql session	Second psql session
Commit the transaction to apply the pending changes: `COMMIT;`	
	The second transaction is now unblocked. Its `UPDATE` will re-read and see the latest `stock_quantity` set by the first transaction, which is 198.
	Commit the transaction: `COMMIT;`

Once both transactions are finished, execute this statement to confirm that the `stock_quantity` is 197:

```
SELECT name, stock_quantity FROM products.catalog WHERE id = 1;
```

The output will be as follows:

```
     name      | stock_quantity
---------------+----------------
 Sunrise Blend |            197
(1 row)
```

Before the transactions began, the `stock_quantity` was 199, and now it's set to 197, indicating that the transactions were successfully executed in parallel without overriding each other's results.

NOTE The `SELECT` statement in listing 2.16 does not affect the outcome of either transaction and can be removed in real-world applications. It is included just to demonstrate the behavior of the read committed isolation level.

The read committed isolation level is the default because it strikes a good balance between consistency and concurrency, allowing you to build high-performance transactional applications without sacrificing data consistency. However, in addition to the dirty read phenomenon that is prevented by the read committed isolation level, the SQL standard also defines other phenomena, such as phantom and nonrepeatable reads. Postgres offers stronger isolation levels—repeatable read and serializable—to address these additional read anomalies.

The official Postgres documentation is an excellent resource that guides you through all phenomena and isolation levels, helping you decide when or if you need to use repeatable reads or serializable isolation levels for your business operations: https://www.postgresql.org/docs/current/transaction-iso.html. In our eCommerce platform, the application backend makes proper use of Postgres transactional capabilities,

ensuring that Jake and other merchants never face data inconsistencies or anomalies in production.

2.5 Joins

Up until now, all the `SELECT` queries we experimented with accessed only one table at a time. With *join queries* (or simply *joins*), we can pull data from several tables at once. All we need to do is choose the tables holding the required data and pair their rows using a join condition.

Let's return to Jake, one of the tenants of our eCommerce SaaS platform, and explore the usage of joins in action. If you'd like to follow along with the steps in this section, make sure you're still connected to the coffee chain's database with psql:

```
docker exec -it postgres psql -U postgres -d coffee_chain
```

It's been an hour since Jake launched the coffee chain's online store, and he already sees the first customers registering and exploring the catalog. Our eCommerce platform empowers every merchant with a sophisticated dashboard that includes analytics related to sales, customer behavior, and other insights to help grow the business. Jake is eager to explore sales-related data, so he opens the sales stats and sees both real-time and historical data.

To compile this sales data, our application backend makes extensive use of joins. For example, to show Jake the top three customers who placed the most orders, the application runs the following query.

Listing 2.17 Showing the top three customers by order volume

```
SELECT c.name, c.id, count(*) as total_orders
FROM customers.accounts c
JOIN sales.orders s ON c.id = s.customer_id
GROUP BY c.id
ORDER BY total_orders DESC
LIMIT 3;
```

The query returns the customer name, ID, and total number of orders by joining the `customers.accounts` and `sales.orders` tables. The `ON` operator defines the join condition that pairs rows from the `accounts` and `orders` tables by matching customer IDs (`c.id = s.customer_id`). Also, we use the `GROUP BY c.id` clause to calculate the total number of orders for each unique customer (`count(*) as total_orders`). Because this is the first hour of sales, the report shows that there is only one order in the system. Still, it's better than nothing!

```
     name      | id | total_orders
---------------+----+--------------
 Alice Johnson |  1 |            1
(1 row)
```

Listing 2.17 is an example of an inner join that returns records with matching values in both tables (the join condition specified in `ON` must find at least one match in both tables). However, our eCommerce platform uses other types of joins as well. For example, Jake sees a report listing customers who haven't placed any orders yet. The application backend executes the following query to provide merchants with those insights.

Listing 2.18 Showing customers with no orders

```
SELECT c.name
FROM customers.accounts c
LEFT JOIN sales.orders s ON c.id = s.customer_id
WHERE s.customer_id IS NULL;
```

The query performs a `LEFT JOIN`, which returns all rows from the left table (`customers.accounts`) and the matched rows from the right table (`sales.orders`). If no matching records are in the right table, `NULL` values are added to the final result.

Then, we use the `WHERE` clause to filter the result, returning only customers with no orders (they didn't have a match in the `sales.orders` table). As of now, only three customers have signed up with Jake's online store, and two of them have not placed any orders yet:

```
     name      |
---------------+
 Bob Smith     |
 Charlie Brown |
(2 rows)
```

Finally, before closing the sales dashboard, Jake notices another report showing statistics about the most popular and least popular products. He understands that this data will become more useful in a few days after more orders are placed. Depending on the insights, he might want to proactively restock popular items and decide what to do with products that have no customer interest. Like the previous reports, this one is also prepared using a `LEFT JOIN` query.

Listing 2.19 Looking at product popularity

```
SELECT c.name, c.category, c.price, SUM(oi.quantity) AS total_sold
FROM products.catalog c
LEFT JOIN sales.order_items oi ON c.id = oi.product_id
GROUP BY c.id
ORDER BY total_sold DESC NULLS LAST, price DESC;
```

Currently, the report shows products with the most sales and highest prices first (`ORDER BY total_sold DESC NULLS LAST, price DESC`). The `NULLS LAST` clause ensures that products with no sales are listed at the end (their `total_sold` column value is set to `NULL`):

```
          name           | category | price | total_sold
-------------------------+----------+-------+------------
 Sunrise Blend           | coffee   | 16.54 |          1
 Sunrise Brew Co. Mug    | mug      |  9.99 |          1
 Sunrise Brew Co. T-Shirt | t-shirt  | 19.99 |
 Morning Glory           | coffee   | 13.99 |
(4 rows)
```

Jake is glad to see that the eCommerce platform provides detailed analytics related to sales and customer behavior. For us, providing all that data was straightforward by relying on joins—one of the most widely used capabilities of Postgres and other relational databases.

2.6 Functions and triggers

Relational databases are not just containers for your application data. They allow you to create and execute comprehensive logic that extends well beyond simple data retrieval or modification. This logic can be implemented using database functions and triggers written in PL/pgSQL (a procedural language specific to Postgres) or your preferred programming language with extensions for Java, JavaScript, Python, or other languages.

Once a function is created, your application layer can call it directly by its name. The next listing defines a simple function implemented in pure SQL.

Listing 2.20 Function that returns the product price

```
CREATE OR REPLACE FUNCTION products.get_product_price(product_id INT)
RETURNS NUMERIC(10, 2) AS $$
    SELECT price
    FROM products.catalog
    WHERE id = product_id;
$$ LANGUAGE sql;
```

The function is named `get_product_price` and is added to the products schema. It accepts `product_id` as an argument and returns the price as a numeric value. Same as with tables, if we don't specify the schema explicitly, the function will be created in the current schema.

Let's connect to Jake's database and see how the function works in practice:

```
docker exec -it postgres psql -U postgres -d coffee_chain
```

First, create the function by copying and pasting the code from listing 2.20. Next, call the function by name to find the price of the product with an ID of 5, which corresponds to a branded t-shirt from Jake's coffee chain:

```
SELECT products.get_product_price(5);
```

The output is as follows:

```
 get_product_price
-------------------
             19.99
(1 row)
```

If you ever need to change the function's implementation, simply update its logic in your preferred editor and re-create it by executing the complete `CREATE OR REPLACE FUNCTION` statement.

The function from listing 2.20 is a simple example with no practical value. So, when and why would you want to use database functions? At least two scenarios come to mind:

- When there's complex business logic tightly coupled with the data that needs to be implemented and maintained by every client application or microservice independently. Instead, you can create a single database function that all clients and microservices can take advantage of. This approach allows the database function to serve as the most efficient implementation of business logic directly within your database. We'll explore an example of such a function in section 2.6.1.
- When you need to deal with complex, multistep business logic that requires multiple round trips between the application and database layers. Implementing the logic as a database function can be more efficient, especially if network latency is high and a large volume of data needs to be transferred between the application and database. Consider a bank managing thousands of savings accounts. On a day when interest needs to be applied, one approach is to read all accounts into the application layer, calculate and apply the interest, and then write the changes back. Alternatively, a single database function could perform all these steps internally, avoiding the need to transfer data back and forth, which saves time and reduces network latency.

NOTE The terms *functions* and *stored procedures* are often used interchangeably, and they are generally considered synonyms—except in Postgres. In Postgres, a stored procedure is a database object similar to a function, which is also used to implement and execute custom logic within the database. However, stored procedures are created using the `CREATE PROCEDURE` statement and do not return a value. Unlike functions, which have been in Postgres from the beginning, stored procedures were introduced in Postgres 11 as a standalone API.

Triggers are a special type of function that you can use when PostgreSQL needs to execute logic before or after data changes occur. Triggers are particularly useful in audit scenarios, where you need to track who made changes in the database, or in event-driven architectures, where a change in the database triggers subsequent actions elsewhere. We'll explore a practical use case for triggers in section 2.6.2.

2.6.1 Functions: A practical example

Imagine sitting at home with your laptop, visiting Jake's coffee chain website or any other online store running on our eCommerce platform. You browse through the product catalog, find something you like, and add it to your shopping cart. The shopping cart icon in the top-right corner now shows the number 1. Then you find another great product and add it to the cart, which updates to indicate two items. Before completing your purchase, you get distracted, close the browser, and leave home. A few

hours later, during a break, you pick up your mobile phone, visit Jake's online store, and see that the shopping cart still shows those two products you wanted to buy while browsing on your laptop at home.

This is a familiar experience, isn't it? In this section, we'll implement a simplified version of a comparable shopping cart experience using database functions. Although this logic can be implemented in the application layer, you'll see that there are numerous corner cases to handle, and several network round trips are required. At the very least, you'll understand how these scenarios can be handled and simplified with PostgreSQL, allowing you to make better choices.

First, let's introduce the concept of a *pending order*, which represents an order that a customer is actively working on. A shopping cart showing two product items is an example of a pending order. We'll reuse our existing `sales.orders` table to track both pending and completed (ordered) orders. The following listing demonstrates how to add the `status` column to the table and sets a check constraint that allows only `pending` and `ordered` as valid values.

Listing 2.21 Adding a `status` column to the `orders` table

```
ALTER TABLE sales.orders
ADD COLUMN status TEXT DEFAULT 'pending'
  CHECK (status in ('pending','ordered'));

UPDATE sales.orders SET status = 'ordered';

ALTER TABLE sales.orders
ADD CONSTRAINT one_pending_order_per_customer
EXCLUDE USING btree (customer_id WITH =)
WHERE (status = 'pending');
```

The first `ALTER TABLE` statement adds the new `status` column, which can be set to either `pending` or `ordered`. If the value is not provided for a new order, its status is set to `pending` by default.

The `UPDATE` statement changes the status to `ordered` for all existing orders. This is necessary because the previous `ALTER TABLE` command set it to `pending`, which is the default value.

The last `ALTER TABLE` statement adds an exclusion constraint to ensure that each customer can have only one pending order at a time. This is necessary because you can have only one shopping cart associated with your account. The `EXCLUDE` clause defines the `(customer_id WITH =)` rule, which means rows with the same `customer_id` must not coexist if they satisfy the `WHERE` condition that follows. In our case, the `WHERE` clause checks for `status = 'pending'`, ensuring that there can't be two records with the same `customer_id` and a `pending` status. The `USING btree` clause specifies that Postgres will use a B-tree index to validate the constraint with logarithmic time complexity.

Next, let's implement the `order_add_item(customer_id, product_id, quantity)` function, which adds or updates the shopping cart associated with the customer. The

function will be added to the sales schema. Listing 2.22 provides an implementation that addresses most corner cases.

The function is implemented in PL/pgSQL that supports all the basic constructs and operators you'd expect from a programming language, and it's easy to learn and use. However, if PL/pgSQL isn't quite your style, Postgres offers various extensions for Java, Python, Rust, and other major programming languages, allowing you to develop database functions in a language of your choice.

Listing 2.22 Implementing the `order_add_item` function

```
CREATE OR REPLACE FUNCTION sales.order_add_item(customer_id_param INT,
  product_id_param INT, quantity_param INT)
RETURNS TABLE (order_id UUID, prod_id INT,
  quantity INT, prod_price DECIMAL) AS $$
DECLARE
    pending_order_id UUID;
BEGIN
    SELECT id INTO pending_order_id
    FROM sales.orders
    WHERE customer_id = customer_id_param
      AND status = 'pending'
    LIMIT 1;

    IF pending_order_id IS NULL THEN
        INSERT INTO sales.orders (customer_id, status)
        VALUES (customer_id_param, 'pending')
        RETURNING id INTO pending_order_id;
    END IF;

    MERGE INTO sales.order_items AS oi
    USING (SELECT id, price FROM products.catalog
      WHERE id = product_id_param) AS prod
    ON oi.product_id = prod.id AND oi.order_id = pending_order_id
    WHEN MATCHED THEN
        UPDATE SET quantity = quantity_param
    WHEN NOT MATCHED THEN
        INSERT (order_id, product_id, quantity, price)
        VALUES (pending_order_id, prod.id, quantity_param, prod.price);

    RETURN QUERY
    SELECT oi.order_id, oi.product_id, oi.quantity, oi.price as prod_price
    FROM sales.order_items as oi
    WHERE oi.order_id = pending_order_id;
END;
$$ LANGUAGE plpgsql;
```

The function logic works as follows:

- It accepts three arguments (a customer ID, a product ID, and the quantity of the product) and returns a table that is effectively a list of all product items from the pending order.

- It declares the `pending_order_id` variable that holds the ID of the pending order associated with the customer.
- The first `SELECT` statement checks whether a pending order already exists for the customer. If found, the order ID is assigned to the `pending_order_id` variable.
- If the `IF pending_order_id IS NULL` condition resolves to `true`, then no pending order exists, and the function creates a new one. The `sales.orders` table automatically generates the UUID using the built-in Postgres `gen_random_uuid()` function, and the `RETURNING` clause lets us capture that generated `id` and store it in the `pending_order_id` variable.
- The `MERGE` statement is used to either add the product item or update its quantity in the order. Finally, the `RETURN QUERY` returns all items from the pending order, which can be displayed in the shopping cart on the application end.

Now, let's see how the application backend can take advantage of the function. Imagine that a loyal customer of Jake's coffee chain, Charlie Brown, signs in to the store interface, and the system executes the following query to get the customer's details:

```
SELECT id, name FROM customers.accounts WHERE name = 'Charlie Brown';
```

The output is as follows:

```
 id |     name
----+---------------
  3 | Charlie Brown
(1 row)
```

Charlie browses the product catalog and adds two bags of Morning Glory coffee to his cart. The internal ID of this coffee flavor is `3`:

```
SELECT id, name FROM products.catalog WHERE name = 'Morning Glory';

 id |     name
----+---------------
  3 | Morning Glory
(1 row)
```

Our application backend uses Charlie's ID, the product ID, and the desired quantity to execute the `order_add_item` function.

Listing 2.23 Adding a product to the shopping cart

```
SELECT * FROM sales.order_add_item(
    customer_id_param => 3,
    product_id_param => 3,
    quantity_param => 2
    );
```

Postgres successfully executes the function by creating a new pending order for Charlie and adding two items of Morning Glory to it. It then returns the following order details to the application layer, which updates Charlie's shopping cart:

```
               order_id               | prod_id | quantity | prod_price
--------------------------------------+---------+----------+------------
 e900fb15-aa65-4eea-bd52-1abe8d5832b6 |       3 |        2 |      13.99
(1 row)
```

NOTE In the function call from listing 2.23, we use *named notation*, where each argument name is separated from its value using the `=>` operator. These names are optional and are added for readability purposes. For example, we could call the function `SELECT * FROM sales.order_add_item(3, 3, 2)` to achieve the same results.

Before checking out, Charlie decides to add a branded t-shirt to his cart. The internal ID of the t-shirt is 5:

```
SELECT id, name FROM products.catalog WHERE id = 5;

 id |           name
----+--------------------------
  5 | Sunrise Brew Co. T-Shirt
(1 row)
```

Charlie clicks the Add button next to the t-shirt, and the application backend executes the `order_add_item` function again to add the new product:

```
SELECT * FROM sales.order_add_item(
    customer_id_param => 3,
    product_id_param => 5,
    quantity_param => 1
    );
```

Postgres locates the existing pending order and adds the t-shirt to it. The database then sends a complete list of items in the order back to the application, which updates the shopping cart's UI accordingly:

```
               order_id               | prod_id | quantity | prod_price
--------------------------------------+---------+----------+------------
 e900fb15-aa65-4eea-bd52-1abe8d5832b6 |       3 |        2 |      13.99
 e900fb15-aa65-4eea-bd52-1abe8d5832b6 |       5 |        1 |      19.99
(2 rows)
```

NOTE Postgres executes a function atomically and transactionally. This means if the function fails at any step during its execution, all previous changes made within that function are rolled back, ensuring data integrity and consistency.

As a result, Charlie now has a pending order that needs to be checked out:

```
SELECT id, customer_id, status FROM sales.orders WHERE customer_id = 3;

                  id                  | customer_id | status
--------------------------------------+-------------+---------
 e900fb15-aa65-4eea-bd52-1abe8d5832b6 |           3 | pending
(1 row)
```

Next, let's implement the `order_checkout` function, which the application backend can use to complete the order and clear the shopping cart. The function is added to the sales schema.

Listing 2.24 Implementing the `order_checkout` function

```
CREATE OR REPLACE FUNCTION sales.order_checkout(customer_id_param INT)
RETURNS TABLE (order_id UUID, customer_id INT, total_amount DECIMAL) AS $$
DECLARE
    pending_order_id UUID;
    final_total_amount DECIMAL := 0;
BEGIN
    SELECT id INTO pending_order_id
    FROM sales.orders as o
    WHERE o.customer_id = customer_id_param
      AND status = 'pending'
    LIMIT 1;

    IF pending_order_id IS NULL THEN
        RAISE EXCEPTION 'No pending order found for customer %',
          customer_id_param;
    END IF;

    SELECT SUM(oi.quantity * oi.price) INTO final_total_amount
    FROM sales.order_items oi
    WHERE oi.order_id = pending_order_id;

    UPDATE sales.orders
    SET status = 'ordered',
        total_amount = final_total_amount,
        order_date = CURRENT_TIMESTAMP
    WHERE id = pending_order_id;

    UPDATE products.catalog
    SET stock_quantity = stock_quantity - oi.quantity
    FROM sales.order_items oi
    WHERE products.catalog.id = oi.product_id
      AND oi.order_id = pending_order_id;

    RETURN QUERY
    SELECT o.id, o.customer_id, o.total_amount
    FROM sales.orders as o
    WHERE o.id = pending_order_id;
END;
$$ LANGUAGE plpgsql;
```

The function execution flow is as follows:

- It accepts the customer ID as an argument to perform the checkout. The first `SELECT` statement locates the pending order associated with the customer. If no pending order is found (`IF pending_order_id IS NULL`), an exception is raised.
- Otherwise, the next `SELECT` statement calculates the total amount of the order and assigns this value to the `final_total_amount` variable.
- The first `UPDATE` statement changes the order `status` to `ordered` and sets the total amount. The following `UPDATE` statement subtracts the purchased product quantities from the `products.catalog` table. This step can fail if any product is no longer available in the desired quantity, which can occur with popular items. The `products.catalog` table has a check constraint that validates the quantity (refer to listing 2.1 for details). Once the order is completed, the `RETURN QUERY` statement returns an order summary to the application layer.

Let's give the checkout function a try. Imagine that Charlie clicks the Buy button and the eCommerce platform executes the following code snippet:

```
SELECT * FROM sales.order_checkout(customer_id_param => 3);
```

Postgres locates the order and successfully completes it by sending the confirmation. The customer will be anticipating delivery with excitement!

```
              order_id                | customer_id | total_amount
--------------------------------------+-------------+-------------
e900fb15-aa65-4eea-bd52-1abe8d5832b6 |           3 |        47.97
(1 row)
```

TIP Use the `SELECT * FROM function_name` format when calling the function to ensure that psql displays the column names alongside the returned values. However, if the column names are not required, you can simplify the function call to `SELECT sales.order_checkout(3)`.

2.6.2 Triggers: A practical example

Triggers are also database functions that PostgreSQL executes in response to data changes or database events. Currently, when a pending order is created and the customer continues adding various items to the shopping cart, the order's total amount is not updated immediately. Instead, the total is calculated when the application executes the `order_checkout` function from listing 2.24.

For example, imagine that another customer of the coffee chain's online store, Bob Smith, decides to purchase a few items. Bob's internal ID is 2:

```
SELECT id, name FROM customers.accounts WHERE name = 'Bob Smith';

 id |   name
----+-----------
  2 | Bob Smith
(1 row)
```

Bob adds a bag of Sunrise Blend coffee, which has an ID of 1, to his cart:

```
SELECT * FROM sales.order_add_item(
    customer_id_param => 2,
    product_id_param => 1,
    quantity_param => 1);
```

The function call returns the following result:

```
               order_id               | prod_id | quantity | prod_price
--------------------------------------+---------+----------+------------
 34221a65-df20-4396-b9f4-3e3cf328f9d2 |       1 |        1 |      16.54
(1 row)
```

The database creates the pending order and adds the bag of coffee, showing that the current price for the item is $16.54. Note that if you follow these steps, the `order_id` value will differ for you because it is automatically generated by Postgres.

However, if you execute this query against the `sales.orders` table, you'll notice that the `total_amount` for the order hasn't been calculated. Instead, you'll see an empty value in the output:

```
SELECT id, status, total_amount FROM sales.orders
WHERE id = '34221a65-df20-4396-b9f4-3e3cf328f9d2';

                  id                  | status  | total_amount
--------------------------------------+---------+--------------
 34221a65-df20-4396-b9f4-3e3cf328f9d2 | pending |
(1 row)
```

To address this oversight, let's create a trigger that automatically updates the `orders.total_amount` column whenever an item is added, updated, or removed from the pending order. This ensures that the total amount of the order remains accurate without requiring manual recalculation each time an order item is modified via the `order_add_item` function or by other means.

First, use the following code to create a trigger function that adjusts the pending order's total amount whenever the shopping cart is changed. The function is created in the sales schema.

Listing 2.25 Trigger function that updates the order's total amount

```
CREATE OR REPLACE FUNCTION sales.update_order_total()
RETURNS TRIGGER AS $$
BEGIN
```

```
    UPDATE sales.orders
    SET total_amount = (
        SELECT COALESCE(SUM(oi.quantity * oi.price), 0)
        FROM sales.order_items oi
        WHERE oi.order_id = COALESCE(NEW.order_id, OLD.order_id)
    )
    WHERE id = COALESCE(NEW.order_id, OLD.order_id) AND status = 'pending';

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

The trigger calculates the new total amount for the order based on the current items in the shopping cart. Upon a trigger execution, Postgres passes the `NEW` and `OLD` variables to the function. As the names suggest, `NEW` contains the updated version of the item record, and `OLD` holds the previous version of the row. The function uses `COALESCE` to select the `order_id` from either the `NEW` variable (which is non-NULL for `INSERT/UPDATE` operations) or the `OLD` variable (which is non-NULL for `DELETE` operations). If the shopping cart is empty—for example, `SUM(oi.quantity * oi.price)` resolves to `NULL`—the `COALESCE` function sets `total_amount` to `0`. In the end, the function returns `NEW` to allow Postgres to store the new version of the row without any modifications.

Now create the trigger, which will call this function after every `INSERT`, `UPDATE`, or `DELETE` operation on the `sales.order_items` table.

Listing 2.26 Trigger that updates the order's total amount

```
CREATE TRIGGER trigger_update_order_total
AFTER INSERT OR UPDATE OR DELETE ON sales.order_items
FOR EACH ROW
EXECUTE FUNCTION sales.update_order_total();
```

The trigger works behind the scenes to keep the total amount of the pending orders accurate. For instance, if Bob changes his mind and decides to buy two bags of Sunrise Blend coffee, the application backend makes the following call with `quantity_param` set to `2`:

```
SELECT * FROM sales.order_add_item(
    customer_id_param => 2,
    product_id_param => 1,
    quantity_param => 2);
```

With the query producing the result as follows:

```
               order_id               | prod_id | quantity | prod_price
--------------------------------------+---------+----------+------------
 34221a65-df20-4396-b9f4-3e3cf328f9d2 |       1 |        2 |      16.54
(1 row)
```

The trigger responds to this update automatically and sets `total_amount` for Bob's order to \$33.08, which is \$16.54 × 2:

```
SELECT id, status, total_amount FROM sales.orders
WHERE id = '34221a65-df20-4396-b9f4-3e3cf328f9d2';

                  id                  | status  | total_amount
--------------------------------------+---------+--------------
 34221a65-df20-4396-b9f4-3e3cf328f9d2 | pending |        33.08
(1 row)
```

Right after that, Bob remembers that he might run out of coffee quickly because his extended family is visiting next week. He adds two more bags of coffee to the list, resulting in a total of four bags. The application sends another update to the database to adjust the shopping cart:

```
SELECT * FROM sales.order_add_item(
    customer_id_param => 2,
    product_id_param => 1,
    quantity_param => 4);

               order_id               | prod_id | quantity | prod_price
--------------------------------------+---------+----------+------------
 34221a65-df20-4396-b9f4-3e3cf328f9d2 |       1 |        4 |      16.54
(1 row)
```

The trigger then updates the total order amount accordingly to $66.16 ($16.54 × 4):

```
SELECT id, status, total_amount FROM sales.orders
WHERE id = '34221a65-df20-4396-b9f4-3e3cf328f9d2';

                  id                  | status  | total_amount
--------------------------------------+---------+--------------
 34221a65-df20-4396-b9f4-3e3cf328f9d2 | pending |        66.16
(1 row)
```

TIP If your application needs Postgres to send notifications in response to data changes or database events, explore its `LISTEN/NOTIFY` capabilities. An application can register for notifications on a specific channel using the `LISTEN` command. When a particular event occurs, the database can asynchronously notify all registered client sessions with the `NOTIFY` counterpart. This capability can also be used if one client session needs to send notifications to others. We'll see this feature in action in chapter 11, where we learn to use Postgres as a message queue.

With his order finalized, Bob proceeds to check out and pay for the order. The application backend completes the order by executing the `order_checkout(2)` function, where 2 is Bob's internal identifier:

```
SELECT * FROM sales.order_checkout(2);

               order_id               | customer_id | total_amount
--------------------------------------+-------------+------------
34221a65-df20-4396-b9f4-3e3cf328f9d2  |            2|       66.16
(1 row)
```

2.7 Views

A *view* is essentially a named query that returns data in a tabular format. Imagine having a complex SQL query that needs to be executed by various applications and other clients. Instead of copying/pasting and maintaining that query across different clients, you can create a view at the database level and let the applications call the view by name. When called, the view executes that complex query, which previously had to be managed at the application layer.

In section 2.5, we discussed how our eCommerce platform empowers Jake and other merchants with a sophisticated dashboard that includes analytics on sales, customer behavior, and other insights to help grow their businesses. The application backend uses joins to aggregate various statistics that Jake can use. Now we've decided to simplify our application logic by implementing a few sales reports using views.

One of the reports provides a summary of sales for each product, including the total quantity sold and revenue generated. Currently, when Jake loads the report, the application backend executes the following query against Postgres.

Listing 2.27 Sales report summary

```
SELECT
    c.name AS product_name,
    c.category,
    SUM(oi.quantity) AS total_quantity_sold,
    SUM(oi.quantity * oi.price) AS total_revenue
FROM products.catalog c
LEFT JOIN sales.order_items oi ON c.id = oi.product_id
GROUP BY c.id
ORDER BY total_quantity_sold DESC, total_revenue DESC;
```

The query returns the following result (the output may differ on your end if you haven't followed along with all the previous listings in the chapter):

```
       product_name       | category | total_quantity_sold | total_revenue
--------------------------+----------+---------------------+---------------
 Sunrise Blend            | coffee   |                   5 |         82.70
 Morning Glory            | coffee   |                   2 |         27.98
 Sunrise Brew Co. T-Shirt | t-shirt  |                   1 |         19.99
 Sunrise Brew Co. Mug     | mug      |                   1 |          9.99
(4 rows)
```

Next, we take the query from listing 2.27 and turn it into a view.

Listing 2.28 Creating a view for the sales report summary

```
CREATE VIEW sales.product_sales_summary AS
SELECT
    c.name AS product_name,
    c.category,
    SUM(oi.quantity) AS total_quantity_sold,
```

```
    SUM(oi.quantity * oi.price) AS total_revenue
FROM products.catalog c
LEFT JOIN sales.order_items oi ON c.id = oi.product_id
GROUP BY c.id
ORDER BY total_quantity_sold DESC, total_revenue DESC;
```

The view is created with the `CREATE VIEW` command and reuses the original SQL query that puts together the report. It's named `product_sales_summary` and added to the sales schema. Same as with tables, if we don't specify the schema explicitly, the view will be created in the current schema.

Once the view is created, we can update our application logic and start generating the report with a query as simple as this:

```
SELECT * FROM sales.product_sales_summary;
```

The query calls the view, which returns the sales summary in a tabular format:

```
       product_name        | category | total_quantity_sold | total_revenue
---------------------------+----------+---------------------+---------------
 Sunrise Blend             | coffee   |                   5 |         82.70
 Morning Glory             | coffee   |                   2 |         27.98
 Sunrise Brew Co. T-Shirt | t-shirt  |                   1 |         19.99
 Sunrise Brew Co. Mug      | mug      |                   1 |          9.99
(4 rows)
```

Because the view returns results in a tabular format, we can further filter its data. Suppose Jake is only interested in the stats for coffee. He goes to the UI and selects the "coffee" category from a drop-down list near the report. The application backend then makes the following call to the database, asking the view to return data only for products in the "coffee" category:

```
SELECT * FROM sales.product_sales_summary WHERE category='coffee';
```

The following data is returned and shown to Jake:

```
 product_name  | category | total_quantity_sold | total_revenue
---------------+----------+---------------------+---------------
 Sunrise Blend | coffee   |                   5 |         82.70
 Morning Glory | coffee   |                   2 |         27.98
(2 rows)
```

Views created with the `CREATE VIEW` command are not physically materialized, meaning the query is executed every time the view is called by the application. If you'd like to cache the result of a view and refresh it only periodically, you can use materialized views. As the name suggests, a *materialized view* executes the underlying query and persists the result. All clients will see the same result until the view is explicitly refreshed. This is useful for queries that consume a lot of resources or for use cases where up-to-date data isn't required.

To create a materialized view, you just need to add `MATERIALIZED` to the `CREATE VIEW` statement. Imagine that our eCommerce platform uses the following materialized view to generate sales numbers for each month.

Listing 2.29 Creating a view for monthly sales

```
CREATE MATERIALIZED VIEW sales.monthly_sales_summary AS
SELECT
    date_trunc('month', o.order_date) AS sales_month,
    SUM(oi.quantity * oi.price) AS total_revenue,
    COUNT(DISTINCT(o.id)) AS total_orders
FROM sales.orders o
JOIN sales.order_items oi ON o.id = oi.order_id
GROUP BY sales_month
ORDER BY sales_month;
```

When Jake looks at the monthly report, the application backend executes the query against the materialized view:

```
SELECT * FROM sales.monthly_sales_summary;
```

Considering that these are just the first hours after Jake launched his online store, the report isn't looking too bad:

```
     sales_month     | total_revenue | total_orders
---------------------+---------------+--------------
 2024-09-01 00:00:00 |        140.66 |            3
(1 row)
```

A few minutes later, Charlie Brown, a loyal customer, returns to the website and buys three bags of Sunrise Blend and two bags of Morning Glory. The eCommerce platform uses the database functions from section 2.6 to add the products to the shopping cart and check out the order. Let's follow the steps in this listing to reproduce this on our end.

Listing 2.30 Buying two flavors of coffee

```
SELECT sales.order_add_item(
    customer_id_param => 3,
    product_id_param => 1,
    quantity_param => 3
    );

SELECT sales.order_add_item(
    customer_id_param => 3,
    product_id_param => 3,
    quantity_param => 2
    );

SELECT * FROM sales.order_checkout(customer_id_param => 3);
```

Adding the execution flow looks as follows:

- The first `order_add_item` call adds three bags of Sunrise Blend (`product_id_param => 1`). The second call to the `order_add_item` function adds two bags of Morning Glory (`product_id_param => 3`).
- The final call to the `order_checkout` function completes the order.

The `order_checkout` function reports the total sales amount as $77.60:

```
               order_id               | customer_id | total_amount
--------------------------------------+-------------+-------------
 4fecee1d-3d07-420b-9dbd-6f6b3a1b35f4 |           3 |        77.60
(1 row)
```

But if Jake goes back to the analytics dashboard and checks the monthly report again, those additional $77.60 of sales are not reflected yet:

```
SELECT * FROM sales.monthly_sales_summary;

     sales_month     | total_revenue | total_orders
---------------------+---------------+--------------
 2024-09-01 00:00:00 |        140.66 |            3
(1 row)
```

The report still shows $140.66, just as it did before Charlie Brown placed the new order. This happens because the materialized view doesn't refresh its persisted data unless explicitly asked. Suppose there's a Refresh button near the monthly report, and Jake clicks it. The application backend executes the following query to refresh the materialized view and then queries it once more to pull the latest data.

Listing 2.31 Refreshing and querying a materialized view

```
REFRESH MATERIALIZED VIEW sales.monthly_sales_summary;

SELECT * FROM sales.monthly_sales_summary;
```

This time, Jake sees an up-to-date monthly summary that includes the latest sale amount of $77.60:

```
     sales_month        | total_revenue | total_orders
------------------------+---------------+--------------
 2024-09-01 00:00:00+00 |        218.26 |            4
(1 row)
```

TIP In addition to providing users with controls to update materialized views, you might want to refresh them periodically using an extension like `pg_cron` or in response to certain events with triggers.

Jake is happy with the stats produced by our eCommerce platform, and we've learned that both regular and materialized views can be valuable tools in our developer toolbox. Views help us simplify application logic by letting the database maintain and execute complex queries.

2.8 Roles and access control

Writing code is a captivating experience. We translate our thoughts into new lines of code, compile the app, and watch as our ideas come to life when it runs—this is what makes software development so addictive. But sometimes, while fully immersed in coding, we might overlook the security side of our application. Security isn't the most exciting aspect of software development, but it's extremely important. Once the app is in production, even a small security oversight can lead to severe consequences.

We've already mastered using various security frameworks at the application layer that first properly authenticate our users and then authorize them to perform specific actions. However, handling security just at the application layer isn't enough. Once a user signs in and starts interacting with the app, the logic will open a database connection and execute SQL queries on the user's behalf. It matters a lot which database role or user is used to open that connection. If the database role has a broad set of permissions that goes beyond what the application allows its users to do, that's already a problem to worry about. If an application user's account is compromised, hackers could gain access to the database and perform actions far beyond what the application logic was intended to permit.

In Postgres, roles are used to define and manage database-level access permissions. You can think of a *role* as either a database user or a group of users, depending on how the role is configured. So, when someone says that user X is used to connect to Postgres, it technically means role X is being used for database connectivity.

Roles allow you to fine-tune access control, making it easy to grant or restrict permissions at the database level, ensuring that application users have exactly the access they need—and nothing more.

Up to this point, we've been using the `postgres` role for all our database connections. For instance, we connected to Jake's coffee chain database using the `postgres` role (as specified by the `-U postgres` parameter):

```
docker exec -it postgres psql -U postgres -d coffee_chain
```

Once connected, you can run the `\du` meta-command in psql to see a list of all existing database roles and their associated permissions:

```
\du

                                  List of roles
 Role name |                         Attributes
-----------+------------------------------------------------------------
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS
```

Currently, `postgres` is the only role configured for our Postgres container. Postgres personally sets up this role during the initial configuration of the database server. It has a broad range of attributes that allow us to perform any action on the database server, from creating and modifying tables to configuring and removing roles or database objects.

Although it's fine to use the `postgres` role for quick experiments or early development, you should avoid using it as your application starts to take shape. Instead, create specific database roles for different types or groups of users. Let's return to our eCommerce SaaS platform and see how this works in practice.

Our platform enables thousands of merchants to launch online stores within a week and start selling goods worldwide. As discussed in section 2.1, the platform is an example of a multitenant application that can store data for various tenants (merchants) on the same Postgres database server. Therefore, at a minimum, we need to define database roles that allow merchants to connect and interact only with their own databases, without access to others.

Currently, our Postgres container hosts databases for two different merchants. Here is a truncated output of the psql `\l` command:

```
\l

     Name      |  Owner   |
---------------+----------+
 brewery       | postgres |
 coffee_chain  | postgres |
```

Jake owns the `coffee_chain` database, and the `brewery` database belongs to Susannah, Jake's friend, who introduced him to our eCommerce platform. Despite their friendship, Jake and Susannah's business data and online stores need to operate in complete isolation from each other.

To achieve this, let's create a database role for Jake that will allow him to update the product catalog, view sales statistics, and perform a broad range of other business-related operations. At the same time, the role will restrict Jake from making changes to the database structure, which is a responsibility of our eCommerce platform. When Jake signs into his store, the application backend will use this role to connect to the `coffee_chain` database in Postgres and perform actions on Jake's behalf. Here's how to create the role and associate it with the coffee chain's database.

Listing 2.32 Creating an admin-level role for the coffee chain

```
CREATE ROLE coffee_chain_admin WITH LOGIN PASSWORD 'password';

GRANT CONNECT ON DATABASE coffee_chain TO coffee_chain_admin;

REVOKE CONNECT ON DATABASE coffee_chain FROM PUBLIC;
```

These statements do the following:

- The CREATE ROLE statement adds the role named coffee_chain_admin and allows it to be used for database connectivity by setting the LOGIN attribute. Any role with the explicitly granted LOGIN attribute can connect to the database, and such roles are usually referred to as *database users*.
- The GRANT statement allows the role to connect to the coffee_chain database.
- The REVOKE statement revokes the connect privilege for all other roles to the database. In Postgres, PUBLIC is a default role that includes all users. Even though the PUBLIC role also includes the coffee_chain_admin role, the access for the latter isn't revoked because it was explicitly granted earlier with the GRANT statement.

Next, use the following code to allow Jake to query and manipulate data in any table of his database.

Listing 2.33 Setting up permissions for the role

```
GRANT USAGE ON SCHEMA public TO coffee_chain_admin;
GRANT USAGE ON SCHEMA products TO coffee_chain_admin;
GRANT USAGE ON SCHEMA customers TO coffee_chain_admin;
GRANT USAGE ON SCHEMA sales TO coffee_chain_admin;

GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES
  IN SCHEMA public TO coffee_chain_admin;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES
  IN SCHEMA products TO coffee_chain_admin;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES
  IN SCHEMA customers TO coffee_chain_admin;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES
  IN SCHEMA sales TO coffee_chain_admin;

GRANT USAGE, SELECT ON ALL SEQUENCES
  IN SCHEMA products TO coffee_chain_admin;
GRANT USAGE, SELECT ON ALL SEQUENCES
  IN SCHEMA customers TO coffee_chain_admin;
GRANT USAGE, SELECT ON ALL SEQUENCES
  IN SCHEMA sales TO coffee_chain_admin;
```

The listing execution flow is as follows:

- The first four GRANT USAGE ON SCHEMA statements allow the role to access public, products, and other schemas in the coffee_chain database (which is a current database we're connected to with the postgres role).
- The next four GRANT SELECT, INSERT, UPDATE, DELETE statements allow the role to query and manipulate data in all tables across all schemas within the coffee_chain database.
- The last three GRANT USAGE, SELECT ON ALL SEQUENCES statements allow the role to generate IDs with sequences used by various tables. For instance, if Jake is going to add a new product to the catalog, his database role needs to be able to generate an ID for the product.

Finally, let's ensure that merchants like Jake cannot connect to and work with other databases, including the `brewery` database that belongs to Susannah.

Listing 2.34 Revoking access on the `brewery` and `postgres` databases

```
REVOKE CONNECT ON DATABASE brewery FROM PUBLIC;
REVOKE CONNECT ON DATABASE postgres FROM PUBLIC;
```

Because the `PUBLIC` role in Postgres includes all roles (including the one just created for Jake), you can execute the command from listing 2.34 once for every database to ensure that, by default, no one can connect to it. This setup will require you to use an explicit `GRANT CONNECT ON DATABASE database TO role` command for each role that needs access to a specific database.

> **NOTE** The number of commands we used to configure just one role for Jake might seem daunting, and it still might not cover all corner cases. But don't worry: once you dive deeper into the topic, you'll find that navigating and using roles and related capabilities in Postgres is straightforward. For example, in our eCommerce platform, we can create a group role that defines a standard set of `REVOKE` and `GRANT` commands applicable to every merchant and their databases. Each merchant-specific role can then be added to the group role, inheriting these permissions.

Now, let's try connecting to Postgres using Jake's role to validate the scope of the defined permissions:

```
docker exec -it postgres psql -U coffee_chain_admin -d brewery
```

Because Jake is restricted to working only with his database, he won't be able to use his role to connect to Susannah's `brewery` database (note the `-d` parameter, which specifies the database, and the `-U` parameter, which specifies the role):

```
psql: error: connection to server on socket
"/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL:  permission denied for database "brewery"
DETAIL:  User does not have CONNECT privilege.
```

Similarly, if the `coffee_chain_admin` role is used to connect to the default `postgres` database, Postgres will generate an error again:

```
docker exec -it postgres psql -U coffee_chain_admin -d postgres
```

```
psql: error: connection to server on socket
"/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL:  permission denied for database "postgres"
DETAIL:  User does not have CONNECT privilege.
```

However, the connection is successfully established when we use the role to connect to Jake's `coffee_chain` database:

```
docker exec -it postgres psql -U coffee_chain_admin -d coffee_chain
```

This time, Postgres lets Jake connect to his coffee chain's database:

```
psql (17.2 (Debian 17.2-1.pgdg120+1))
Type "help" for help.

coffee_chain=>
```

Next, to ensure that Jake has read access to his data, let's select products from the `catalog` table:

```
SELECT name, category, price FROM products.catalog;

           name           | category | price
--------------------------+----------+-------
 Sunrise Brew Co. Mug     | mug      |  9.99
 Sunrise Brew Co. T-Shirt | t-shirt  | 19.99
 Morning Glory            | coffee   | 13.99
 Sunrise Blend            | coffee   | 16.54
(4 rows)
```

Additionally, let's confirm that Jake can modify the database's data by updating the price of the branded coffee mug:

```
UPDATE products.catalog
SET price = 10.50
WHERE name = 'Sunrise Brew Co. Mug';
```

The operation succeeded, and the new price for the coffee mug is set to $10.50:

```
SELECT name, price
FROM products.catalog
WHERE name = 'Sunrise Brew Co. Mug';

         name         | price
----------------------+------
 Sunrise Brew Co. Mug | 10.50
(1 row)
```

Even though our eCommerce platform doesn't include any logic that would allow Jake to manipulate the database structure, his account could be compromised. In such cases, let's ensure that Jake's role cannot be used to drop existing tables:

```
DROP TABLE products.catalog;

ERROR:  must be owner of table catalog
```

Also, Postgres will generate a similar error if an attempt is made to create a new table:

```
CREATE TABLE test (id int);

ERROR:  permission denied for schema public
LINE 1: CREATE TABLE test (id int);
```

Both commands failed because the `coffee_chain_admin` role is not the owner of the `coffee_chain` database or its objects, such as tables, schemas, or views. If we execute the psql `\l` command, we'll see that the owner is the `postgres` role, which we used to create the database structure for Jake and other tenants of the eCommerce platform:

```
\l

    List of databases
     Name     |  Owner   |
 brewery      | postgres |
 coffee_chain | postgres |
 postgres     | postgres |
```

Because the `postgres` role has a broad set of permissions, its usage should be limited to individuals with admin-level access to the Postgres server or application components that need to initialize or modify the database structure.

Overall, Jake's role is in good shape, and our application backend will use it for database connectivity whenever Jake signs in to the coffee store. This same role can also be used for managers who assist Jake in running the day-to-day operations of the coffee chain. For the coffee chain's customers, we can create a dedicated database role with even more restricted access to the data and tables.

Summary

- Postgres supports all the standard capabilities you would expect from a relational database.
- Databases, schemas, and tables are essential building blocks that let you easily map the database structure to your application architecture.
- SQL is a highly flexible language that allows you to query and manipulate individual tables or join data stored across multiple tables.
- Postgres's transactional and data integrity capabilities ensure that your data remains accurate and consistent at all times.
- Database functions and triggers enable you to create and execute application logic directly within the database.
- With roles, you can define and manage database-level access at a granular level.

3 Modern SQL

This chapter covers

- Why it's important to learn modern SQL
- Using common table expressions
- Processing hierarchical or tree-like data
- Using window functions to perform calculations

SQL was invented in the early 1970s, predating the creation of Altair, the first personal computer. Despite being invented decades ago, SQL, like many human languages, continues to evolve to meet the demands of the modern world.

Most people, however, are still familiar with SQL as defined by the SQL-92 standard, which fully completed the original relational model and idea. But much has changed since the introduction of that standard. The scope of relational databases has expanded far beyond the relational model, and these changes have been reflected in subsequent versions of the language. Modern SQL introduces new capabilities that not only make the language more readable and easier to follow but also simplify complex calculations and enable working with unstructured and semistructured data.

Let's explore the modern SQL capabilities of Postgres as we build a music streaming service that thousands of people use to listen to their favorite songs and artists. We'll see how modern SQL makes it easy to analyze user preferences, song popularity, and the other data the service tracks.

3.1 What is modern SQL?

The term *modern SQL* was coined by Markus Winand relatively recently. Markus is a well-known database expert who puts effort into raising awareness and adoption of the latest SQL capabilities added to the specification *after* the SQL-92 standard. Markus describes the significance of these changes on his website dedicated to the modern SQL (https://modern-sql.com/) as follows:

> *Since 1999, SQL is not limited to the relational model anymore. Back then, ISO/IEC 9075 (the "SQL standard") added arrays, objects, and recursive queries. In the meantime, the SQL standard has grown five times bigger than SQL-92. In other words: relational SQL is only about 20% of modern SQL.*

Overall, under modern SQL, we can assume a category of features and capabilities that were introduced after the SQL-92 standard and allow us to use Postgres beyond the classic relational model and its traditional use cases. The modern SQL supports new data types, functions, and other capabilities that let us work not only with structured data but also with unstructured and semi-structured data:

- *Data types for unstructured data*—The relational model is based on structured data and normalization. Data must be properly structured, split into atomic pieces, and related to each other. However, today's SQL databases support arrays, JSON objects, and custom composite types, allowing us to store and process unstructured data efficiently.
- *Capabilities for modern workloads*—SQL has a reputation for being verbose, especially when writing nontrivial queries. However, modern SQL capabilities such as common table expressions (CTEs) and window functions allow us to perform complex operations and transformations with concise and readable queries. Additionally, features like recursive queries and property graphs simplify working with hierarchical, tree-like, and graph structures.

Considering the benefits of modern SQL capabilities, why isn't it in every developer's toolbox yet? There are at least a few reasons:

- *Stickiness of gained knowledge*—Some developers learned SQL many years ago and mastered the SQL-92 version of the language for various data processing tasks. Even if their SQL queries are verbose or less efficient, the tasks are still solvable. As a result, many people continue doing things the way they originally learned.
- *Object-relational mapping (ORM) frameworks*—Some developers fully rely on ORM frameworks as a layer between their application and the database. They trust the ORM framework to generate SQL queries, believing it knows the best way to query or manipulate data. However, ORM frameworks are designed for generic

workloads and might not recognize when to use efficient window functions instead of cumbersome self-joins in your specific use cases.

The goal of this chapter is to make modern SQL part of your developer toolbox. We'll learn how to take advantage of CTEs, recursive queries, and window functions in Postgres. In chapters 5 and 6, we'll continue exploring modern SQL capabilities by learning to work with semi-structured and text data. If you'd like to go beyond these topics, be sure to visit Markus Winand's Modern SQL website (https://modern-sql.com/).

3.2 Loading the music service dataset

We'll explore the modern SQL capabilities by working through an example of a music streaming service where users can listen to their favorite songs and artists. The database schema is minimalistic and looks like figure 3.1.

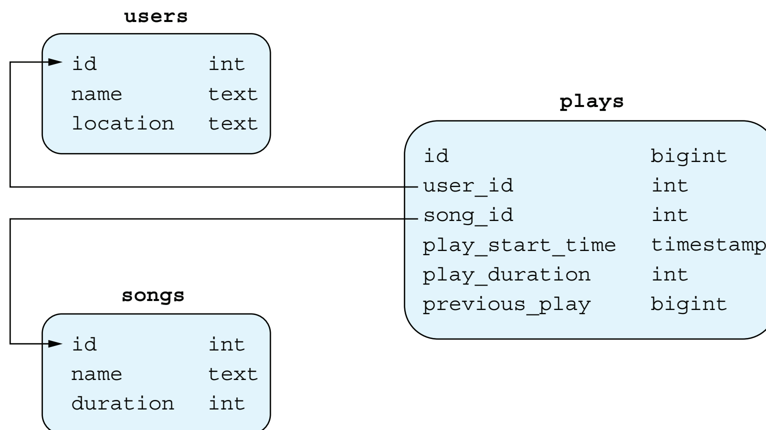


Figure 3.1 A simple database schema for a music streaming service

The `songs` and `users` tables are self-explanatory, storing information about the users subscribed to the streaming service and the songs available for listening. The `plays` table tracks the songs users listen to. When a user plays a new song, a new record is added to the `plays` table with the following details:

- The `id` column is a unique identifier for the current play session.
- The `user_id` and `song_id` columns reference the respective user and song records.
- The `play_start_time` column stores the timestamp of the time when the user started listening to the song.
- The `play_duration` column tracks how long the song was played by the user.
- The `previous_play` column is either set to `NULL` or holds the `id` of the previous play session. This helps track a sequence of songs played one after another without interruption.

NOTE If you'd like to gain practical experience while reading the chapter, connect to the Postgres instance you started in chapter 1 using the `docker exec -it postgres psql -U postgres` command.

Follow these steps to preload the dataset into the Postgres instance in Docker:

1. Clone the book's repository with listings and sample data:

```
git clone https://github.com/dmagda/just-use-postgres-book
```

2. Copy the music streaming dataset to your Postgres container:

```
cd just-use-postgres-book/
docker cp data/streaming/. postgres:/home/.
```

3. Preload the dataset by connecting to the container and using the `\i` meta-command of psql to execute the copied SQL scripts:

```
docker exec -it postgres psql -U postgres -c "\i /home/streaming_ddl.sql"
docker exec -it postgres psql -U postgres -c "\i /home/streaming_data.sql"
```

4. Once the dataset is preloaded, connect to Postgres and check the created tables with the `\Dt Streaming.*` command:

```
docker exec -it postgres psql -U postgres
\dt streaming.*
```

The output will be as follows:

```
          List of relations
  Schema   | Name  | Type  |  Owner
-----------+-------+-------+----------
 streaming | plays | table | postgres
 streaming | songs | table | postgres
 streaming | users | table | postgres
(3 rows)
```

With the dataset in place, let's move forward and begin our journey into modern SQL with CTEs.

3.3 Common table expressions

CTEs allow us to break down large or complex queries into smaller, more manageable pieces. They are defined using the `WITH` clause in the following format:

```
WITH cte_name AS (
    auxiliary_statement
)
primary_statement
```

The WITH clause defines a named CTE that can be referenced later in the primary statement. The CTE evaluates the auxiliary statement, which is a regular SQL query that either SELECTs existing data or modifies it with INSERT, UPDATE, DELETE, or MERGE statements. The result of the execution is available to the primary statement that is executed next.

The primary statement can be a SELECT query or a data modification command such as INSERT or UPDATE that uses the data produced by the CTE. Overall, you can think of a CTE as a temporary table or view generated and used solely by the given query. Let's return to our imaginary music streaming service and learn more about CTEs by seeing them in action.

3.3.1 Selecting data with CTEs

Suppose we need to track the most popular songs played on the service within a specific time range. The following listing shows how such a report can be generated using a CTE.

Listing 3.1 Tracking the most popular songs

```
WITH plays_cte AS (
    SELECT s.title, s.duration
    FROM streaming.plays p
    JOIN streaming.songs s ON p.song_id = s.id
    WHERE p.play_start_time::DATE BETWEEN '2024-09-15' AND '2024-09-16'
      AND p.play_duration = s.duration
)
SELECT title, COUNT(*) AS play_count
FROM plays_cte
GROUP BY title
ORDER BY play_count DESC;
```

The query defines the plays_cte CTE that executes the auxiliary statement returning all the songs played by users to the end (p.play_duration = s.duration) between 2024-09-15 and 2024-09-16. The songs table stores information about the compositions, and the plays table tracks how long the songs were played by the users. The primary statement calculates the rank by using the result produced by the CTE (SELECT … FROM plays_cte) and counting the total number of times each song was played by users (COUNT(*) AS play_count).

The query returns the rank of the most popular songs as follows:

```
 title  | play_count
--------+------------
 Song D |          4
 Song A |          3
 Song G |          2
(3 rows)
```

NOTE Depending on the complexity of the query, Postgres may decide to fold a CTE into the primary statement or transform the original query into a more efficient form. For instance, the query in listing 3.1 uses a CTE for educational

purposes but can be rewritten as a single `SELECT` statement. Postgres recognizes this and will rewrite or optimize the query accordingly. You can verify this by checking the execution plan with the `EXPLAIN ANALYZE` statement.

Just as you can easily read and modify clearly written application logic, queries using CTEs can also be adjusted seamlessly. Suppose the music streaming service needs to track the least popular songs within a specific time range. This ranking can be helpful when algorithms need to decide what to add or remove from users' recommendations. The next query shows how the rank can be calculated with a few adjustments to the previous query.

Listing 3.2 Finding the least popular songs

```
WITH plays_cte AS (
    SELECT s.title, s.duration, p.play_duration
    FROM streaming.plays p
    JOIN streaming.songs s ON p.song_id = s.id
    WHERE p.play_start_time::DATE BETWEEN '2024-09-15' AND '2024-09-16'
      AND p.play_duration < (s.duration / 2)
)
SELECT title, MIN(play_duration) AS min_play_duration
FROM plays_cte
GROUP BY title
ORDER BY min_play_duration ASC LIMIT 3;
```

As before, the `plays_cte` CTE populates a list of songs with their played duration between `2024-09-15` and `2024-09-16`. However, this time the CTE returns songs that were listened to for less than half of their original duration (`p.play_duration < (s.duration / 2)`). These songs qualify as less popular on our streaming service. The primary statement queries `plays_cte` to find the three songs that were played the least (`MIN(play_duration)`).

The least popular songs are as follows, with `min_play_duration` measured in seconds:

```
 title  | min_play_duration
--------+-------------------
 Song E |                32
 Song F |                78
 Song J |                91
(3 rows)
```

3.3.2 Using multiple CTEs in a query

So far, we've looked at queries with a single CTE. However, a single query can define multiple CTEs, where each subsequent CTE can reference the ones defined before it (except for the first CTE, which has no preceding CTEs). Suppose we find the query in listing 3.2 a bit unfair for just-released songs, and we want to modify it so that a song is added to the least-popular rank only if three or more users listened to it and didn't find it interesting. This can be achieved by using the following version of the query.

Listing 3.3 Finding least-popular songs across three or more users

```
WITH plays_cte AS (
    SELECT s.title, s.duration, p.play_duration, p.user_id
    FROM streaming.plays p
    JOIN streaming.songs s ON p.song_id = s.id
    WHERE p.play_start_time::DATE BETWEEN '2024-09-15' AND '2024-09-16'
      AND p.play_duration < (s.duration / 2)
),
user_play_counts AS (
    SELECT title, duration, COUNT(DISTINCT user_id) AS user_count,
      MIN(play_duration) AS min_play_duration,
      COUNT(*) AS total_play_count
    FROM plays_cte
    GROUP BY title, duration
)
SELECT title, duration, min_play_duration, total_play_count
FROM user_play_counts
WHERE user_count >= 3
ORDER BY min_play_duration ASC
LIMIT 3;
```

The `plays_cte` CTE remains unchanged and looks exactly the same as in listing 3.2. It generates a list of songs that were played for less than half of their duration. The `user_play_counts` CTE queries the `plays_cte` CTE and counts how many times each song was played by unique users. The primary statement ranks the least popular songs by querying the results of the `user_play_counts` CTE and returns only those songs that were listened to by at least three users (`WHERE user_count >= 3`).

The output of the query is as follows:

```
 title  | duration | min_play_duration | total_play_count
--------+----------+-------------------+------------------
 Song E |      150 |                32 |                3
(1 row)
```

This query clearly achieves our goal, and its logic is easy to read and follow. However, based on this query, Postgres needs to execute three separate `SELECT` statements to produce the final result. Wouldn't it be more efficient and performant to use nested queries (subqueries) or consolidate them into a single `SELECT` statement? The answer is: it depends. We can't know for certain until we compare the execution plans. But one thing you can be confident about, though, is that Postgres doesn't always execute queries the way they're written. A SQL query is simply a request to do some action or produce a result, and it's up to the database to decide how to execute that request.

If we want to understand how a query is actually executed by Postgres, we can look at the query execution plan using the `EXPLAIN` statement. We'll cover `EXPLAIN` in more detail in chapter 4, but, in the meantime, let's add the `EXPLAIN(analyze, costs off, timing off)` command to the beginning of the query from listing 3.3 and execute it again. Once we do this, Postgres will produce the following execution plan:

```
                                QUERY PLAN
------------------------------------------------------------------------
 Limit (actual rows=1 loops=1)
   ->  Sort (actual rows=1 loops=1)
         Sort Key: user_play_counts.min_play_duration
         Sort Method: quicksort  Memory: 25kB
         ->  Subquery Scan on user_play_counts (actual rows=1 loops=1)
               ->  GroupAggregate (actual rows=1 loops=1)
                     Group Key: s.title, s.duration
                     Filter: (count(DISTINCT p.user_id) >= 3)
                     Rows Removed by Filter: 5
                     ->  Sort (actual rows=9 loops=1)
                           Sort Key: s.title, s.duration, p.user_id
                           Sort Method: quicksort  Memory: 25kB
                           ->  Hash Join (actual rows=9 loops=1)
                                 Hash Cond: (s.id = p.song_id)
                                 Join Filter: (p.play_duration
                                 ➥   < (s.duration / 2))
                                 Rows Removed by Join Filter: 27
                                 ->  Seq Scan on songs s
                                     ➥ (actual rows=10 loops=1)
                                 ->  Hash (actual rows=36 loops=1)
                                       Buckets: 1024  Batches: 1
                                       ➥   Memory Usage: 10kB
                                       ->  Seq Scan on plays p
                                           ➥ (actual rows=36 loops=1)
                                             Filter: ((
      ➥ (play_start_time)::date >= '2024-09-15'::date) AND
      ➥ ((play_start_time)::date <= '2024-09-16'::date))
```

Even though we define the `plays_cte` CTE at the very beginning of our SQL query, it doesn't appear in the execution plan. Instead, Postgres folds `plays_cte` into the `user_play_counts` and evaluates the latter:

- `Hash Join` phase—The database joins the `plays` and `songs` tables, returning only the rows that satisfy the search condition.
- `GroupAggregate` phase—Because there are no secondary indexes, Postgres sorts the result of the `Hash Join` phase and then groups the rows according to the condition from the primary statement that at least three users must have listened to the song (`COUNT(DISTINCT p.user_id) >= 3`).
- `Subquery Scan on user_play_counts` phase—Postgres materializes the `user_play_counts` CTE and iterates through the final result, calculating the rank of the songs.

Thus, even if the CTE version of a query appears longer and seems more complex to execute, that doesn't necessarily mean it will be less efficient. In the case of the query from listing 3.3, Postgres traverses the plays and songs tables once, filtering, sorting, and grouping the data based on the conditions defined in the two CTEs and the primary statement.

So, when deciding between using nested queries or CTEs for a particular task, choose the option that makes the flow of your SQL query clearer and more readable for you and your team. In most cases, Postgres can efficiently fold CTEs into each other and the primary statement, allowing you to benefit from the improved readability of CTEs without compromising performance.

NOTE If a CTE is referenced more than once from the primary statement or other subsequent CTEs, Postgres will materialize it by evaluating it once and storing the result for future references. You can also enforce materialization by explicitly adding `MATERIALIZED` to the CTE definition as follows: `WITH cte_name AS MATERIALIZED`. On the other hand, using `NON MATERIALIZED` has the opposite effect, instructing Postgres to reevaluate the CTE every time it is referenced, regardless of how many times it's called.

3.3.3 Modifying data with CTEs

In addition to querying data with CTEs, it's also possible to create data-modifying CTEs. These CTEs use `INSERT`, `UPDATE`, `DELETE`, or `MERGE` statements within the `WITH` clause, making the new/modified/deleted records available to the primary statement. For example, let's assume that our music streaming service uses the following data-modifying CTE to update the play duration of a song while users are listening to it, and then checks if the song's rank has changed after each update.

Listing 3.4 Data-modifying CTE

```
WITH updated_play AS (
    UPDATE streaming.plays
    SET play_duration = 200
    WHERE id = 30
    RETURNING song_id, play_duration
)
SELECT s.title, s.duration,
       CASE
           WHEN up.play_duration = s.duration THEN 'Moved Up the Rank'
           ELSE 'Rank Not Changed'
       END AS rank_change_status
FROM updated_play up
JOIN streaming.songs s ON s.id = up.song_id;
```

The query performs the following steps:

- The CTE updates the `plays` table by setting `play_duration` to 200 seconds for the play session with `id = 30`. If 200 seconds is the total duration of the song, it indicates that the user listened to the entire song, which may cause the song to move up in rank.
- The CTE uses the `RETURNING` clause to return `song_id` and its updated `play_duration` to the primary statement.

- The primary statement checks whether `play_duration` from the CTE matches the song's total duration. If so, it means that the song's rank has changed.

If you execute this data-modifying CTE, the result should be as follows:

```
 title  | duration | rank_change_status
--------+----------+-------------------
 Song A |      200 | Moved Up the Rank
(1 row)
```

TIP If you need to return all the columns of the change set, use the `RETURNING *` clause.

The `RETURNING` clause is optional and can be omitted in a data-modifying CTE if the primary statement doesn't need to process the result of the CTE's execution. Another interesting point about data-modifying CTEs is that they are executed even if the primary statement doesn't reference them. This behavior differs from CTEs with `SELECT` statements, which are evaluated only if explicitly referenced by the primary query.

As with CTEs that use `SELECT` statements, a single query can include multiple CTEs that either modify or query data. However, in this case, all CTEs are executed concurrently, and the changes made by data-modifying CTEs are not visible to other CTEs within the same query.

Let's see how concurrent execution works in practice by updating and querying the `play_duration` of the play session with `id = 12` from two different CTEs:

```
SELECT song_id, play_duration FROM streaming.plays WHERE id = 12;
```

Currently, this play session corresponds to the song with `song_id = 3`, and the user has already listened to 118 seconds of the song:

```
 song_id | play_duration
---------+---------------
       3 |           118
(1 row)
```

Let's use the following CTE to update the `play_duration` of that play session and check whether this change is visible to the subsequent CTE.

Listing 3.5 Querying and modifying CTEs executed concurrently

```
WITH updated_play AS (
    UPDATE streaming.plays
    SET play_duration = 150
    WHERE id = 12
    RETURNING song_id, play_duration
),
current_play_duration AS (
    SELECT song_id, (play_duration = 150) as is_change_visible_to_cte
```

```
    FROM streaming.plays
    WHERE id = 12
)
SELECT is_change_visible_to_cte,
    (play_duration = 150) as is_change_visible_to_primary
FROM updated_play up
JOIN current_play_duration cp ON up.song_id = cp.song_id;
```

The `updated_play` CTE changes `play_duration` for the session to 150 seconds. The `current_play_duration` CTE reads the current value of the same play session and sets the `is_change_visible_to_cte` column to `true` if `play_duration` is 150 seconds. The primary statement then queries the results of both CTEs, indicating whether the changes made by the data-modifying CTE were visible to the subsequent CTE and to the primary statement.

The output of the query looks as follows:

```
 is_change_visible_to_cte | is_change_visible_to_primary
--------------------------+------------------------------
 f                        | t
(1 row)
```

As we can see, the `is_change_visible_to_cte` column is set to `false`, indicating that the changes made by the `updated_play` CTE were not visible to the `current_play_duration` CTE. This occurs because both CTEs were executed concurrently on the same snapshot of data. If we want the `current_play_duration` CTE to run after `updated_play` and see its changes, we need to have `current_play_duration` reference `updated_play` directly, as shown next.

Listing 3.6 Querying and modifying CTEs executed sequentially

```
WITH updated_play AS (
    UPDATE streaming.plays
    SET play_duration = 160
    WHERE id = 12
    RETURNING id, song_id, play_duration
),
current_play_duration AS (
    SELECT song_id, (play_duration = 160) as is_change_visible_to_cte
    FROM updated_play
    WHERE id = 12
)
SELECT is_change_visible_to_cte,
    (play_duration = 160) as is_change_visible_to_primary
FROM updated_play up
JOIN current_play_duration cp ON up.song_id = cp.song_id;
```

This time, the `is_change_visible_to_cte` column is set to `true`, meaning the `current_play_duration` CTE was able to see the changes made by `updated_play` by querying it directly:

```
 is_change_visible_to_cte | is_change_visible_to_primary
--------------------------+-----------------------------
 t                        | t
(1 row)
```

In this section, we explored how CTEs can improve the readability and manageability of SQL queries. In the next section, we'll learn how CTEs enable recursive queries in Postgres for processing hierarchical or tree-like structures.

3.4 Recursive queries

A *recursive* query is a special type of CTE that is created by adding the `RECURSIVE` clause to the CTE definition. The general form of a recursive query is as follows:

```
WITH RECURSIVE cte_name                      <-- Recursive CTE definition
  (column1, column2, columnN)
AS(
    SELECT columns FROM table1               <-- The non-recursive term
    WHERE condition

    UNION [ALL]

    SELECT columns FROM cte_name             <-- Recursive term
    WHERE recursive_condition
)
SELECT columns FROM cte_name;                <-- Primary statement
```

- The `RECURSIVE` clause instructs Postgres that the CTE is a recursive query. The definition can include an optional list of columns (`column1, column2, columnN`) that are carried between recursion steps and are expected to be returned from both the non-recursive and recursive terms.
- The non-recursive term is executed once, populating a temporary working table with the initial data. This initial data is added to the query result and also used by the recursive term during the first recursion step.
- The `UNION` or `UNION ALL` clause merges the results of the non-recursive and recursive terms until the recursion stops. The difference between the two is that `UNION` discards duplicates from the result, and `UNION ALL` includes all rows and doesn't skip duplicates.

The recursive term continues executing recursively while the `recursive_condition` resolves to `true`. Each subsequent recursive step uses the result from the previous step, which is stored in the working table. The result of the current recursive step is deduplicated if the `UNION` clause is used. In the end, the result of the current recursive step, stored in the temporary intermediate table, is added to the final CTE result and replaces the contents of the working table. Then the recursion proceeds to the next step. Once the recursive CTE completes, the primary statement can query the result of the recursion.

As developers, we often understand concepts better when they are explained through code. The next listing uses pseudocode to demonstrate the execution flow of a recursive CTE and the role of the temporary working and intermediate tables.

Listing 3.7 Recursive query execution flow in pseudocode

```
// Step 1: Evaluate the non-recursive term, which is a SQL query defined
// at the beginning of the recursive CTE and executed only once.
non_recursive_result = execute(non_recursive_term);

// Step 2: Remove duplicates, if UNION is used
// Skip if the UNION ALL is used instead.
if (using UNION)
    non_recursive_result = remove_duplicates(non_recursive_result);

// Step 3: Add the non-recursive result to the final result set
final_result.add(non_recursive_result);

// Step 4: Initialize the working table with the non-recursive result
working_table = non_recursive_result;

// Step 5: Execute the recursive term while the working table is not empty
// The working table is not empty while the recursive
// condition resolves to true
while (working_table is not empty) {

    // Step 6: Evaluate the recursive term using
    // the current working table as input.
    // The recursive term is the SQL query defined in the recursive CTE
    // after the UNION clause.
    intermediate_table = execute(recursive_term, using=working_table);

    // Step 7: If UNION is used (not UNION ALL), discard duplicates from:
    //    1. The recursive result stored in the intermediate_table
    //    2. Any rows that already exist in the final result set
    if (using UNION)
        intermediate_table =
            remove_duplicates(intermediate_table, excluding=final_result);

    // Step 8: Add the recursive result from
    // the intermediate table to the final result
    final_result.add(intermediate_table);

    // Step 9: Replace the working table
    // with the contents of the intermediate table
    working_table = intermediate_table;
}

// Step 10: Return the final result
return final_result;
```

Now, let's put this knowledge into practice by exploring how recursive queries can be helpful for our music streaming service.

3.4.1 Querying hierarchical data

Our streaming service is capable of tracking songs played in sequence, which occurs when a user listens to songs one after another without interruption. The information about the played songs is stored in the `streaming.plays` table:

```
\d streaming.plays

                              Table "streaming.plays"
     Column      |            Type             | Collation | Nullable |
-----------------+-----------------------------+-----------+----------+
 id              | bigint                      |           | not null |
 user_id         | integer                     |           |          |
 song_id         | integer                     |           |          |
 play_start_time | timestamp without time zone |           |          |
 play_duration   | integer                     |           |          |
 played_after    | bigint                      |           |          |
```

Every time a user starts playing a new song, the streaming service adds a new record to the `streaming.plays` table, setting the `song_id` column to the current song's ID and the `id` column to the next available ID, which is incremented automatically. Additionally, if the song is played in sequence after another, the song's `played_after` column is set to the play session's `id` of the previous song. If the song is the first in the sequence, then its `played_after` column is set to `NULL`.

Song sequences are an example of a hierarchical structure that we can query and analyze using recursive queries. The sample dataset contains 13 songs that were the first in their sequence:

```
SELECT count(*) FROM streaming.plays WHERE played_after IS NULL;

 count
-------
    13
(1 row)
```

Let's create a recursive query that returns all the songs played in a sequence starting with a given song.

Listing 3.8 Finding songs played in sequence after a specific one

```
WITH RECURSIVE play_sequence AS (
    SELECT id, user_id, song_id,
      play_start_time, play_duration, played_after
    FROM streaming.plays
    WHERE id = 5

    UNION ALL

    SELECT p.id, p.user_id, p.song_id, p.play_start_time,
       p.play_duration, p.played_after
```

```
    FROM streaming.plays p
    JOIN play_sequence ps ON p.played_after = ps.id
)
SELECT user_id, song_id, play_start_time,
  play_duration as duration, played_after
FROM play_sequence
ORDER BY play_start_time;
```

The non-recursive term retrieves information about the play session with `id = 5` and adds its song to the final result. The recursive term then looks for the next song played after the one returned by the non-recursive term (`p.played_after = ps.id`). If such a song exists, the recursion adds it to the final result and continues down the sequence until it reaches the last song. The last song is the one whose `plays.id` value is not referenced in the `plays.played_after` column of any other song.

Once the recursion stops, the primary statement outputs the sequence, ordering the records by the `play_start_time` column. The song sequence, starting with `plays.id = 5`, looks as follows:

```
 user_id | song_id |   play_start_time    | duration | played_after
---------+---------+---------------------+----------+--------------
       1 |       5 | 2024-09-15 01:34:00 |       61 |
       1 |       6 | 2024-09-15 01:37:30 |      139 |            5
       1 |       7 | 2024-09-15 01:42:00 |      104 |            6
       1 |       8 | 2024-09-15 01:45:00 |       99 |            7
(4 rows)
```

Additionally, if we modify the previous query slightly, we can retrieve the hierarchy level for each song in the sequence. This can be done by adding a custom column named `level`, which is set to `1` in the non-recursive term and incremented by 1 at each step of the recursion. Here's how the updated query looks:

```
WITH RECURSIVE play_sequence AS (
    SELECT id, user_id, song_id, play_start_time,
    play_duration, played_after, 1 as level
    FROM streaming.plays
    WHERE id = 5

    UNION ALL

    SELECT p.id, p.user_id, p.song_id, p.play_start_time,
    p.play_duration, p.played_after, level + 1
    FROM streaming.plays p
    JOIN play_sequence ps ON p.played_after = ps.id
)
SELECT user_id, song_id, play_start_time,
  play_duration as duration, played_after, level
FROM play_sequence
ORDER BY play_start_time;
```

The query returns the same data as the original query from listing 3.8, with the added hierarchy level for each played song:

```
 user_id | song_id |   play_start_time   | duration | played_after | level
---------+---------+---------------------+----------+--------------+-------
       1 |       5 | 2024-09-15 01:34:00 |       61 |              |     1
       1 |       6 | 2024-09-15 01:37:30 |      139 |            5 |     2
       1 |       7 | 2024-09-15 01:42:00 |      104 |            6 |     3
       1 |       8 | 2024-09-15 01:45:00 |       99 |            7 |     4
(4 rows)
```

After learning how to use recursive queries in practice, let's see how to use additional arguments that can be passed into the recursion.

3.4.2 Using arguments in recursion

Recursive queries allow us to define a list of columns that are carried through recursion steps and are expected to be returned to the primary statement once the recursion stops. These columns are listed in the definition of the recursive CTE as follows: `WITH RECURSIVE cte_name (column1, column2, columnN)`. They can be used as aliases for existing table columns or store values that are generated, accumulated, or updated as the recursion progresses.

Suppose our streaming service needs to track the total play duration of a specific sequence. We can calculate the total duration by defining a `total_duration` column as a CTE argument, passing it through each recursion step, where it will be incremented by the play duration of every song in the sequence. The next listing shows how the total duration can be calculated.

Listing 3.9 Calculating the total play duration for a sequence

```
WITH RECURSIVE play_sequence(parent_id, sequence, total_duration) AS (
    SELECT id, ARRAY[id], play_duration
    FROM streaming.plays
    WHERE id = 5

    UNION ALL

    SELECT p.id, ps.sequence || p.id, total_duration + p.play_duration
    FROM streaming.plays p
    JOIN play_sequence ps ON p.played_after = ps.parent_id
)
SELECT p.song_id, p.play_start_time, p.play_duration,
  ps.sequence, ps.total_duration
FROM play_sequence ps
JOIN streaming.plays p ON ps.parent_id = p.id
ORDER BY ps.sequence, p.play_start_time;
```

The recursive query defines three columns that are carried through the recursion steps: `parent_id`, `sequence`, and `total_duration`. The non-recursive term runs first, returning the value of the `id` column as `parent_id`, initializing the path column with `ARRAY[id]`, and setting `total_duration` to the value of `play_duration`. The values are assigned in the order in which the arguments are listed in the CTE definition.

The recursive term then increments `total_duration` with the current song's `play_duration` and appends the current song's `id` to the sequence as follows: `ps.sequence || p.id`. Once the recursion stops, the primary statement can access the data stored in the columns defined as arguments of the recursive query.

The query produces the following result, with the total duration of the entire sequence shown in the last row (403 seconds):

```
 song_id |   play_start_time   | play_duration | sequence  | total_duration
---------+---------------------+---------------+-----------+---------------
       5 | 2024-09-15 01:34:00 |            61 | {5}       |             61
       6 | 2024-09-15 01:37:30 |           139 | {5,6}     |            200
       7 | 2024-09-15 01:42:00 |           104 | {5,6,7}   |            304
       8 | 2024-09-15 01:45:00 |            99 | {5,6,7,8} |            403
(4 rows)
```

Finally, all our previous examples demonstrated how to use recursive queries to build the song sequence for the play session starting with `id = 5`. If you'd like to retrieve all existing song sequences, simply change the condition from `WHERE id = 5` to `WHERE played_after IS NULL` in the non-recursive term from listing 3.9. Here's how the updated query looks:

```
WITH RECURSIVE play_sequence(parent_id, sequence, total_duration) AS (
    SELECT id, ARRAY[id], play_duration
    FROM streaming.plays
    WHERE played_after is NULL

    UNION ALL

    SELECT p.id, ps.sequence || p.id, total_duration + p.play_duration
    FROM streaming.plays p
    JOIN play_sequence ps ON p.played_after = ps.parent_id
)
SELECT p.song_id, p.play_start_time, p.play_duration,
  ps.sequence, ps.total_duration
FROM play_sequence ps
JOIN streaming.plays p ON ps.parent_id = p.id
ORDER BY ps.sequence, p.play_start_time;
```

The query returns a long list of song sequences along with their total durations. This truncated output shows the first two sequences:

```
 song_id |   play_start_time   | play_duration |  sequence  | total_duration
---------+---------------------+---------------+------------+---------------
       1 | 2024-09-15 12:14:00 |           200 | {1}        |            200
       2 | 2024-09-15 12:17:00 |           144 | {1,2}      |            344
       3 | 2024-09-15 12:21:00 |           110 | {1,2,3}    |            454
       4 | 2024-09-15 12:25:30 |           300 | {1,2,3,4}  |            754
       5 | 2024-09-15 01:34:00 |            61 | {5}        |             61
       6 | 2024-09-15 01:37:30 |           139 | {5,6}      |            200
       7 | 2024-09-15 01:42:00 |           104 | {5,6,7}    |            304
       8 | 2024-09-15 01:45:00 |            99 | {5,6,7,8}  |            403
(truncated output)
```

With this, we've learned how to use the modern SQL capabilities of Postgres to take advantage of recursive queries for processing hierarchical and tree-like structures.

3.5 Window functions

Like recursive queries, window functions are among the most effective modern SQL capabilities. However, although the applicability of recursive queries is primarily limited to hierarchical and tree-like structures, window functions have much broader applications. They are widely used for ranking and sorting, calculating running totals, cumulative sums, moving averages, lag and lead analysis, and much more.

A window function allows us to perform calculations over a set of rows that are related to each other. Queries using window functions divide data into subsets (windows) with the `OVER` clause and then apply a function to each window. The most basic and minimal form of a window function looks as follows:

```
SELECT function_name(arguments)
OVER (PARTITION BY columnA)
FROM table;
```

`function_name(arguments)` is the window function applied to specific columns and arguments. This can be an aggregate function such as `SUM()` or `AVG()`, or a built-in window function like `RANK()` or `ROW_NUMBER()`. The `OVER` clause partitions the data into subsets (windows), placing rows with the same `columnA` value into the same set.

Overall, the definition and applicability of window functions have some correlation with regular non-window aggregates, which also perform calculations over groups of data. However, regular non-window aggregates, which are always accompanied by the `GROUP BY` clause, return a single row for each group, whereas window functions retain every row of a set in the output. Let's explore what this difference means in practice by experimenting with our music streaming service.

Imagine that we need to calculate the total play duration for every song. This task can be easily accomplished by using the `SUM()` aggregate over grouped data.

Listing 3.10 Calculating the total play duration for every song

```
SELECT song_id, SUM(play_duration) as total_duration
FROM streaming.plays
GROUP BY song_id ORDER BY total_duration DESC;
```

The query returns the following result, listing the songs with the longest total play duration first:

```
 song_id | total_duration
---------+----------------
       4 |           1200
       1 |            800
       7 |            798
       2 |            654
```

```
       3 |            467
       9 |            463
       6 |            451
       8 |            430
      10 |            389
       5 |            268
(10 rows)
```

But what if our streaming service needs to not just calculate the total play duration of a song but also include in the output every user who listened to the song? Essentially, we need an individual row for each user who listened to a song while still displaying the total play duration of the song across all users. This task can no longer be accomplished with a regular non-window `SUM()`, as we would need to add the `user_id` column to the `GROUP BY` clause as follows:

```
SELECT song_id, user_id, SUM(play_duration) as total_duration
FROM streaming.plays
GROUP BY (song_id, user_id) ORDER BY song_id;
```

And if we do that, the `SUM()` function will calculate the total play duration per user, not across all users, which is not what we want to achieve:

```
 song_id | user_id | total_duration
---------+---------+----------------
       1 |       1 |            200
       1 |       2 |            200
       1 |       3 |            400
       2 |       3 |            304
       2 |       1 |            144
       2 |       2 |            206
(truncated output)
```

To solve this task without window functions, we would need to use a self-join.

Listing 3.11 Calculating total play duration with users using a self-join

```
SELECT DISTINCT p.song_id,
       p.user_id,
       t.total_duration
FROM streaming.plays p
JOIN (
    SELECT song_id,
           SUM(play_duration) AS total_duration
    FROM streaming.plays
    GROUP BY song_id
) t ON p.song_id = t.song_id
ORDER BY p.song_id;
```

The query does its job by calculating the total play duration for each song and listing all the users who listened to that particular song:

```
 song_id | user_id | total_duration
---------+---------+---------------
       1 |       1 |            800
       1 |       2 |            800
       1 |       3 |            800
       2 |       1 |            654
       2 |       2 |            654
       2 |       3 |            654
(truncated output)
```

But although the self-join approach works as expected, it's not the most efficient, because every row of the table is accessed twice. Additionally, it's not the easiest to follow when trying to understand the query logic.

In contrast, with window functions, the same task can be solved in a more concise, efficient, and readable manner. The following listing shows how to perform the same calculation using the window version of the `SUM()` aggregate.

Listing 3.12 Calculating total play duration with users using window functions

```
WITH plays_with_total AS (
  SELECT
    song_id, user_id, SUM(play_duration)
    OVER (PARTITION BY song_id) AS total_duration
  FROM streaming.plays
)
SELECT DISTINCT song_id, user_id, total_duration
FROM plays_with_total
ORDER BY song_id, user_id;
```

First the query executes the `plays_with_total` CTE that defines the window function. That window function iterates through the table by placing each record into a subset (window) based on the `song_id` value and calculating the total play duration for each window separately. Figure 3.2 provides a visual representation of this process.

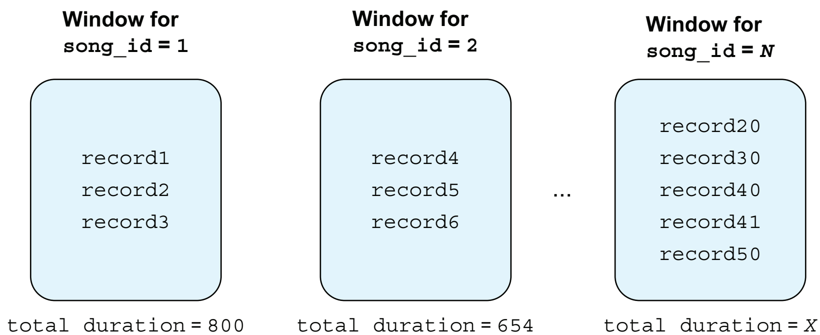


Figure 3.2 Records are grouped in windows by song ID.

After the CTE is evaluated, the query executes the `SELECT DISTINCT` statement over the window function's result, removing any duplicate `(song_id, user_id)` rows with the same `total_duration`. In the end, the query produces a result similar to the one we saw in the version of the query using the self-join:

```
 song_id | user_id | total_duration
---------+---------+----------------
       1 |       1 |            800
       1 |       2 |            800
       1 |       3 |            800
       2 |       1 |            654
       2 |       2 |            654
       2 |       3 |            654

(truncated output)
```

The records grouped into windows by the `PARTITION BY` clause can be further divided into frames by adding `ORDER BY` to the `OVER` clause. Sorting records within each window lets us calculate running totals from the first to the last window frame.

For example, the query in listing 3.13 calculates the total play duration for the song with `song_id = 2`. The query also orders the records within the window by the `user_id` value. As a result, the window function evaluates over a series of frames—each frame includes the current user's row plus rows from previous users, if any.

Listing 3.13 Calculating running totals and total duration

```
SELECT song_id, user_id, play_duration, SUM(play_duration)
OVER (PARTITION BY song_id ORDER BY user_id) AS total_play_duration
FROM streaming.plays
WHERE song_id = 2;
```

The query produces the following result, calculating the running total for the song:

```
 song_id | user_id | play_duration | total_play_duration
---------+---------+---------------+---------------------
       2 |       1 |           144 |                 144
       2 |       2 |           206 |                 350
       2 |       3 |           186 |                 654
       2 |       3 |           118 |                 654
```

The calculation starts with the first window frame, which holds the data for the user with `user_id = 1`. Because this frame has only one record, the total play duration equals the time that the user listened to the song (144 seconds).

The second frame includes the rows from the previous frame for `user_id = 1` plus the rows for the user with `user_id = 2`. The second user listened to the song for 206 seconds, which is added to the total for the first user (144 seconds), giving us a running total of 350 seconds.

The window function processes the last frame, which holds the rows from two preceding frames plus the rows for the user with `user_id = 3`. This user listened to the song twice, for 186 and 118 seconds, totaling 304 seconds. This value is summed with the totals for the first and second users, resulting in 654 seconds. This final total represents the total play duration of the song across all users. Figure 3.3 illustrates this process.

NOTE The `PARTITION BY` clause is optional in listing 3.13 and can be omitted as long as the `WHERE` clause ensures that only the records with `song_id = 2` are passed to the window function for processing. As a result, with the current `WHERE` condition, there will always be a single window containing multiple window frames.

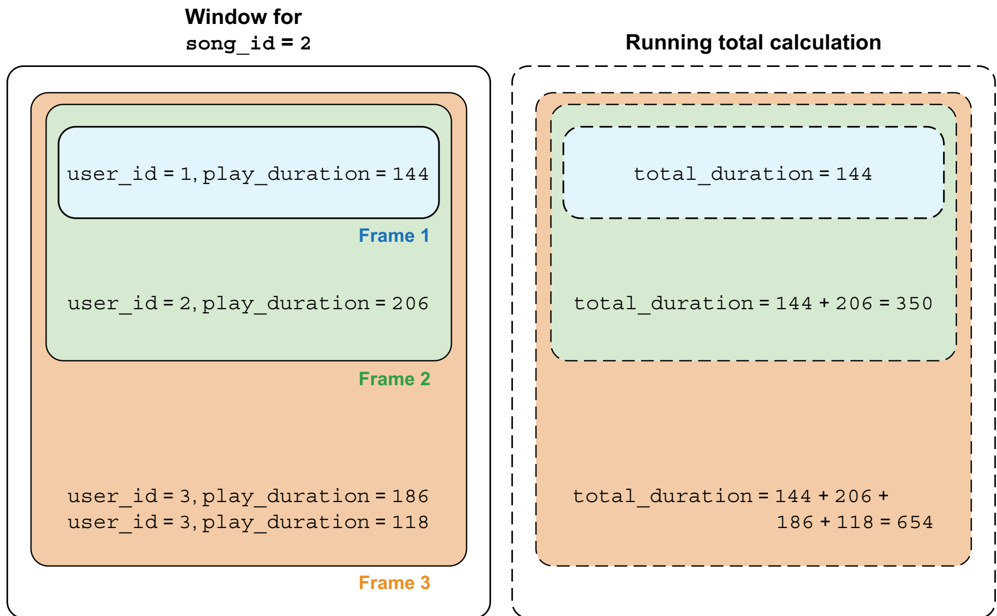


Figure 3.3 Window frames with a running total calculation

So far, we've seen how to use the `SUM()` aggregate as a window function. In general, any built-in aggregate function, such as `AVG()`, `MIN()`, or `COUNT()`, can be used as a window function. In addition, Postgres provides specialized built-in functions that can only be used in window mode, for example, with the `OVER` clause. These window functions allow us to calculate ranks or perform lag-lead analysis by comparing different sets of data.

Suppose our music streaming service needs to rank songs by their total play duration. The next listing shows how to accomplish this task with the `RANK()` window function.

Listing 3.14 Ranking songs by total play duration

```
SELECT song_id, SUM(play_duration) AS total_play_duration,
RANK() OVER (ORDER BY SUM(play_duration) DESC) AS song_rank
FROM streaming.plays
GROUP BY song_id
ORDER BY song_rank;
```

The query executes as follows:

- `GROUP BY song_id`—Postgres traverses the `streaming.plays` table, grouping the records by `song_id` and calculating the total play duration for each song as `SUM(play_duration)`. As a result, each song will have a single record with its calculated total duration.
- `RANK() OVER (ORDER BY SUM(play_duration) DESC)`—The grouped data is fed into the window function, which orders and ranks the songs based on `SUM(play_duration)` in a descending manner. If two songs have the same total play duration, they will belong to the same window frame, and the rank function will assign them the same rank.
- `ORDER BY song_rank`—The result is ordered by the calculated `song_rank` in ascending order. The song with the longest total play duration will be assigned rank 1.

The query outputs the following result:

```
 song_id | total_play_duration | song_rank
---------+---------------------+-----------
       4 |                1200 |         1
       1 |                 800 |         2
       7 |                 798 |         3
       2 |                 654 |         4
       3 |                 467 |         5
       9 |                 463 |         6
       6 |                 451 |         7
       8 |                 430 |         8
      10 |                 389 |         9
       5 |                 268 |        10
(10 rows)
```

NOTE In listing 3.14, the `PARTITION BY` clause in the `OVER` statement is omitted. As a result, the window function operates on a single window of data. The `song_rank` differs for each row in listing 3.14 because the records in this single window are further divided into window frames by the `ORDER BY SUM(play_duration)` clause. The rank is then calculated similarly to how a running total is computed: the first frame with the longest play duration is assigned rank 1, and the rank of each subsequent frame is calculated as `1 + rank_of_the_previous_window_frame`.

With this, we've learned how to use window functions over subsets of data that are related to each other.

Summary

- Even though it was invented over 50 years ago, like most languages humans speak, SQL continues to evolve to meet the needs of the modern world.
- Modern SQL is a category of features and capabilities that were introduced after the SQL-92 standard and allow us to use Postgres beyond the classic relational model and its traditional use cases.
- Common table expressions (CTEs) allow us to break down large or complex queries into pieces that are smaller and more readable and manageable.
- Recursive queries let us process hierarchical and tree-like structures in SQL.
- Window functions allow us to divide data into groups (windows) and perform aggregations, ranking, lag-lead analysis, and other calculations within each window.

4 Indexes

This chapter covers

- Exploring index types
- Understanding query execution plans
- Optimizing query performance
- Reducing the number of table lookups
- Indexing a subset of data
- Indexing the result of a function or expression

Indexes are often the first optimization technique that comes to mind when dealing with a long-running query or a slow database operation. They've proven so effective in many scenarios that we sometimes overlook other optimization methods, turning to indexes right away. As a result, indexes may be the second most common database object we create and use, right after tables.

Let's explore Postgres's indexing capabilities as we build a multiplayer online game played by people around the world. We'll learn how and when to use various index types to keep the gamers' experience smooth and uninterrupted even during peak hours.

4.1 Why are indexes so popular?

Suppose we have a table with 100 records, and we ask Postgres to return a record with `ID` equal to `5`. The algorithmic complexity of this search operation, expressed in Big O notation, is $O(N)$, where N is the number of records in the table. $O(N)$ represents linear search complexity, meaning that Postgres needs to perform a sequential table scan, potentially visiting all N records to find the requested one or confirm that it doesn't exist. And the larger the N, the longer it takes for Postgres to find the requested record.

With just 100 records, Postgres would need a maximum of 100 lookups to find the record with an `ID` value of `5`. But what if the table grows to thousands or even millions of records? In that case, the database might need to scan through thousands or millions of entries before finding the one holding `ID = 5`.

Now, let's see how a database index can help expedite a search. An *index* is a data structure that provides faster access to data by storing indexed values in sorted order. Postgres supports many index types, with B-tree being the default general-purpose index type.

Imagine there's a B-tree index on the `ID` column, and we're searching for a record with an `ID` value of `5`. The B-tree stores the `ID` values in sorted order and looks like figure 4.1.

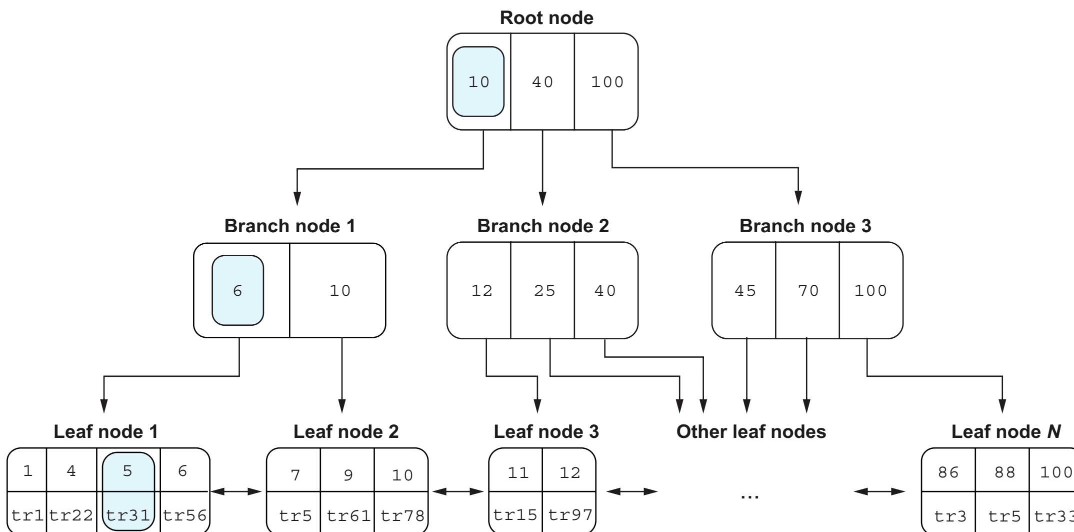


Figure 4.1 A sample B-tree index showing the search path to the index entry with `ID = 5`

The search through the index has an algorithmic complexity of $O(\log_b N)$, where N is the total number of index entries and b is the branching factor, representing the average number of entries per branch node. A root node points to branch nodes, which in turn point to leaf nodes containing references to the actual table records. When a

node at a higher level points to a node at a lower level, it means the lower-level node's index entries are less than or equal to the entry the pointer originates from. For example, as figure 4.1 shows, the root node has index entry 10, which points to branch node 1 storing entries 6 and 10. Index entry 6 in that branch node points to leaf node 1, which stores entries 1, 4, 5, and 6.

The leaf nodes are linked together, forming an ordered linked list. As shown in figure 4.1, all index entries in leaf node 1 are smaller than those in leaf node 2, and both nodes are linked together. This structure allows the leaf nodes to be traversed in order without navigating back up the tree, which helps optimize range queries and ordered data retrieval.

With this data arrangement, the number of index lookups required to reach the leaf node holding the reference to the record with `ID = 5` is proportional to $O(\log_b N)$, where an index lookup is defined as an operation that visits a specific index node.

Assuming the sample index in figure 4.1 stores 100 index entries (N) and has a branching factor of 3 (b), it will take up to $O(\log_3 100) = 4$ lookups to reach the leaf node holding a table reference for `ID = 5`. Figure 4.1 illustrates the search path through the index. In this example, it takes three index lookups to reach the correct node: the root node (first lookup), branch node 1 (second lookup), and leaf node 1 (third lookup). Within the leaf node, we retrieve the index entry for `ID = 5`, which stores the table reference (`tr`) to the actual record in the table—represented as `tr31`, essentially an address.

Now, let's briefly review how search performance changes as the table volume grows. Suppose we have a different B-tree index with a branching factor of 10, but still with 100 records. With this new branching factor, the database needs at most two lookups ($\log_{10} 100 = 2$) to locate the leaf node containing the reference to the record with `ID = 5`. Even if the data volume increases to thousands or millions of records, as shown in table 4.1, the index still requires only a small number of lookups to locate the record.

Table 4.1 Algorithmic complexity of $O(\log_{10} N)$

Table size	Number of index lookups
100 records	2
1,000 records	3
1,000,000 records	6
10,000,000 records	7
1,000,000,000 records	9

As a result, it's no surprise that indexes are such a popular optimization technique. The $O(\log_b N)$ algorithmic complexity is so much faster than $O(N)$ that the search performance through the index barely changes even as the table grows from a handful of records to millions.

NOTE In real-world implementations, database indexes such as B-trees typically have a much larger branching factor, capable of storing hundreds or even thousands of entries per branch node. This means the database needs to traverse even fewer index nodes to find a specific record.

4.2 Overview of Postgres index types

As you dive deeper into the indexing capabilities of Postgres, you'll come across many terms, definitions, and capabilities without a clear categorization. There are single-column and composite indexes, partial and functional, B-trees and HNSW, and the list goes on. In this section, let's bring some order by introducing and discussing the two broad categories of index types:

- *By index scope or functional purpose (what we index)*—With indexes from this category, we decide which columns or table data to index and whether a function needs to be applied to the indexed data. Examples include single-column, composite, covering, partial, and functional/expression indexes.
- *By underlying data structure and access method (how we index)*—This category is defined by the underlying data structure, which determines how indexed data is stored and accessed internally by the database. Examples include B-tree, hash, bloom, GIN, RUM, GiST, SP-GiST, BRIN, HNSW, and IVFFlat indexes.

Each time we create an index in Postgres with the `CREATE INDEX` statement, the database needs to know which data to index and how to build and organize the index structure internally. It's our responsibility to choose an index type from the first category (what to index), although the selection of the underlying data structure (how to index) is optional—Postgres defaults to a B-tree index unless another data structure is specified.

For example, the following `CREATE INDEX` command instructs Postgres to create a single-column index on `columnA` of `tableA`. The `ON` clause defines what to index:

```
CREATE INDEX new_index_name ON tableA(columnA);
```

In this case, the statement doesn't specify how to index the data in `columnA`, so Postgres defaults to a B-tree index, which is a general-purpose index suitable for most use cases.

If we need to use a different index type from the "how we index" category instead of the default B-tree, we can add the `USING` clause to the `CREATE INDEX` statement:

```
CREATE INDEX new_index_name ON tableA USING GIN (columnA);
```

This statement specifies that the index on `columnA` should be built using the GIN (generalized inverted index) data structure instead of the default B-tree. GIN indexes are commonly used for columns storing arrays, JSON data, or text data used in full-text search queries.

In this chapter, we review all the index types from the "what we index" category, starting with single-column indexes and finishing with functional and expression ones.

Exploring index types

I discuss most of the index types from the "how we index" category throughout several chapters of the book:

- We continue to explore B-tree and hash indexes in greater detail in this chapter.
- Chapter 5 uses GIN indexes to optimize searches over JSON data.
- In chapter 6, we learn to use GIN and GiST indexes for full-text search queries.
- Chapter 8 covers HNSW and IVFFlat indexes, where we build a generative AI application with Postgres.
- In chapter 9, we examine a practical use case for BRIN indexes with time-series data.
- Chapter 10 shows how to optimize the search over the geospatial data with the GiST indexes.

Now that we understand why indexes are a popular optimization technique and what index types exist in Postgres, let's dive into how and when to use them in practice.

4.3 Loading the multiplayer game dataset

We'll begin exploring Postgres indexing capabilities using an example of a multiplayer game where people from around the world join to play together online. The sample dataset consists of a single table with the following structure, which is more than enough for discussing the indexes from the "what we index" category:

```
game.player_stats (
    player_id BIGINT PRIMARY KEY,
    username TEXT NOT NULL UNIQUE,
    level INTEGER NOT NULL CHECK (level BETWEEN 1 AND 10),
    score INTEGER DEFAULT 0,
    champion_title TEXT,
    join_date DATE NOT NULL,
    last_active TIMESTAMP,
    region TEXT CHECK (region IN ('NA', 'EMEA', 'APAC')),
    play_time INTERVAL DEFAULT '0 seconds',
    win_count INTEGER DEFAULT 0,
    loss_count INTEGER DEFAULT 0
);
```

The roles of the columns in the `player_stats` table are as follows:

- The `player_id` and `username` columns serve as the unique identifier and name associated with each player.
- The `level` column tracks the player's level as they progress through the game. A level of 1 is assigned to beginners who have just joined, whereas a level of 10 is for advanced players who have spent years on the multiplayer platform.
- The `score` column records the total score or points players have accumulated over their lifetime. It helps to rank players within the same level. For example, if two players are at level 10, their scores can be used to differentiate their rank.

- The `champion_title` column is set only for a limited number of players who have earned a transferable champion title.
- The `join_date` and `last_active` columns store the date the player joined the multiplayer game and the time of their last activity.
- The `region` specifies the geographic location of the player and helps in building regional player rankings.
- The `play_time`, `win_count`, and `loss_count` columns track the total time players have spent in the game, as well as their number of wins and losses.

NOTE If you'd like to gain practical experience while reading through the chapter, connect to your Postgres instance started in chapter 1 using the `docker exec -it postgres psql -U postgres` command.

Let's follow these steps to preload the dataset into the Postgres instance in Docker:

1. Clone the book's repository with listings and sample data:

```
git clone https://github.com/dmagda/just-use-postgres-book
```

2. Copy the gaming dataset to your Postgres container:

```
cd just-use-postgres-book/
docker cp data/gaming/. postgres:/home/.
```

3. Preload the dataset by connecting to the container and using the `\i` meta-command of psql to apply the copied SQL scripts:

```
docker exec -it postgres psql -U postgres -c "\i /home/gaming_ddl.sql"
docker exec -it postgres psql -U postgres -c "\i /home/gaming_data.sql"
```

The table will be created under the `game` schema, and you can confirm this by connecting to Postgres and executing the `\dt game.*` command:

```
docker exec -it postgres psql -U postgres
 \dt game.*
```

The output will be as follows:

```
            List of relations
 Schema |     Name     | Type  |  Owner
--------+--------------+-------+----------
 game   | player_stats | table | postgres
(1 row)
```

With the dataset in place, let's move forward and begin exploring Postgres's indexing capabilities. We'll start with the `EXPLAIN` statement, which helps us decide whether an index would be beneficial for a particular query.

4.4 Learning to use the EXPLAIN statement

If you have a slow query or long-running operation, the `EXPLAIN` statement in Postgres should be used first before deciding to create an index to optimize the query. The statement returns an execution plan generated by the Postgres planner for the provided query, helping you see how the query is executed now and whether an index is really necessary. Let's go through several examples to learn the basic capabilities of the `EXPLAIN` statement and see how it can be used to confirm whether indexes are already being utilized for a query.

Suppose the gaming platform uses the following query to calculate the total number of players with more than 100 wins:

```
SELECT count(*)
FROM game.player_stats
WHERE win_count > 100;
```

The query produces the following result:

```
 count
-------
  8061
(1 row)
```

If we want to know exactly how Postgres executes this query, we can use the `EXPLAIN` statement, which returns the execution plan used by the database. All we need to do is add `EXPLAIN` to the beginning of the query:

```
EXPLAIN
SELECT count(*)
FROM game.player_stats
WHERE win_count > 100;
```

The database uses the following execution plan:

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Aggregate  (cost=282.18..282.19 rows=1 width=8)
   ->  Seq Scan on player_stats  (cost=0.00..262.00 rows=8070 width=0)
         Filter: (win_count > 100)
(3 rows)
```

The `Seq Scan` step in the plan indicates that Postgres performs a full sequential scan—often referred to as simply as a table scan—on the `player_stats` table. The database visits every row, checking whether the `win_count` value is greater than 100 (`Filter: (win_count > 100)`). Postgres does the full table scan because there is no index on the `win_count` column yet.

Next, imagine that every player of our multiplayer game can view their current score, level, and other stats on a personal dashboard in the gaming UI (user interface). For

instance, to display the latest score and level for a player, the dashboard retrieves the data from the database using the following query:

```
SELECT username, level, score
FROM game.player_stats
WHERE player_id = 250;
```

The result for the player with an ID of 250 looks as follows:

```
 username | level | score
----------+-------+-------
 kelly20  |     9 |  8536
(1 row)
```

Suppose we monitor metrics for our multiplayer gaming platform, reported in real time, and notice that over the last 30 minutes, it has been taking much longer to load the personal dashboards for players. We, the creators of the gaming platform, decide to investigate the root cause of the increased latency, starting with the query that retrieves the player's score and level.

We use the EXPLAIN statement to inspect the execution plan generated by Postgres:

```
EXPLAIN
SELECT username, level, score
FROM game.player_stats
WHERE player_id = 250;
```

Postgres returns the following execution plan:

```
                          QUERY PLAN
--------------------------------------------------------------
 Index Scan using player_stats_pkey on player_stats
 ➥   (cost=0.29..8.30 rows=1 width=18)
   Index Cond: (player_id = 250)
(2 rows)
```

The `Index Scan` step in the plan indicates that the database already uses an index for query execution. The index being used is named `player_stats_pkey`, which is the primary key defined in the table structure (`player_id BIGINT PRIMARY KEY`). Postgres automatically creates an index for the primary key columns. The following query provides detailed information about the existing index.

Listing 4.1 Primary index details

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE schemaname = 'game'
  AND tablename = 'player_stats'
  AND indexdef LIKE '%player_id%';
```

The output of the query confirms that Postgres automatically created an index on the `player_id` column using a B-tree as the data structure:

```
     indexname     |                     indexdef
-------------------+------------------------------------------------
 player_stats_pkey | CREATE UNIQUE INDEX player_stats_pkey
                   |  ON game.player_stats USING btree (player_id)
(1 row)
```

The `EXPLAIN` statement shows only the execution plan that the database would use, but it does not actually run the query. To see both the plan and the actual execution time, we need to use `EXPLAIN ANALYZE`. Let's check whether the existing index is performing well for our multiplayer platform:

```
EXPLAIN ANALYZE
SELECT username, level, score
FROM game.player_stats
WHERE player_id = 250;
```

The database produces the following plan:

```
                         QUERY PLAN
-----------------------------------------------------------------
Index Scan using player_stats_pkey on player_stats
 ➥  (cost=0.29..8.30 rows=1 width=18)
 ➥  (actual time=0.046..0.049 rows=1 loops=1)
   Index Cond: (player_id = 250)
 Planning Time: 0.119 ms
 Execution Time: 0.077 ms
(4 rows)
```

According to the plan, Postgres used the `Index Scan` access method to find the record with `player_id = 250`. The `Planning Time` indicates how much time the query planner spent choosing the most efficient execution plan. The `Execution Time` shows how long it took to run the query using the selected plan. The total execution time for the query is the sum of `Planning Time` and `Execution Time`.

> **TIP** Execute a query with `EXPLAIN ANALYZE` a few times until Postgres produces consistent `Planning Time` and `Execution Time` values. The first executions might be slower because the database hasn't yet selected the most optimal execution plan.

Apart from the query execution time, the `EXPLAIN` statement supports additional options that let us view memory usage, the cost of serializing the query's output data, and more. The options can be specified in parentheses using the following format:

```
EXPLAIN ( option [, ...] ) statement
```

The statement can be `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or any other command for which Postgres can generate an execution plan. If `analyze` is added to the list of options, Postgres also executes the statement and returns additional execution metrics. For example, if you run an `EXPLAIN (analyze) UPDATE...` command, Postgres will not only return the execution plan for the `UPDATE` statement but also execute it and modify the data.

Suppose we want to check whether Postgres reads data from shared memory buffers or retrieves it from disk. This information is crucial because a query might run slowly not due to a suboptimal execution plan or missing index but because memory has become a limited resource, causing the database to access the disk more frequently during execution. We can include this information in the execution plan output by adding the `buffers on` option as follows:

```
EXPLAIN (analyze, buffers on)
SELECT username, level, score
FROM game.player_stats
WHERE player_id = 250;
```

Additionally, we use the `analyze` option, asking Postgres to execute the query after the plan is generated.

> **NOTE** The query uses `EXPLAIN (analyze)` instead of `EXPLAIN ANALYZE` because the latter does not support additional options; it is simply a shortcut for the former.

The generated plan looks as follows and includes information about shared buffer usage in the `Buffers` section:

```
                              QUERY PLAN
-----------------------------------------------------------------
 Index Scan using player_stats_pkey on player_stats
 ➥   (cost=0.29..8.30 rows=1 width=18)
 ➥   (actual time=0.042..0.044 rows=1 loops=1)
   Index Cond: (player_id = 250)
   Buffers: shared hit=3
 Planning Time: 0.119 ms
 Execution Time: 0.076 ms
(5 rows)
```

The `Buffers: shared hit=3` metric implies that Postgres read all the data from its shared buffers, which are preallocated memory spaces where Postgres caches data from regular tables and indexes. Specifically, it accessed three pages while scanning the index and retrieving the requested data from the table—without needing to load the data from disk. If the data needed for a query didn't fit in shared buffers, the `Buffers` metric might look like this: `Buffers: shared hit=3 read=4`. The `read=4` part implies that Postgres had to read four pages from disk while executing the query, and only three pages were already in memory (`hit=3`).

NOTE Throughout the book, we'll use the `EXPLAIN (analyze, costs off)` statement most of the time to analyze execution plans. But that doesn't mean this combination of options is a best practice or should be used in every case. The plans produced with `analyze, costs off` are compact enough to include in the book while still providing enough detail to understand how Postgres executes a particular query.

Overall, after using the `EXPLAIN` statement, we, the creators of the gaming platform, conclude that the discussed database query does not appear to be the root cause of the long load time for players' personal dashboards over the past 30 minutes. We need to continue investigating other components of the application stack. The `EXPLAIN` statement helped us determine the following:

- The query already uses the index on the `player_id` column for fast data access.
- The database reads the data from shared memory buffers without needing to access the slower disk.
- The actual execution time of the query seems to be reasonable for the given CPU resources.

TIP Refer to the Postgres official documentation to review additional options supported by the `EXPLAIN` command: https://www.postgresql.org/docs/current/sql-explain.html.

4.5 Single-column indexes

The primary key on the `player_stats.player_id` column from the previous section is an example of a single-column index that Postgres creates automatically. In this section, we learn how to create single-column indexes to optimize search performance for certain operations in our multiplayer online game. We also take the opportunity to review the differences between B-tree and hash indexes, helping you make the best choice when creating an index on a specific column.

4.5.1 Single-column B-tree indexes

As discussed earlier, indexes backed by the B-tree data structure allow us to search through data in logarithmic time. With this algorithmic complexity, search time remains comparable as data volume grows.

For example, our multiplayer game currently tracks 10,000 players in the `player_stats` table. Assuming the branching factor of the B-tree is 10, it would take $O(\log_{10} 10{,}000) = 4$ lookups to reach a leaf node containing a reference to the record for a specific `player_id`. If the game becomes a hit and the number of players grows to 1 million, it will take only $O(\log_{10} 1{,}000{,}000) = 6$ lookups to find a record for a given `player_id`.

NOTE Remember that B-trees are the default index type used by Postgres because they support a wide range of data types and can be used for both

equality searches (`=`) and range queries using greater than (`>`, `>=`), less than (`<`, `<=`), and `BETWEEN` operators.

Let's learn how to create single-column indexes using B-trees to build a leaderboard of the top players. The leaderboard ranks players by the `player_stats.score` column and needs to be recalculated in real time.

Before writing a SQL query to generate the leaderboard, let's use the `EXPLAIN` statement to check which data access method Postgres uses when the gaming platform filters data by the score. For example, the following query returns the number of players with a score between 5,000 and 6,000:

```
SELECT count(*)
FROM game.player_stats
WHERE score BETWEEN 5000 AND 6000;
```

The output looks as follows:

```
 count
-------
   970
(1 row)
```

Let's add the `EXPLAIN` statement to view the query's execution plan. Note that we pass the `costs off` option to make the execution plan more compact:

```
EXPLAIN (analyze, costs off)
SELECT count(*) FROM game.player_stats
WHERE score BETWEEN 5000 AND 6000;
```

The execution plan is as follows:

```
                                  QUERY PLAN
----------------------------------------------------------------------------
 Aggregate (actual time=1.702..1.703 rows=1 loops=1)
   ->  Seq Scan on player_stats (actual time=0.762..1.615 rows=970 loops=1)
         Filter: ((score >= 5000) AND (score <= 6000))
         Rows Removed by Filter: 9030
 Planning Time: 0.246 ms
 Execution Time: 1.753 ms
```

According to the plan, the database had to perform a sequential scan (`Seq Scan`), checking every record in the table to see if its `score` value satisfied the condition in the `Filter`. Only 970 rows met the search criteria (`rows = 970`); 9,030 did not (`Rows Removed by Filter: 9030`).

Currently, there is nothing wrong with using the `Seq Scan` access method because only 10,000 players are in the table, and it takes just over 1 ms to produce the result. However, as more players join the game, the search performance will degrade linearly, taking progressively longer to complete the full table scan.

To make sure search performance on the `player_stats.score` column doesn't grow linearly as the table grows, we can create an index on the `score` column as follows:

```
CREATE INDEX idx_score
ON game.player_stats(score DESC);
```

`idx_score` is the custom name we chose for the index, and the scores will be ordered in descending order (`DESC`) from highest to lowest.

Although it's not explicitly specified, this single-column index is backed by a B-tree. Let's run the following query to see the final definition of the index.

Listing 4.2 Score index details

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE schemaname = 'game'
  AND tablename = 'player_stats'
  AND indexdef LIKE '%idx_score%';
```

The query produces the following output:

```
 indexname |                    indexdef
-----------+---------------------------------------------
 idx_score | CREATE INDEX idx_score ON game.player_stats
           |    USING btree (score DESC)
(1 row)
```

The `indexdef` column shows the complete definition of the `CREATE INDEX` statement executed by Postgres. The statement specifies the B-tree as the underlying data structure with the `USING btree` clause.

With this new index in place, let's check the execution plan for the query that filters players by their scores once again:

```
EXPLAIN (analyze, costs off)
SELECT count(*) FROM game.player_stats
WHERE score BETWEEN 5000 AND 6000;
```

This time, the plan looks as follows:

```
                          QUERY PLAN
--------------------------------------------------------------
 Aggregate (actual time=0.303..0.303 rows=1 loops=1)
   ->  Index Only Scan using idx_score on player_stats
(actual time=0.078..0.198 rows=970 loops=1)
         Index Cond: ((score >= 5000) AND (score <= 6000))
         Heap Fetches: 0
 Planning Time: 0.164 ms
 Execution Time: 0.350 ms
```

Postgres uses the `idx_score` index to find the records that satisfy the `Index Cond` condition. Additionally, because the query only calculates the total number of players using the `count(*)` aggregate, the database performs an `Index Only Scan`, avoiding access to the `player_stats` table entirely (`Heap Fetches: 0`). This means the index already has all the data needed to complete the request, and Postgres doesn't need to access the table. As a result, in this particular example, the query execution time improved 5×, dropping from 1.75 ms with the `Seq Scan` method to 0.35 ms with the `Index Only Scan`.

Note that the execution time for this and the following queries may differ on your system. It can also vary depending on data volume, available system resources, and other runtime conditions.

Using the ANALYZE statement to update Postgres statistics

In real-world production deployments of Postgres, it's good practice to run the `ANALYZE` command after creating an index:

```
CREATE INDEX on tableA(columnB);
ANALYZE tableA;
```

When a new index is created, the query planner may not take advantage of it immediately and may continue relying on previously collected statistics. Running `ANALYZE` updates the table statistics, allowing the planner to take full advantage of the new index.

Now everything is set to start building a real-time leaderboard for the players. The SQL query that retrieves data for the leaderboard of the top five players is as simple as shown next.

Listing 4.3 Retrieving the player leaderboard

```
SELECT username, score, level
FROM game.player_stats
ORDER BY score DESC
LIMIT 5;
```

The query produces the following result:

```
    username    | score | level
----------------+-------+-------
 kfowler        | 10000 |    10
 tvillegas      | 10000 |    10
 ashley18       | 10000 |    10
 brookecampbell | 10000 |    10
 harrisoscar    |  9999 |    10
(5 rows)
```

Finally, let's use the `EXPLAIN` statement to verify that the query from listing 4.3 uses the index on the `player_stats.score` column:

```
EXPLAIN (analyze, costs off)
SELECT username, score, level
FROM game.player_stats
ORDER BY score DESC
LIMIT 5;
```

The execution plan is as straightforward as the query itself:

```
                           QUERY PLAN
--------------------------------------------------------------
 Limit (actual time=0.044..0.047 rows=5 loops=1)
   ->  Index Scan using idx_score on player_stats
(actual time=0.042..0.044 rows=5 loops=1)
 Planning Time: 0.311 ms
 Execution Time: 0.105 ms
```

Because the `idx_score` index already stores the score values in descending order, Postgres opts for an `Index Scan` and generates the leaderboard in well under a millisecond. The plan also shows that Postgres spent more time planning and selecting the final execution plan (`Planning Time: 0.311 ms`) than executing it (`Execution Time: 0.105 ms`). This can happen because the database may need extra time to evaluate available indexes and internal statistics to choose the most efficient execution strategy.

> **TIP** Indexes can also help optimize queries that modify data. For instance, if you need to update the `score` for a particular player identified by `player_id`, an index on the `player_id` column will help locate and update the record much more quickly. However, it's worth noting that because the update operation modifies the `score`, Postgres will also need to visit and update the index on the `score` column that we created earlier in this chapter, adding slightly to the latency of the update.

4.5.2 Single-column hash indexes

Hash indexes in Postgres work similarly to hash tables or hash maps found in most programming languages. The index computes a hash for the column value and uses it to map the value to a specific bucket. The algorithmic complexity of hash calculations and lookups is O(1), or constant time. This means that even as a table grows from thousands to millions of records, Postgres will spend roughly the same amount of time checking whether a value exists in the table by computing its hash and performing a quick lookup against the index.

Hash indexes are supported for single-column indexes and can only be used with the equality (`=`) operator. Range queries cannot take advantage of hash indexes and must use B-tree indexes instead.

Let's learn how to use hash indexes in practice by finding champions among the players in our multiplayer game. Five champion titles are used in the game, and they are transferable:

- Airforce Warlord
- Naval Warlord
- Land Warlord
- Cyber Warlord
- Space Warlord

Currently, thousands of players are in our game, but only five can hold champion titles. For example, if we want to find out who has the title of Naval Warlord, we can use the following query:

```
SELECT username, score, level
FROM game.player_stats
WHERE champion_title = 'Naval Warlord';
```

The current champion is the player with the `kenneth50` username:

```
 username  | score | level
-----------+-------+-------
 kenneth50 |  9739 |    10
(1 row)
```

How efficient is this query? Let's use the `EXPLAIN` statement to check the execution plan:

```
EXPLAIN (analyze, costs off)
SELECT username, score, level
FROM game.player_stats
WHERE champion_title = 'Naval Warlord';
```

The plan looks as follows:

```
                             QUERY PLAN
--------------------------------------------------------------------
 Seq Scan on player_stats (actual time=0.064..1.429 rows=1 loops=1)
   Filter: (champion_title = 'Naval Warlord'::text)
   Rows Removed by Filter: 9999
 Planning Time: 0.154 ms
 Execution Time: 1.463 ms
```

According to the plan, Postgres performed a full table scan (`Seq Scan`) to locate a single champion record. Without a `UNIQUE` constraint on the column (which would create a B-tree index automatically to validate the uniqueness), the database had to scan the entire table from start to finish, filtering out 9,999 records (`Rows Removed by Filter: 9999`).

We can improve the efficiency of champion lookups by creating an index on the `champion_title` column. Because only a few champions are in the table and we only need equality (`=`) and `IN` operators for their lookups, a hash index is an ideal fit for this column. The following query uses the `USING hash (champion_title)` clause to create a hash index on the column.

Listing 4.4 Creating a hash index

```
CREATE INDEX idx_champion_title
ON game.player_stats
USING hash(champion_title);
```

Once the index is created, we can check the execution plan for the query that retrieves the player holding the Naval Warlord title one more time:

```
EXPLAIN (analyze, costs off)
SELECT username, score, level
FROM game.player_stats
WHERE champion_title = 'Naval Warlord';
```

This time, Postgres uses the hash index to locate the champion record in well under a millisecond:

```
                               QUERY PLAN
-------------------------------------------------------------------
 Index Scan using idx_champion_title on player_stats
➥   (actual time=0.037..0.038 rows=1 loops=1)
   Index Cond: (champion_title = 'Naval Warlord'::text)
 Planning Time: 0.151 ms
 Execution Time: 0.073 ms
```

The database can also take advantage of the index in queries using the `IN` operator. For example, suppose we need to display all five champions. This can be done using the following query.

Listing 4.5 Getting all champions

```
SELECT username, champion_title, score, level
FROM game.player_stats
WHERE champion_title IN (
  'Airforce Warlord', 'Naval Warlord',
  'Land Warlord', 'Cyber Warlord',
  'Space Warlord');
```

The query produces the following list of champions:

```
  username     |  champion_title  | score | level
--------------+------------------+-------+-------
 gregoryscott | Land Warlord     |  9790 |    10
 rcastillo    | Space Warlord    |  9772 |    10
 kenneth50    | Naval Warlord    |  9739 |    10
 bpope        | Cyber Warlord    |  9684 |    10
 ldorsey      | Airforce Warlord |  9309 |    10
(5 rows)
```

Next, let's add the `EXPLAIN` statement to the query from listing 4.5 to check its execution plan:

```
EXPLAIN (analyze, costs off)
SELECT username, champion_title, score, level
FROM game.player_stats
WHERE champion_title IN (
  'Airforce Warlord', 'Naval Warlord',
  'Land Warlord', 'Cyber Warlord',
  'Space Warlord');
```

The database outputs the following plan:

```
                              QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on player_stats (actual time=0.068..0.082 rows=5 loops=1)
   Recheck Cond: (champion_title = ANY ('{"Airforce Warlord",
➥    "Naval Warlord", "Land Warlord","Cyber Warlord",
➥    "Space Warlord"}'::text[]))
   Heap Blocks: exact=4
   ->  Bitmap Index Scan on idx_champion_title
➥    (actual time=0.047..0.047 rows=5 loops=1)
         Index Cond: (champion_title = ANY ('{"Airforce Warlord",
➥    "Naval Warlord","Land Warlord","Cyber Warlord",
➥    "Space Warlord"}'::text[]))
 Planning Time: 0.242 ms
 Execution Time: 0.166 ms
```

This time, Postgres uses the bitmap index scan access method, which always executes in two phases: `Bitmap Index Scan` and `Bitmap Heap Scan`. In this example, the bitmap index scan works as follows:

- The `Bitmap Index Scan` searches through the `idx_champion_title` index, locating all index entries where the `champion_title` value satisfies the `Index Cond` condition. Each time Postgres finds a matching index entry, it doesn't immediately fetch `username`, `score`, and other data from a respective row in the `player_stats` table. Instead, it produces a bitmap structure that specifies all table rows satisfying the condition and that need to be accessed later—essentially preparing for a "bulk mode" retrieval.
- The `Bitmap Heap Scan` then accesses the `player_stats` table and retrieves `username`, `score`, and other data only for the rows pointed to by the bitmap. This way, the search is optimized by reducing the number of table lookups during the index scan. Instead of accessing the table each time the index finds a matching entry, the table is accessed just once, during the `Bitmap Heap Scan` phase.

Exact and lossy information in a bitmap

If the number of rows matching the index condition is high and the bitmap starts consuming more memory, Postgres begins storing location information about entire table pages instead of individual rows in the bitmap, which makes the bitmap lossy.

Additionally, a particular index type may choose to always specify locations only for pages in the bitmap—like the BRIN index does.

In this case, during the `Bitmap Heap Scan` phase, Postgres visits each page from the bitmap marked as lossy and rechecks the index condition for every row in that page. This is necessary because a page usually stores many rows, but not all of the rows may satisfy the search criteria.

The `Heap Blocks` section of the `Bitmap Heap Scan` phase clarifies what information is actually stored in the bitmap:

- `Heap Blocks: exact=X` means that *X* pages were visited during the `Bitmap Heap Scan` phase, and the bitmap provided the exact locations of the rows within those pages that matched the search condition. For example, in the previous query, there were four such pages (`Heap Blocks: exact=4`) and Postgres knew exactly which rows from those pages to fetch.
- `Heap Blocks: lossy=Y` means that *Y* pages were visited during the `Bitmap Heap Scan`, but the bitmap didn't specify which rows within those *Y* pages matched the search criteria. The information about those *Y* pages was lossy. As a result, Postgres had to scan every row in those *Y* pages and recheck the index condition.

Finally, a bitmap can store both exact and lossy information, in which case the `Heap Blocks` section will look like this: `Heap Blocks: exact=X, lossy=Y`.

In this section, we explored how to use single-column indexes backed by B-tree and hash-based data structures to speed up the creation of the player leaderboard and champion lookups. In the next section, we'll dive into composite indexes and use them to further optimize the performance of our multiplayer online game.

4.6 Composite indexes

Composite indexes in Postgres are created over multiple columns of a table, as opposed to single-column indexes, which cover only one column. Composite indexes in Postgres are also referred to as *multicolumn indexes*. Currently, the database supports creating composite indexes using B-tree, GIN, GiST, BRIN, or bloom filters as the underlying data structures.

The way composite indexes work is straightforward. Imagine an index on the columns `(region, score)`. The index sorts the data by the first column (`region`) and then by the second column (`score`) within each `region`. The first column is referred to as the *leading column* and must be included in the query for Postgres to consider using the index. For example, if the leading column (`region`) is not used in the query's `WHERE` clause and only the second column (`score`) is specified, the composite index won't be used because it is sorted by the `region` column first, which is missing in the query.

Let's explore how and when to use composite indexes in Postgres by creating a leaderboard with the top gamers in each distinct geographical region of our multiplayer game using the query from listing 4.6. Note that `DISTINCT ON` helps to return the first

row for each unique `region` value based on the specified sort order. In our case, Postgres will return the player with the highest `score` (and highest `win_count` in case of ties) for each `region`.

Listing 4.6 Finding the top gamer in each distinct region

```
SELECT DISTINCT ON (region)
username, region, score, win_count
FROM game.player_stats
ORDER BY region, score DESC, win_count DESC;
```

However, before checking the execution plan and considering the creation of a composite index for this query, we'll start with a simpler scenario, as shown next: it finds all players in a specific region with a particular score and win count.

Listing 4.7 Finding regional players with a specific score and win count

```
SELECT username, region, score, win_count
FROM game.player_stats
WHERE region = 'NA' and score > 5000 and win_count > 10
ORDER BY score DESC, win_count DESC;
```

We need to review these scenarios together because the queries from both listings 4.6 and 4.7 are frequently executed against the database and require indexes for fast access. At the same time, we should aim to minimize the number of indexes in the database, as each additional index must be maintained by Postgres.

4.6.1 Considering an additional single-column index

If we execute the query from listing 4.7, the truncated output for players from North America who've scored more than 5,000 points and have more than 10 wins looks as follows:

```
      username       | region | score | win_count
---------------------+--------+-------+-----------
 kfowler             | NA     | 10000 |       500
 brookecampbell      | NA     | 10000 |       186
 megan10             | NA     |  9994 |        12
 bishopbenjamin      | NA     |  9990 |       289
 ann49               | NA     |  9989 |        73
 jonesjacob          | NA     |  9983 |        59

... more players
```

Now, let's run the query again, this time adding `EXPLAIN (analyze, costs off)` to view the execution plan chosen by Postgres:

```
EXPLAIN (analyze, costs off)
SELECT username, region, score, win_count
```

```
FROM game.player_stats
WHERE region = 'NA' and score > 5000 and win_count > 10
ORDER BY score DESC, win_count DESC;
```

According to the plan, the database is already using the existing index named `idx_score`, which we created earlier for fast searches on the score column:

```
                             QUERY PLAN
--------------------------------------------------------------------
 Incremental Sort (actual time=0.186..2.107 rows=1580 loops=1)
   Sort Key: score DESC, win_count DESC
   Presorted Key: score
   Full-sort Groups: 50  Sort Method: quicksort  Average Memory: 26kB
 ➥   Peak Memory: 26kB
   -> Index Scan using idx_score on player_stats
 ➥   (actual time=0.066..1.575 rows=1580 loops=1)
         Index Cond: (score > 5000)
         Filter: ((win_count > 10) AND (region = 'NA'::text))
         Rows Removed by Filter: 3361
 Planning Time: 0.203 ms
 Execution Time: 2.220 ms
```

The database filtered out 3,361 players (`Rows Removed by Filter: 3361`) after finding all records with a score above 5,000 (`Index Cond: (score > 5000)`). It then sorted the results (`Sort Key: score DESC, win_count DESC`) before returning them to the application. Despite the additional filtering and sorting steps, this approach is still more efficient than a full table scan over 10,000 players, which is why Postgres selected the `idx_score` index.

However, this doesn't mean the `idx_score` index will always be used for the query. Let's see what happens if we change the condition in the query from `score > 5000` to `score > 1000`, aiming to find all players in North America with a score higher than 1,000:

```
EXPLAIN (analyze, costs off)
SELECT username, region, score, win_count
FROM game.player_stats
WHERE region = 'NA' and score > 1000 and win_count > 10
ORDER BY score DESC, win_count DESC;
```

This time, the execution plan looks different, with Postgres no longer using the index on the `score` column:

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Sort (actual time=2.905..3.066 rows=2916 loops=1)
   Sort Key: score DESC, win_count DESC
   Sort Method: quicksort  Memory: 238kB
   ->  Seq Scan on player_stats (actual time=0.028..2.031 rows=2916 loops=1)
         Filter: ((score > 1000) AND (win_count > 10)
 ➥       AND (region = 'NA'::text))
         Rows Removed by Filter: 7084
 Planning Time: 0.234 ms
 Execution Time: 3.210 ms
```

As you can see, the database performed a full table scan (`Seq Scan on player_stats`) because far more players have a score above 1,000. Using the index would have returned too many records for further processing, making it less efficient. Thus, the database selected the sequential scan instead, which the Postgres planner considered to be equally or more efficient than using the index.

One way to optimize this query is by creating an additional single-column index on the `region` column using the following statement:

```
CREATE INDEX idx_region ON game.player_stats (region);
```

Once the index is created, the execution plan for the query that retrieves players in North America with `score > 1000` and `win_count > 10` looks as follows:

```
                              QUERY PLAN
-------------------------------------------------------------------
 Sort (actual time=1.880..1.995 rows=2916 loops=1)
   Sort Key: score DESC, win_count DESC
   Sort Method: quicksort  Memory: 238kB
   ->  Bitmap Heap Scan on player_stats
 ➥   (actual time=0.128..1.388 rows=2916 loops=1)
         Recheck Cond: (region = 'NA'::text)
         Filter: ((score > 1000) AND (win_count > 10))
         Rows Removed by Filter: 391
         Heap Blocks: exact=137
         ->  Bitmap Index Scan on idx_region
 ➥   (actual time=0.098..0.098 rows=3307 loops=1)
               Index Cond: (region = 'NA'::text)
 Planning Time: 0.179 ms
 Execution Time: 2.112 ms
```

This time, the database opted for a `Bitmap Index Scan` on the newly created `idx_region` index, which improved the execution time compared to the full table scan.

However, none of the existing single-column indexes can be used for the query in listing 4.6, which returns the top gamer in each region. Let's run the following statement to check its execution plan:

```
EXPLAIN (analyze, costs off)
SELECT DISTINCT ON (region)
username, region, score, win_count
FROM game.player_stats
ORDER BY region, score DESC, win_count DESC;
```

According to the plan, Postgres still defaults to a full table scan:

```
                              QUERY PLAN
------------------------------------------------------------------
 Unique (actual time=13.148..14.314 rows=3 loops=1)
   ->  Sort (actual time=13.145..13.591 rows=10000 loops=1)
         Sort Key: region, score DESC, win_count DESC
```

```
        Sort Method: quicksort  Memory: 881kB
        ->  Seq Scan on player_stats
➥   (actual time=0.065..1.969 rows=10000 loops=1)
Planning Time: 0.183 ms
Execution Time: 14.357 ms
```

The database had to iterate through all 10,000 records in the table (`Seq Scan on player_stats (... rows=10000 ...)`) just to find three players at the final stage (`Unique (... rows=3 ...)`) who happened to be the top players in their respective geographical regions. This is far from optimal, especially as the number of players continues to grow.

Rather than creating additional single-column indexes for various queries, let's create a composite index that will work for both queries in listings 4.6 and 4.7. However, before doing so, let's drop the index on the `region` column, as it doesn't fully meet our needs:

```
DROP INDEX game.idx_region;
```

4.6.2 Creating a composite index

The next listing shows the composite index we need to create to optimize both queries from the previous section.

Listing 4.8 Creating a composite index

```
CREATE INDEX idx_region_score_win_count
ON game.player_stats (region, score DESC, win_count DESC);
```

The index is built on three columns: `region`, `score`, and `win_count`. The order of the columns in the definition determines how the data is sorted within the index:

1. The index first sorts and arranges the data by the `region` column.
2. Within each unique `region`, the data is sorted by `score` in descending order.
3. Within each unique combination of `region` and `score`, the data is sorted by `win_count`, also in descending order.

Like single-column indexes, composite indexes use the B-tree data structure by default. We can confirm this by checking the final `CREATE INDEX` statement definition using the following query:

```
SELECT indexdef FROM pg_indexes
WHERE schemaname = 'game'
  AND tablename = 'player_stats'
  AND indexdef LIKE '%idx_region_score_win_count%';
```

The query produces the following output, with `btree` specified in the `USING` clause:

```
                                indexdef
---------------------------------------------------------------------
 CREATE INDEX idx_region_score_win_count
 ON game.player_stats
 USING btree (region, score DESC, win_count DESC)
(1 row)
```

With this new composite index in place, let's check the execution plan for the query that returns players in North America with `score > 1000` and `win_count > 10`:

```
EXPLAIN (analyze, costs off)
SELECT username, region, score, win_count
FROM game.player_stats
WHERE region = 'NA' and score > 1000 and win_count > 10
ORDER BY score DESC, win_count DESC;
```

The execution plan looks as follows:

```
                                QUERY PLAN
-----------------------------------------------------------------------
 Sort (actual time=2.517..2.682 rows=2916 loops=1)
   Sort Key: score DESC, win_count DESC
   Sort Method: quicksort  Memory: 238kB
   ->  Bitmap Heap Scan on player_stats
➥    (actual time=0.472..1.812 rows=2916 loops=1)
         Recheck Cond: ((region = 'NA'::text) AND (score > 1000)
➥        AND (win_count > 10))
         Heap Blocks: exact=124
         ->  Bitmap Index Scan on idx_region_score_win_count
➥            (actual time=0.434..0.434 rows=2916 loops=1)
            Index Cond: ((region = 'NA'::text) AND (score > 1000)
➥           AND (win_count > 10))
 Planning Time: 0.287 ms
 Execution Time: 2.844 ms
```

Postgres uses the newly created `idx_region_score_win_count` index to perform a `Bitmap Index Scan`. This approach is as efficient as the single-column index on the `region` column that we experimented with earlier.

However, this composite index has a far greater effect on the query that returns the top players in each region. Let's check the latest execution plan for that query:

```
EXPLAIN (analyze, costs off)
SELECT DISTINCT ON (region)
username, region, score, win_count
FROM game.player_stats
ORDER BY region, score DESC, win_count DESC;
```

The query returns the following execution plan, confirming that the database performs an index scan on the composite index named `idx_region_score_win_count`:

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Result (actual time=0.038..3.994 rows=3 loops=1)
   ->  Unique (actual time=0.034..3.984 rows=3 loops=1)
         ->  Index Scan using idx_region_score_win_count
 ➥   on player_stats (actual time=0.032..2.698 rows=10000 loops=1)
 Planning Time: 0.147 ms
 Execution Time: 4.038 ms
```

Moreover, there is no additional sorting step in the plan because the `score` and `win_count` columns are already presorted in descending order in the index definition.

With this single composite index, we optimized the performance of both the query that retrieves all players in a specific region based on score and win count, and the query that generates a leaderboard of top gamers in each geographical location. Now, let's explore how we can further use this composite index and what its limitations are.

4.6.3 Caveats of composite indexes

Composite indexes are a powerful tool in a developer's toolbox, but there are times when they may not work as expected. Let's review a few caveats related to this type of index.

As we discussed, the order of the columns in the index definition matters because it dictates how the columns should be listed in the `ORDER BY` clause. For example, let's say we decide to change the criteria for the leaderboard of top players in each region. Presently, the players are ranked by `score` first, followed by `win_count` (see the order of the columns in the `ORDER BY` clause):

```
SELECT DISTINCT ON (region)
username, region, score, win_count
FROM game.player_stats
ORDER BY region, score DESC, win_count DESC;
```

With these criteria, the leaderboard looks as follows, and we already know that Postgres uses the existing composite index to generate it efficiently:

```
  username    | region | score | win_count
-------------+--------+-------+-----------
 tvillegas   | APAC   | 10000 |       482
 harrisoscar | EMEA   |  9999 |       438
 kfowler     | NA     | 10000 |       500
(3 rows)
```

Now we want to update the ranking criteria by ordering players first by the number of wins and then by score. Essentially, we are swapping the order of the `win_count` and `score` columns in the `ORDER BY` clause.

Listing 4.9 Finding the top regional players by win count and score

```
SELECT DISTINCT ON (region)
username, region, score, win_count
```

```
FROM game.player_stats
ORDER BY region, win_count DESC, score DESC;
```

By making this minor change, we get a different result, which is expected:

```
 username   | region | score | win_count
------------+--------+-------+-----------
 foxdavid   | APAC   |  9694 |       500
 rcastillo  | EMEA   |  9772 |       500
 kfowler    | NA     | 10000 |       500
(3 rows)
```

What might not be expected, however, is that Postgres can no longer use our composite index. Let's add the `EXPLAIN` statement to the query to see its execution plan:

```
EXPLAIN (analyze, costs off)
SELECT DISTINCT ON (region)
username, region, score, win_count
FROM game.player_stats
ORDER BY region, win_count DESC, score DESC;
```

The plan looks as follows:

```
                            QUERY PLAN
----------------------------------------------------------------------
 Unique (actual time=17.596..19.245 rows=3 loops=1)
   ->  Sort (actual time=17.593..18.300 rows=10000 loops=1)
         Sort Key: region, win_count DESC, score DESC
         Sort Method: quicksort  Memory: 881kB
         ->  Seq Scan on player_stats
➥            (actual time=0.025..2.282 rows=10000 loops=1)
 Planning Time: 0.177 ms
 Execution Time: 19.287 ms
```

Postgres had to perform a full table scan (`Seq Scan on player_stats`), sorting the result according to the order specified in the query (`Sort Key: region, win_count DESC, score DESC`). This order differs from the column order in the index definition from listing 4.8, which is `region, score DESC, win_count DESC`. This difference in ordering prevents Postgres from using the existing composite index.

Another caveat of composite indexes arises when we need to filter data using only a subset of the indexed columns. For example, let's say our gaming platform needs to find the total number of players in the EMEA region (Europe, the Middle East, and Africa). This is how we can see the execution plan for the query that provides this information:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM game.player_stats
WHERE region = 'EMEA';
```

According to the execution plan, the database is using our composite index:

```
                              QUERY PLAN
----------------------------------------------------------------------
 Aggregate (actual time=0.526..0.526 rows=1 loops=1)
   ->  Index Only Scan using idx_region_score_win_count
➥      on player_stats (actual time=0.046..0.375 rows=3277 loops=1)
         Index Cond: (region = 'EMEA'::text)
         Heap Fetches: 0
 Planning Time: 0.201 ms
 Execution Time: 0.564 ms
```

If we modify the condition to find the number of players in EMEA who scored more than 1,000 points, the updated query will look as follows:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM game.player_stats
WHERE region = 'EMEA' and score > 1000;
```

The execution plan shows that Postgres continues using the same composite index to generate the result:

```
                           QUERY PLAN
----------------------------------------------------------------
 Aggregate (actual time=0.772..0.773 rows=1 loops=1)
   ->  Index Only Scan using idx_region_score_win_count
➥       on player_stats (actual time=0.061..0.584 rows=2972 loops=1)
         Index Cond: ((region = 'EMEA'::text) AND (score > 1000))
         Heap Fetches: 0
 Planning Time: 0.188 ms
 Execution Time: 0.819 ms
```

However, if we modify the query to find the total number of players with `score > 1000` and `win_count > 30`, the database can no longer use the composite index. The updated query is as follows:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM game.player_stats
WHERE score > 1000 and win_count > 30;
```

And Postgres selects the following execution plan:

```
                                 QUERY PLAN
--------------------------------------------------------------------------
 Aggregate (actual time=2.502..2.503 rows=1 loops=1)
   ->  Seq Scan on player_stats (actual time=0.022..1.986 rows=8503 loops=1)
         Filter: ((score > 1000) AND (win_count > 30))
         Rows Removed by Filter: 1497
 Planning Time: 0.175 ms
 Execution Time: 2.549 ms
```

This time, the database had to perform a full table scan (`Seq Scan on player_stats`) to calculate the total number of players satisfying the `Filter` condition. Postgres couldn't use the composite index because the index is sorted first by the `region` column, which is not included in the query.

Finally, let's see what happens if we create a query that uses the first (`region`) and third (`win_count`) columns from the composite index definition (`region, score DESC, win_count DESC`) while skipping the second (`score`). Suppose we want to retrieve all players in EMEA with a `win_count` above 30 using the following query:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM game.player_stats
WHERE region = 'EMEA' and win_count > 30;
```

The execution plan looks as follows:

```
                              QUERY PLAN
------------------------------------------------------------------------
 Aggregate (actual time=0.784..0.785 rows=1 loops=1)
   ->  Index Only Scan using idx_region_score_win_count
➥        on player_stats (actual time=0.074..0.590 rows=3082 loops=1)
         Index Cond: ((region = 'EMEA'::text) AND (win_count > 30))
         Heap Fetches: 0
 Planning Time: 0.296 ms
 Execution Time: 0.838 ms
```

Postgres decided to use our composite index, even though the `score` column is not included in the query. This execution plan is still more efficient than a full table scan on the `player_stats` table because the database can quickly locate all records with `region = EMEA` and then scan that subset of the index tree to find nodes satisfying the `win_count > 30` condition.

> **NOTE** Even though Postgres used the composite index in the last scenario, where the second column (`score`) from the index definition was skipped, this behavior may not apply to other composite indexes. In general, if a query skips any leading or intermediate columns in a composite index definition, the chances of the database using the index are lower in Postgres 17 or earlier versions. However, starting with Postgres 18, the database introduced support for skip scan lookups on composite B-tree indexes, allowing us to skip leading columns and still use the index in more scenarios.

In this section, we've learned how composite indexes can optimize queries that filter data across multiple columns and how a single composite index can replace several single-column indexes without compromising performance. Next, let's explore covering indexes, which can further optimize performance by reducing the number of lookups to table data.

4.7 Covering indexes

A *covering index* is an index that stores additional columns from the table, enabling the database to retrieve column values directly from the index without accessing the table rows. The additional columns are added to the index using the `INCLUDE` clause:

```
CREATE INDEX index_name ON table_name(columnA)
INCLUDE(columnB, columnC [, additional_columns]);
```

This statement creates a single-column index sorted by `columnA`. The included columns `columnB` and `columnC` are not used to build or sort the index structure. Instead, their values are simply stored in the index for faster data access.

For example, suppose there is a query that selects the value of `columnB` and orders the result by `columnC` as follows:

```
SELECT columnB FROM table_name WHERE columnA > 50 ORDER BY columnC;
```

In this case, the database doesn't need to access table `table_name` to retrieve the values of `columnB` and `columnC`. Instead, it can obtain these values directly from the index, improving performance. However, as a tradeoff, all included columns must remain consistent with the table data. This requires Postgres to update the index whenever `columnB` or `columnC` is modified by the application logic, which increases latency for queries updating the columns.

Now, let's return to our multiplayer game and see how to take advantage of covering indexes in practice. While exploring composite indexes, we used the following SQL query, which retrieves all players in a specific region with a particular score and win count:

```
EXPLAIN (analyze, costs off)
SELECT username, region, score, win_count
FROM game.player_stats
WHERE region = 'NA' and score > 5000 and win_count > 10
ORDER BY score DESC, win_count DESC;
```

We've optimized this query with its execution plan, which currently looks as follows:

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort (actual time=1.574..1.662 rows=1580 loops=1)
   Sort Key: score DESC, win_count DESC
   Sort Method: quicksort  Memory: 125kB
   ->  Bitmap Heap Scan on player_stats
➥       (actual time=0.249..1.005 rows=1580 loops=1)
         Recheck Cond: ((region = 'NA'::text) AND (score > 5000)
➥        AND (win_count > 10))
         Heap Blocks: exact=68
         ->  Bitmap Index Scan on idx_region_score_win_count
➥              (actual time=0.225..0.225 rows=1580 loops=1)
```

```
              Index Cond: ((region = 'NA'::text) AND (score > 5000)
➥             AND (win_count > 10))
 Planning Time: 0.342 ms
 Execution Time: 1.856 ms
```

This query uses the existing `idx_region_score_win_count` index to perform the `Bitmap Index Scan` over the indexed data. Once the bitmap is created, the database proceeds with the `Bitmap Heap Scan`, accessing all table pages with rows marked to be visited in the bitmap. Specifically, during the table scan phase, Postgres accesses 68 pages (`Heap Blocks: exact=68`) and retrieves the actual rows specified in the bitmap. The `username` column is the reason the database had to build the bitmap during the index scan and then visit specific table pages. The column is selected in the query but not stored in the index and therefore must always be retrieved from the table.

Now, suppose our gaming platform executes this query frequently. It is one of our "hot" queries, run numerous times by the application. To optimize it further, we decide to create a covering index that adds the username column to the index structure.

The new covering index will be similar to the existing composite index, with the exception that it will include the `username` column. To avoid having two comparable indexes, let's first drop the existing composite index:

```
DROP INDEX game.idx_region_score_win_count;
```

Next, we create the new covering index using the following definition.

Listing 4.10 Creating a covering index

```
CREATE INDEX idx_composite_covering_index
ON game.player_stats (region, score DESC, win_count DESC)
INCLUDE (username);
```

The index is still built on three columns (`region, score, win_count`), but it now also stores the value of the `username` column in the index structure. As a result, once the index is created, we can check the execution plan for the query one more time:

```
EXPLAIN (analyze, costs off)
SELECT username, region, score, win_count
FROM game.player_stats
WHERE region = 'NA' and score > 5000 and win_count > 10
ORDER BY score DESC, win_count DESC;
```

The new plan looks as follows:

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Only Scan using idx_composite_covering_index on player_stats
➥   (actual time=0.076..0.497 rows=1580 loops=1)
   Index Cond: ((region = 'NA'::text) AND (score > 5000)
```

```
➥   AND (win_count > 10))
   Heap Fetches: 0
 Planning Time: 0.320 ms
 Execution Time: 0.602 ms
```

This time, Postgres performed the `Index Only Scan`, meaning the database didn't need to access the table at all. It found all index records satisfying the `Index Cond` and retrieved the corresponding username directly from those records. There were no lookups to the table (`Heap Fetches: 0`), which improved the execution time to 0.6 ms: 3× faster than the previous execution time of 1.8 ms with the `Bitmap Index Scan` method.

Additionally, because users rarely update their usernames, including this column in the covering index has minimal effect on the latency of the write workload. Postgres will only need to update the `username` column in both the table and index occasionally.

4.8 Partial indexes

A *partial index,* also known as a *filtered index,* is an index created over a subset of a table that satisfies a specific condition. The condition can range from a simple check of a column value to a more complex expression involving multiple columns. The condition is defined in the `WHERE` clause of the index creation statement:

```
CREATE INDEX partial_index_name ON table_name(columnA) WHERE condition;
```

In this example, the partial index is created on `columnA` but only for rows that meet the specified `condition`.

Partial indexes are particularly useful in scenarios where you need to optimize queries for a specific subset of data while ignoring the rest. Additionally, partial indexes are smaller in size and result in faster table updates because the index doesn't need to be updated for rows that don't meet the condition. For example, if you have a large table but your application frequently queries only a specific subset of its data, a partial index can optimize those queries and make more efficient use of system resources like memory and disk.

Let's see how to benefit from partial indexes in practice. Imagine that our multiplayer game needs to track new or occasional players and incentivize them. An occasional player is someone who has played fewer than 50 hours.

There are 10,000 players in the game, and we expect this number to grow rapidly:

```
SELECT count(*) FROM game.player_stats;

 count
-------
 10000
(1 row)
```

Among these players, currently just over 70 are recognized as occasional players, representing less than 1% of the total number of players:

```
SELECT count(*) FROM game.player_stats WHERE play_time <= '50 hours';

 count
-------
    74
(1 row)
```

Because we plan to track the behavior of occasional players in real time to better understand why some stop playing and which reward strategies work best, it makes sense to create a partial index that includes only the rows for occasional players.

Listing 4.11 Querying occasional players

```
SELECT username, play_time, score, last_active
FROM game.player_stats
WHERE play_time <= '50 hours'
ORDER BY play_time;
```

The output for the first five players from the query looks as follows:

```
                        Table "game.player_stats"
    username      |     play_time     | score |          last_active
------------------+-------------------+-------+----------------------------
 justinrodriguez | 04:00:00          |    34 | 2025-07-07 03:29:23.05813
 clifford28      | 04:00:00          |    16 | 2023-06-17 03:29:22.068583
 robinsonanne    | 05:00:00          |    57 | 2024-03-20 03:29:22.274161
 pattonkristen   | 07:00:00          |    25 | 2025-03-08 03:29:21.133161
 elizabethrhodes | 08:00:00          |    76 | 2025-05-08 03:29:22.559829
```

Now let's add the `EXPLAIN (analyze, costs off)` clause to the beginning of this query to see its execution plan:

```
EXPLAIN (analyze, costs off)
SELECT username, play_time, score, last_active
FROM game.player_stats
WHERE play_time <= '50 hours'
ORDER BY play_time;
```

The database performs a full table scan (`Seq Scan`) on the `player_stats` table and then sorts the final result before returning it to the gaming platform:

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Sort (actual time=2.125..2.129 rows=74 loops=1)
   Sort Key: play_time
   Sort Method: quicksort  Memory: 30kB
   ->  Seq Scan on player_stats (actual time=2.015..2.085 rows=74 loops=1)
         Filter: (play_time <= '50:00:00'::interval)
         Rows Removed by Filter: 9926
 Planning Time: 0.205 ms
 Execution Time: 2.168 ms
```

Although the current execution time is reasonable, it will increase linearly as more players join the game. To optimize this query, let's create the partial index shown next.

Listing 4.12 Creating a partial index

```
CREATE INDEX idx_occasional_players
ON game.player_stats (play_time)
WHERE play_time <= '50 hours';
```

This index is built on the `play_time` column only for players whose `play_time` is equal to or less than 50 hours.

After creating the index, we can check the execution plan for the following query one more time:

```
EXPLAIN (analyze, costs off)
SELECT username, play_time, score, last_active
FROM game.player_stats
WHERE play_time <= '50 hours'
ORDER BY play_time;
Now, the plan looks as follows:
                               QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using idx_occasional_players on player_stats
 ➥   (actual time=0.036..0.128 rows=74 loops=1)
 Planning Time: 0.334 ms
 Execution Time: 0.168 ms
```

Postgres now uses our partial index, `idx_occasional_players`, performing the `Index Scan` on the data belonging to occasional players. The query execution time has improved significantly, dropping from 2 ms with a full table scan to 0.1 ms with the partial index: a 20× boost! Additionally, the partial index is smaller than it would have been if we had indexed the entire table on the `play_time` column.

The database will consider using the index for query execution whenever the `play_time` value is equal to or less than 50 hours. However, if the `play_time` value exceeds 50 hours—by even 1 second—Postgres will ignore the partial index for query execution. Let's check the execution plan for the following query to confirm:

```
EXPLAIN (analyze, costs off)
SELECT username, play_time, score, last_active
FROM game.player_stats
WHERE play_time <= '50 hours 1 second'
ORDER BY play_time;
```

The database now reverts to a full table scan because the partial index does not include data for players with `play_time <= '50 hours 1 seconds'`:

```
                                 QUERY PLAN
---------------------------------------------------------------------------
 Sort (actual time=1.886..1.890 rows=74 loops=1)
   Sort Key: play_time
   Sort Method: quicksort  Memory: 30kB
   ->  Seq Scan on player_stats (actual time=1.797..1.854 rows=74 loops=1)
         Filter: (play_time <= '50:00:01'::interval)
         Rows Removed by Filter: 9926
 Planning Time: 0.183 ms
 Execution Time: 1.921 ms
```

Partial indexes let us index only the rows that meet a specific condition, improving performance for targeted queries while reducing storage overhead.

4.9 Functional and expression indexes

Functional indexes, also known as *expression indexes,* allow us to index the result of a function or expression applied to one or more columns of a table. This type of index is defined using the following format:

```
CREATE INDEX functional_index_name
ON table_name(function_or_expression(columnA ,[additional_columns]))
```

The function can be a built-in Postgres function, such as `sum(expression)` or `lower(text)`, as well as a custom function or expression defined by your application.

Postgres will consider using the index whenever the function or expression appears in the `WHERE` or `ORDER BY` clause of a query, as shown in the following example:

```
SELECT columnA, columnB FROM table_name
WHERE function_or_expression(columnA, [additional_columns])
```

Let's explore how to use this type of index in practice by optimizing the search for players based on their performance margin, defined as the difference between wins and losses. Suppose our multiplayer game executes the following query to retrieve a list of players with a performance margin within a specific range.

Listing 4.13 Finding players list by performance margin

```
SELECT username, win_count, loss_count, (win_count - loss_count) as margin
FROM game.player_stats
WHERE (win_count - loss_count) BETWEEN 300 and 450
ORDER BY margin DESC;
```

The execution plan for the query looks as follows:

```
                                QUERY PLAN
------------------------------------------------------------------------
 Sort (actual time=2.387..2.459 rows=731 loops=1)
   Sort Key: ((win_count - loss_count)) DESC
```

```
  Sort Method: quicksort  Memory: 61kB
  ->  Seq Scan on player_stats (actual time=0.079..2.188 rows=731 loops=1)
        Filter: (((win_count - loss_count) >= 300)
➥       AND ((win_count - loss_count) <= 450))
        Rows Removed by Filter: 9269
Planning Time: 0.183 ms
Execution Time: 2.524 ms
```

According to the execution plan, the database performed a full table scan (`Seq Scan on player_stats`), applying the expression from the `Filter` to the `win_count` and `loss_count` columns of every row. Postgres found 731 rows that satisfied the search condition (`Sort (...rows=731)`) and filtered out thousands of rows that did not meet the criteria (`Rows Removed by Filter: 9269`).

Because this query is executed frequently by our application, it's reasonable to create an expression index on the result of the `(win_count - loss_count)` expression. Let's create such an index using the following statement.

Listing 4.14 Creating an expression index

```
CREATE INDEX idx_perf_margin
ON game.player_stats ((win_count - loss_count));
```

Once the expression index is created, we can revisit the execution plan for the query from listing 4.13:

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Sort (actual time=1.029..1.090 rows=731 loops=1)
   Sort Key: ((win_count - loss_count)) DESC
   Sort Method: quicksort  Memory: 61kB
   ->  Bitmap Heap Scan on player_stats
➥        (actual time=0.093..0.853 rows=731 loops=1)
         Recheck Cond: (((win_count - loss_count) >= 300)
➥        AND ((win_count - loss_count) <= 450))
         Heap Blocks: exact=136
         ->  Bitmap Index Scan on idx_perf_margin
➥              (actual time=0.061..0.061 rows=731 loops=1)
               Index Cond: (((win_count - loss_count) >= 300)
➥              AND ((win_count - loss_count) <= 450))
 Planning Time: 0.177 ms
 Execution Time: 1.200 ms
```

As shown, Postgres now utilizes the newly created index, performing the `Bitmap Index Scan` on the result of the `(win_count - loss_count)` expression. The execution time is already twice as fast as the full table scan access method, and this difference will only continue to grow as more players join our multiplayer game.

Beware of over-indexing

Throughout this chapter, we've explored and added various indexes to the `game.player_stats` table, bringing the total number of indexes for the table to seven:

```
SELECT count(*) FROM pg_indexes
WHERE schemaname = 'game' AND tablename = 'player_stats';
 count
-------
     7
(1 row)
```

Although this is acceptable for learning purposes, in practice, it represents a classic case of *over-indexing*, where the table has too many indexes, suggesting a need to reevaluate its indexing strategy.

Indexes can significantly optimize query performance, but they don't come for free. Each time we create a new index, Postgres must maintain it by updating its structure whenever the value of an indexed column changes in the primary table. The more indexes you have and the more frequently the table is updated, the greater the likelihood that Postgres will need to update the existing indexes to ensure that they remain in a consistent state and continue to deliver their $O(\log_b N)$ or other index-specific algorithmic complexity.

Apart from the index maintenance aspects, the more indexes Postgres has, the more time it will spend in the planning phase while selecting and generating the most efficient execution plan for a query.

Note that we discuss over-indexing and under-indexing in more detail in appendix A of the book.

Summary

- Indexes are a popular optimization technique because they improve search performance by querying only a subset of the indexed data. Many indexes have an algorithmic complexity of $O(\log_b N)$.
- Indexes in Postgres can be categorized by index scope or functional purpose (what we index) and by underlying data structure and access method (how we index).
- Postgres uses B-tree as the default underlying data structure (how we index).
- The `EXPLAIN` statement lets us analyze the query execution plan so that we can decide on the most efficient query optimization techniques.
- Single-column and composite indexes optimize queries that filter or order data by one or more columns.
- Covering indexes include table data in the index structure, reducing the number of lookups to the table.

- Partial indexes are useful when indexing only a subset of table data.
- Functional and expression indexes allow indexing the result of a function or expression applied to one or more table columns.

Part 2

Core Postgres beyond relational

Over the years, Postgres has evolved into a general-purpose database that supports use cases beyond traditional relational workloads. In the second part of the book, we explore two core capabilities of Postgres that let us use the database to process semi-structured and unstructured text data.

We learn how to efficiently work with JSON data and perform full-text searches in Postgres using built-in operators, functions, and specialized index types. After completing this part of the book, we'll be prepared to design and build applications that take full advantage of these core capabilities of Postgres.

5 Postgres and JSON

This chapter covers

- Storing JSON data in Postgres
- Querying JSON data using built-in operators and functions
- Updating specific fields in existing JSON objects
- Optimizing JSON data access with GIN and B-tree indexes

Postgres introduced initial support for JSON back in 2012. Since then, JSON-specific capabilities and improvements have been added on a regular basis. Today, Postgres offers mature and comprehensive JSON support so that it can easily function as a document database, storing and processing JSON objects of various complexity.

Let's explore Postgres's JSON capabilities as we build an application for a pizza chain with hundreds of locations nationwide. We'll learn how to store customer order details in JSON format and then work with this data while fulfilling orders and analyzing customers' pizza preferences.

5.1 Storing JSON data

Imagine that we are part of the team building the application for the pizza chain. When a customer places a pizza order via the mobile or web frontend, our application backend receives the order details in JSON format. A sample order looks as follows:

```
{
  "size": "medium",
  "type": "margherita",
  "crust": "regular",
  "sauce": "marinara",
  "toppings": {
    "cheese": [
      {"mozzarella": "regular"}
    ],
    "veggies": [
      {"tomato": "light"}
    ]
  }
}
```

This particular order is for a medium-sized margherita pizza on a regular crust with standard toppings. Each topping belongs to one of the predefined categories, such as `toppings.cheese` or `toppings.veggies`. These categories store arrays of specific ingredients with their required amounts, such as a regular amount of mozzarella cheese (`"mozzarella": "regular"`) and light tomatoes (`"tomato": "light"`).

The reason toppings categories use arrays to store ingredients is that a recipe can include multiple kinds of cheese or veggies. For instance, another customer might order a three-cheese pizza that comes with several types of cheese:

```
{
  "size": "large",
  "type": "three cheese",
  "crust": "thin",
  "sauce": "marinara",
  "toppings": {
    "cheese": [
      {"cheddar": "regular"},
      {"mozzarella": "extra"},
      {"parmesan": "light"}
    ]
  }
}
```

Now, assume we want to store JSON objects with pizza order details as is in the database, letting the application layer define the JSON structure. With this approach, the customer selects and customizes a pizza via the UI, and the application layer prepares the order details in JSON format, which is then stored in Postgres.

The next question is how we should store these JSON objects in the database. Postgres provides several data types for storing JSON values, and we need to choose the one that best fits the application's requirements.

The first option is to store pizza order details in a column of the `text` data type. After all, any JSON object is a text string that can be stored in a character type like `text`. This option works well if the application only needs Postgres to persist JSON objects when orders are placed and retrieve those details when necessary. However, this option is not sufficient if the application needs to process JSON data at the database level by checking the values of particular JSON fields or performing other manipulations on the JSON structure.

If the structure of JSON objects needs to be queried rather than just persisted, Postgres provides two additional data types: `json` and `jsonb`. By using either of these data types, our application can run SQL queries in Postgres to access fields and traverse the structure of JSON pizza objects. However, `json` and `jsonb` store JSON data differently, with each type having its own trade-offs.

The `json` data type stores JSON objects in a textual format, preserving their original structure. Postgres doesn't perform any transformations on these objects, making it fast to store them in the database and returning the objects back to the application in their original form. However, every time an application needs to access a field in the `json` column, Postgres needs to parse the stored object, which increases the latency of read requests.

In contrast, the `jsonb` data type stores JSON objects in an internal binary format. Postgres parses and transforms these objects when they are written to the database. During this transformation, the database removes whitespace and duplicate field keys and may reorder field keys to improve search performance. Because of this conversion to the binary form, write performance can be slightly slower, and the application might retrieve a JSON object with a modified structure (for example, no whitespace or duplicates). However, search performance on `jsonb` columns is faster because the data is stored in an efficient binary format, and search operations can be further optimized with specialized generalized inverted index (GIN) indexes. Plus, Postgres provides a richer set of functions and operators for the `jsonb` type, allowing more advanced data processing.

Overall, the `jsonb` data type is the recommended default for storing and processing JSON data in Postgres, unless you have a specific use case that requires preserving the order of keys in the original JSON objects. Because our application for the pizza chain doesn't require Postgres to preserve the key order in JSON objects containing pizza order details, we will use the `jsonb` data type to store the orders in the database and support fast searches over the JSON structure.

5.2 Loading the pizza order dataset

Next, let's explore a sample dataset that stores orders for the pizza chain. The dataset consists of a single table with the following structure, which is more than enough for our experiments with JSON objects in Postgres:

```
pizzeria.order_items (
    order_id INT NOT NULL,
    order_item_id INT NOT NULL,
    pizza JSONB NOT NULL,
    price numeric(5,2) NOT NULL,
    PRIMARY KEY (order_id, order_item_id)
);
```

The table columns serve the following purposes:

- The `order_id` column serves as the unique identifier for a customer order.
- `order_item_id` is the unique identifier of a particular pizza item. A customer can order multiple pizzas in the same order, with each pizza having its own unique ID. The `PRIMARY KEY (order_id, order_item_id)` ensures the uniqueness of each pizza item in the order.
- The `pizza` column holds JSON objects with pizza order details, as discussed in the previous section. These JSON objects are stored using the `jsonb` data type.
- The `price` column stores the amount the customer paid for this particular item in the order.

NOTE If you'd like to gain practical experience while reading the chapter, connect to your Postgres instance started in chapter 1 using the `docker exec -it postgres psql -U postgres` command.

Follow these steps to preload the dataset into the Postgres instance in Docker:

1. Clone the book's repository with listings and sample data:

   ```
   git clone https://github.com/dmagda/just-use-postgres-book
   ```

2. Copy the pizza order dataset to your Postgres container:

   ```
   cd just-use-postgres-book/
   docker cp data/pizzeria/. postgres:/home/.
   ```

3. Preload the dataset by connecting to the container and using the `\i` meta-command of psql to apply the copied SQL scripts:

   ```
   docker exec -it postgres psql -U postgres -c "\i /home/pizzeria_ddl.sql"
   docker exec -it postgres psql -U postgres -c "\i /home/pizzeria_data.sql"
   ```

The table will be created under the `pizzeria` schema, and you can confirm this by connecting to Postgres and executing the `\dt pizzeria.*` command:

```
docker exec -it postgres psql -U postgres
\dt pizzeria.*
```

The output will be as follows:

```
             List of relations
  Schema  |    Name     | Type  |  Owner
----------+-------------+-------+----------
 pizzeria | order_items | table | postgres
(1 row)
```

Next, confirm that there are slightly more than 2,900 orders in the table:

```
SELECT count(*) FROM pizzeria.order_items;

 count
-------
  2938
(1 row)
```

Finally, get one of the orders to view its details stored in the `pizza` column of the `jsonb` type:

```
SELECT jsonb_pretty(pizza) FROM pizzeria.order_items
WHERE order_id = 30 and order_item_id = 2;

             jsonb_pretty
---------------------------------------
 {                                    +
     "size": "small",                 +
     "type": "custom",                +
     "crust": "regular",              +
     "sauce": "marinara",             +
     "toppings": {                    +
         "meats": [                   +
             {                        +
                 "bacon": "extra"     +
             }                        +
         ],                           +
         "cheese": [                  +
             {                        +
                 "cheddar": "extra"   +
             }                        +
         ],                           +
         "veggies": [                 +
             {                        +
                 "mushroom": "regular"+
             }                        +
         ]                            +
     }                                +
 }
(1 row)
```

jsonb_pretty() function

Note that the `jsonb_pretty(jsonb)`function outputs the data in a more readable format in a psql session. However, there is no need to use this function when returning a JSON object to the application layer. For example, here's how the query output appears without using the function:

```
SELECT pizza FROM pizzeria.order_items
WHERE order_id = 30 and order_item_id = 2;
   pizza
---------------------------------------------------------------------
 {"size": "small", "type": "custom", "crust": "regular",
 "sauce": "marinara", "toppings": {"meats": [{"bacon": "extra"}],
 "cheese": [{"cheddar": "extra"}], "veggies": [{"mushroom": "regular"}]}}
(1 row)
```

With the dataset in place, let's move forward and explore how to query, modify, and index JSON data in Postgres.

5.3 JSON in Postgres: Striking the balance

Even though Postgres provides full-fledged support for JSON, you should avoid storing all application data in JSON-specific data types as you would in a pure document database. Postgres offers a broad range of data types that allow you to strike the right balance between the flexibility of JSON and the efficiency of classic relational types (`int`, `text`, `date`, and so on), for which the database has many built-in optimizations.

Overall, you should consider using JSON when

- Data is static or updated infrequently (for example, configuration settings, metadata, customer preferences, or session history).
- Data is sparse, which is characterized by a significant presence of zeros, nulls, or placeholders (for example, feature flags, user preferences, or configuration settings with dozens of options to choose from).
- Schema flexibility is required, or normalization is hard to achieve (for example, third-party API responses, telemetry events, or semi-structured text documents).

If performance is critical for operations involving updates, joins, aggregations, or filtering, then consider using relational data types with a fully normalized model or a hybrid approach.

With a *hybrid approach*, JSON is used alongside relational data. For instance, our `pizzeria.order_items` table stores pizza order details in JSON format. These details represent customer preferences that are unlikely to change once the order is in the system. At the same time, we store `order_id`, `order_item_id`, and `price` as standalone columns to enable more efficient searches on frequently accessed data and enforce data integrity with the `PRIMARY KEY` and `NOT NULL` constraints.

In a traditional normalized model, each part of the pizza—type, crust, sauce, size, and toppings—would be stored in separate columns or even separate tables. Without putting too much thought into column data types and data integrity checks, here's what a rough sample structure might look like if we replaced the `order_items.pizza` column that presently stores complete pizza recipes in the JSON format with `order_items.pizza_id` and dedicated tables storing the recipes information:

```
-- Stores ordered pizza items with core attributes such as type and crust
-- The id is referenced from other tables
CREATE TABLE pizzeria.pizzas (
    id SERIAL PRIMARY KEY,
    type TEXT NOT NULL,
    crust TEXT NOT NULL,
    sauce TEXT NOT NULL,
    size TEXT NOT NULL
);

-- Links pizza items to particular customer orders
CREATE TABLE pizzeria.order_items (
    order_id INT NOT NULL,
    order_item_id INT NOT NULL,
    pizza_id INT REFERENCES pizzeria.pizzas(id),
    price NUMERIC(5,2) NOT NULL,
    PRIMARY KEY (order_id, order_item_id)
);

-- Stores cheese toppings and their amounts for each pizza
CREATE TABLE pizzeria.pizza_cheeses (
    pizza_id INT REFERENCES pizzeria.pizzas(id),
    cheese TEXT NOT NULL,
    amount TEXT NOT NULL,
    PRIMARY KEY (pizza_id, cheese)
);

-- Stores veggie toppings and their amounts for each pizza
CREATE TABLE pizzeria.pizza_veggies (
    pizza_id INT REFERENCES pizzeria.pizzas(id),
    veggie TEXT NOT NULL,
    amount TEXT NOT NULL,
    PRIMARY KEY (pizza_id, veggie)
);

-- Stores meat toppings and their amounts for each pizza
CREATE TABLE pizzeria.pizza_meats (
    pizza_id INT REFERENCES pizzeria.pizzas(id),
    meat TEXT NOT NULL,
    amount TEXT NOT NULL,
    PRIMARY KEY (pizza_id, meat)
);
```

With this normalized approach, the data model looks more complex, which would, at a minimum, lead to the following downsides for the discussed usage scenario:

- *Write overhead*—Our application would need to update several tables transactionally when adding a new pizza to the order. Moreover, each topping is now stored as a standalone table record. If a customer orders a pizza with two types of meats, three types of cheeses, and two types of veggies, the application will need to insert seven records across the respective tables transactionally.
- *Read overhead*—The application would need to join data across several tables to re-create a complete pizza recipe before showing it to the kitchen crew or the customer.
- *Transformation overhead*—Because our application frontend works with pizza recipes in JSON format—making it easier to display and process orders—we will need to transform data between JSON and the normalized model every time the application creates or retrieves a pizza order from the database.

Because pizza recipes represent static data that might be updated occasionally—and given the downsides of the normalized model for this specific scenario—we use a hybrid approach in this book that helps to strike the balance in using JSON data in Postgres.

5.4 Querying JSON data

If our application needs to display order details to the kitchen crew or the customer who placed the order, it can simply retrieve the entire JSON object from Postgres and display it. For example, the application only needs to pass the `order_id` and request the `pizza` column, which contains the JSON data with the order details:

```
SELECT pizza
FROM pizzeria.order_items
WHERE order_id = 100;
```

This particular order includes three pizzas, and the output looks like this:

```
                                     pizza
--------------------------------------------------------------------------
 {"size": "small", "type": "margherita", "crust": "thin",
  "sauce": "marinara", "toppings": {"cheese": [{"mozzarella": "regular"}],
  "veggies": [{"tomato": "light"}]}}

 {"size": "small", "type": "custom", "crust": "gluten_free",
  "sauce": "pesto", "toppings": {"meats": [{"bacon": "light"}],
  "cheese": [{"mozzarella": "regular"}], "veggies": [{"onion": "light"}]}}

 {"size": "extra_large", "type": "pepperoni", "crust": "thin",
  "sauce": "marinara", "toppings": {"meats": [{"salami": "regular"}],
  "cheese": [{"mozzarella": "regular"}]}}
(3 rows)
```

But what if the application doesn't need the entire JSON object? Instead, it might require specific fields, like the pizza crust type or a list of ingredients. Additionally, the

application might need to retrieve a list of pizza orders that meet specific criteria—for example, the next five pizza orders requiring mozzarella cheese. In such cases, we can use Postgres operators and functions to easily query JSON fields and navigate their structure.

In this section, we'll explore some of the basic JSON operators and functions available in Postgres in the following order:

1. We'll start with the `->` and `->>` operators, which extract fields from a JSON object. The `->` operator returns fields in JSON format, and `->>` converts them to text for further processing.
2. We'll examine the `@>` and `?` operators. The `@>` operator checks whether a JSON object contains another JSON object, and the `?` operator verifies the existence of a specific key in the object.
3. We'll dive into Postgres JSON path expressions, which allow us to retrieve fields from a JSON object that meet specified path rules.

TIP Refer to the Postgres documentation for a full list of operators and functions that the database supports for the `json` and `jsonb` data types: https://www.postgresql.org/docs/current/functions-json.html

5.4.1 Extracting fields with the -> and ->> operators

The `->` and `->>` operators allow us to extract fields from a JSON object by specifying a field key. The main difference between these operators is that `->` returns the extracted field in JSON format and `->>` converts the field into the text data type.

For example, suppose our application needs to retrieve the crust type and size of a pizza for a specific order. This information can be easily queried from Postgres using either of these operators.

Listing 5.1 Extracting JSON object fields

```
SELECT
    order_id, order_item_id,
    pizza->'size' as pizza_size,
    pizza->>'crust' as pizza_crust
FROM pizzeria.order_items
WHERE order_id = 100;
```

Note that both operators require the key field names to be enclosed in single quotes (`'`), such as `pizza->'size'` or `pizza->>'crust'`. The output for the query looks as follows:

```
 order_id | order_item_id |  pizza_size   | pizza_crust
----------+---------------+---------------+-------------
      100 |             1 | "small"       | thin
      100 |             2 | "small"       | gluten_free
      100 |             3 | "extra_large" | thin
(3 rows)
```

Even though both the `pizza_size` and `pizza_crust` columns return text data, their output formats differ for the following reason:

- The `pizza_size` value was extracted using the `->` operator, which returns the field in JSON format. According to the JSON specification, text values must be enclosed in double quotes (`"`).
- The `pizza_crust` value was extracted using the `->>` operator, which converts the field into Postgres's text data type. In this format, text values are not enclosed in double quotes.

The difference in output format between the `->` and `->>` operators needs to be considered when the application processes the result or filters data on the Postgres side. For example, the following listing shows how to correctly filter data when using these operators in the `WHERE` clause of a query.

Listing 5.2 Filtering data with the -> and ->> operators

```
SELECT
    order_id, order_item_id,
    pizza->'size' as pizza_size,
    pizza->>'crust' as pizza_crust
FROM pizzeria.order_items
WHERE
    order_id = 100 AND
    pizza->'size' = '"small"' AND pizza->>'crust' = 'gluten_free';
```

The `pizza->'size'` value is compared to the `"small"` JSON object with double quotes, and the `pizza->>'crust'` value is checked against a regular `gluten_free` text value. If we remove the double quotes from the `pizza->'size'` condition, changing it to `pizza->'size' = 'small'`, Postgres will return the following error:

```
SELECT
    order_id, order_item_id,
    pizza->'size' as pizza_size,
    pizza->>'crust' as pizza_crust
FROM pizzeria.order_items
WHERE
    order_id = 100 AND
    pizza->'size' = 'small' AND pizza->>'crust' = 'gluten_free';

DETAIL:  Token "small" is invalid.
CONTEXT:  JSON data, line 1: small
```

The `->` and `->>` operators can also be chained to access nested JSON fields. For example, suppose our application needs to query all the vegetables a customer requested as pizza toppings. The next listing demonstrates how to extract this data by chaining the `->` operator.

Listing 5.3 Chaining the -> operator

```
SELECT
    order_id, order_item_id,
    pizza->'toppings'->'veggies' as veggies_toppings
FROM pizzeria.order_items
WHERE order_id = 100;
```

The query produces the following result:

```
 order_id | order_item_id |   veggies_toppings
----------+---------------+-----------------------
      100 |             1 | [{"tomato": "light"}]
      100 |             2 | [{"onion": "light"}]
      100 |             3 |
(3 rows)
```

As shown in the output, the kitchen crew needs to add tomatoes to the first pizza in the order and onions to the second pizza. The `veggies_toppings` column is empty for the third pizza because the customer chose not to add any vegetables for that pizza.

The previous output also shows that the toppings are returned as an array of ingredients enclosed in `[]` brackets. If we want to extract a specific object from the JSON array, we can do so by passing the object's position within the array to the `->` or `->>` operator. The following query demonstrates how to return the first (and only, for this particular order item) veggie ingredient, which is at position 0 in the array:

```
SELECT
    order_id, order_item_id,
    pizza->'toppings'->'veggies'->0 as veggies_toppings
FROM pizzeria.order_items
WHERE order_id = 100;
```

This time, the `veggies_toppings` column value is no longer enclosed in `[]` brackets because the query returns only the first topping from the array of ingredients:

```
 order_id | order_item_id |  veggies_toppings
----------+---------------+---------------------
      100 |             1 | {"tomato": "light"}
      100 |             2 | {"onion": "light"}
      100 |             3 |
(3 rows)
```

Finally, if needed, it's possible to use both the `->` and `->>` operators in a chain. For example, the next query continues using the `->` operator to navigate to the first veggie topping in the array but then applies the `->>` operator to retrieve the amount of onions required:

```
SELECT
    order_id, order_item_id,
```

```
    pizza->'toppings'->'veggies'->0->>'onion' as onions_amount
FROM pizzeria.order_items
WHERE order_id = 100;
```

The output for the query looks as follows:

```
 order_id | order_item_id | onions_amount
----------+---------------+---------------
      100 |             1 |
      100 |             2 | light
      100 |             3 |
(3 rows)
```

The kitchen crew needs to add a light amount of onions to the pizza with `order_item_id = 2`. The other pizzas in the order don't include this ingredient in their list.

5.4.2 Using the ? operator to check for the presence of a key

The `?` operator is useful for checking whether a specific key exists in a JSON object's structure. This operator is supported only for objects stored in the `jsonb` data type in Postgres.

Suppose the chef is planning to go to the storage room to get some ingredients and wants to check which meats are needed for the orders already in the queue. The application can easily provide this information by executing the following query against Postgres.

Listing 5.4 Checking for the existence of a field key

```
SELECT
    order_id, order_item_id,
    pizza->'toppings'->'meats' as meats
FROM pizzeria.order_items
WHERE pizza->'toppings' ? 'meats'
ORDER BY order_id LIMIT 5;
```

The query uses the `?` operator to return information only for orders that include `meats` in the `toppings` list. It produces the following output, showing the meats needed for the next five orders:

```
 order_id | order_item_id |          meats
----------+---------------+--------------------------
        1 |             1 | [{"salami": "regular"}]
        1 |             2 | [{"chicken": "regular"}]
        2 |             1 | [{"salami": "regular"}]
        3 |             1 | [{"salami": "regular"}]
        5 |             1 | [{"salami": "regular"}]
(5 rows)
```

Now imagine that the chef returns from storage and informs the kitchen crew that they are about to run out of Italian sausage. The crew needs to contact customers who have

already placed pizza orders with this type of sausage to discuss alternative options. The application can easily generate a list of such orders by executing the next query.

Listing 5.5 Checking the existence of a field key in an array

```
SELECT
    order_id, order_item_id,
    pizza->'toppings'->'meats' AS meats
FROM pizzeria.order_items
WHERE EXISTS (
    SELECT 1
    FROM jsonb_array_elements(pizza->'toppings'->'meats') AS meats
    WHERE meats ? 'sausage'
)
ORDER BY order_id LIMIT 5;
```

Because the `meats` array can contain multiple ingredients, the query uses a subquery that works as follows:

- The `jsonb_array_elements(pizza->'toppings'->'meats')` function expands the meats array into individual JSON objects.
- The `WHERE meat ? 'sausage'` clause filters and includes only those expanded objects that have sausage in the ingredients list.
- The subquery doesn't need to return any specific values—just a row—so it uses `SELECT 1` to indicate to the outer query that at least one object in the `meats` array satisfies the condition.

> **NOTE** We'll optimize and improve the readability of the query from listing 5.5 in the following sections by replacing the subquery with a JSON path expression.

The output of the query from listing 5.5 looks as follows, helping the kitchen crew reach out to the appropriate customers:

```
 order_id | order_item_id |          meats
----------+---------------+------------------------
       20 |             5 | [{"sausage": "light"}]
       26 |             5 | [{"sausage": "extra"}]
       34 |             2 | [{"sausage": "extra"}]
       37 |             2 | [{"sausage": "extra"}]
       47 |             1 | [{"sausage": "extra"}]
(5 rows)
```

5.4.3 Comparing objects with the @> operator

The containment operator `@>` is another feature supported only for JSON objects stored in the `jsonb` data type of Postgres. This operator allows us to check whether one JSON object contains another within its structure.

Imagine that our application provides a business insights service used by the pizza chain's management and marketing teams to monitor business-related statistics. This service features a dashboard where the teams can analyze order history to better understand customer preferences, measure the popularity of specific pizza items, and evaluate the effectiveness of marketing campaigns. The dashboard allows the teams to generate reports by slicing and dicing the order data with flexible filtering capabilities.

Suppose the marketing team launched a promo campaign to boost sales of gluten-free crusts. Now a marketing manager wants to generate a report showing the total number of pizza orders with that type of crust. The dashboard logic can quickly retrieve this data by executing the following query against the database.

Listing 5.6 Filtering data with the @> operator

```
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"crust": "gluten_free"}';
```

The query uses the `@>` operator to find and count only those pizza orders that contain the `{"crust": "gluten_free"}` object. The containment operator checks whether the JSON object on the left of the operator (`pizza`) contains the JSON object on the right (`{"crust": "gluten_free"}`).

Based on the query output, customers have placed more than 700 orders with gluten-free crust since the campaign launched:

```
 count
-------
   749
(1 row)
```

Next, the marketing manager becomes curious and wants to see how many of those orders were for custom pizza recipes. The manager adds a filter for custom pizzas, and the dashboard executes a slightly modified query in Postgres:

```
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"crust": "gluten_free", "type": "custom"}';
```

The query continues to use the containment operator to filter the data, but this time it returns only those pizza orders that contain both the `"crust": "gluten_free"` and `"type": "custom"` fields at the top level of the `pizza` object structure. Once the query is executed, the dashboard shows the following data:

```
 count
-------
   148
(1 row)
```

Finally, the manager wants to see how many of those orders included an extra amount of tomatoes. The manager adjusts the dashboard's filter, which executes the following query over the database:

```
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"crust": "gluten_free",
"type": "custom", "toppings": {"veggies": [{"tomato": "extra"}]}}';
```

This updated query verifies that the `pizza` object also contains `{"tomato": "extra"}` as a topping. Because all veggie ingredients are stored in the `veggies` JSON array inside the `toppings` object, the query instructs the `@>` operator to search for the `tomato` ingredient in the `"toppings": {"veggies": [{"tomato": "extra"}]}` object hierarchy. The output for this query looks as follows:

```
 count
-------
     7
(1 row)
```

> **Using the -> and @> operators together**
>
> We can make the last query more readable by combining the `->` and `@>` operators as follows:
>
> ```
> SELECT count(*)
> FROM pizzeria.order_items
> WHERE pizza @> '{"crust": "gluten_free", "type": "custom"}' AND
>      pizza->'toppings'->'veggies' @> '[{"tomato": "extra"}]';
> ```
>
> The chain of `->` operators allows us to navigate to the `veggies` array in the `pizza` object, and the `@>` operator is then used to check whether the array contains the `{"tomato": "extra"}` object.

The marketing manager is eager to share the campaign results with their colleagues. Meanwhile, we've learned how to use the containment operator to compare JSON objects of varying complexity.

5.4.4 Using JSON path expressions

In addition to the basic JSON functions and operators covered in the previous sections, Postgres offers a powerful path engine capable of executing expressions written in the SQL/JSON path language. These expressions are particularly useful when we need to retrieve items from multilevel JSON structures or arrays of JSON objects. A simple path expression can eliminate the need for subqueries or other SQL constructs that might make queries less readable and harder to maintain.

For example, in listing 5.5, we used a subquery to find orders that included sausage as a topping. That subquery can be easily replaced with the following path expression, allowing us to achieve the same result in a more compact form:

```
SELECT
    order_id, order_item_id,
    pizza->'toppings'->'meats' AS meats
FROM pizzeria.order_items
WHERE jsonb_path_exists(pizza, '$.toppings.meats[*] ? (exists(@.sausage))')
ORDER BY order_id LIMIT 5;
```

At this point, you might not fully understand how this expression works or the purpose of some of its operators, such as `$` or `@`. Don't worry—everything will become crystal clear by the end of this section. We'll build up gradually, starting with simpler use cases.

Imagine that our application needs to send a daily report to the pizza chain's management team, sharing valuable end-of-day business data. The application relies heavily on Postgres JSON path expressions to provide insights about the most popular pizza types, favorite ingredients, and more. The first section of the report highlights the most popular types of pizzas among customers, which are populated using the following query.

Listing 5.7 Accessing JSON fields with path expressions

```
SELECT
    count(*) as total_cnt,
    jsonb_path_query(pizza,'$.type') as pizza_type
FROM pizzeria.order_items
GROUP BY pizza_type ORDER BY total_cnt DESC;
```

The query uses the `jsonb_path_query` function to extract the pizza type for each order and then groups the retrieved data and calculates the total for each group. The path expression is evaluated and executed as follows:

- The `jsonb_path_query` function is applied to the `pizza` column of a row, with the path expression evaluated and executed from left to right.
- The path expression always starts with the `$` operator, which refers to the JSON object being queried—in this case, the `pizza` object.
- The `$` operator is followed by the accessor operator `.key`, which in this query retrieves the value referenced by the `.type` field of the `pizza` object.

The output of the query looks as follows, listing the most popular pizzas first:

```
 total_cnt |   pizza_type
-----------+----------------
       646 | "pepperoni"
       585 | "three cheese"
       582 | "veggie"
       563 | "custom"
       562 | "margherita"
(5 rows)
```

JSON path expressions are particularly useful when querying JSON arrays. For instance, the next section of the business report provides statistics on the most popular types of cheese.

Listing 5.8 Querying JSON arrays with path expressions

```
SELECT
    count(*) as total_cnt,
    jsonb_object_keys(
        jsonb_path_query(pizza, '$.toppings.cheese[*]')
    ) as cheese_topping
FROM pizzeria.order_items
GROUP BY cheese_topping ORDER BY total_cnt DESC;
```

The query evaluates and executes the path expression as follows:

- The `jsonb_path_query` function applies the `$.toppings.cheese[*]` expression to the JSON object stored in the `pizza` column.
- The `$.toppings.cheese` part of the expression navigates to the `cheese` array, and the `[*]` operator retrieves all objects stored in the array.
- The `jsonb_object_keys` function extracts the keys from the JSON objects returned by the `jsonb_path_query` function call. These keys represent cheese names, such as cheddar or Parmesan.

The query produces the following statistics, which are added to the report, showing that mozzarella is the most popular cheese type in recent orders:

```
 total_cnt | cheese_topping
-----------+----------------
      2575 | mozzarella
       771 | cheddar
       762 | parmesan
(3 rows)
```

A path expression can also include a filter, which functions similarly to a `WHERE` clause in SQL. The filter starts with the `?` operator, followed by the filter condition enclosed in parentheses.

Suppose the next section of the report displays the total number of pizza types that used Parmesan cheese as an ingredient.

Listing 5.9 Using filters in path expressions

```
SELECT
    count(*) AS total_cnt,
    pizza->'type' as pizza_type
FROM pizzeria.order_items
WHERE jsonb_path_exists(pizza,
        '$.toppings.cheese[*] ? (exists(@.parmesan))')
GROUP BY pizza_type
ORDER BY total_cnt DESC;
```

The `jsonb_path_exists` function checks whether the path expression evaluates to any value when applied to the `pizza` column. If the function evaluates to `true`, the row is included in the query result. The path expression uses a filter and is executed as follows:

- The `$.toppings.cheese[*]` part of the expression retrieves all types of cheese used for the current `pizza` object.
- The `?` operator applies the `(exists(@.parmesan))` condition to check whether the retrieved cheese types include `parmesan`.
- The `@` variable represents the current evaluated cheese type object from the array, and `.parmesan` accesses the corresponding field in that object. If the `parmesan` field exists, the `(exists(@.parmesan))` condition evaluates to `true`.

The query produces the following output, showing that Parmesan is used in only two types of pizzas:

```
 total_cnt |   pizza_type
-----------+----------------
       585 | "three cheese"
       177 | "custom"
(2 rows)
```

Finally, path expressions support chaining multiple filter expressions, where each subsequent filter is applied to the result of the previous one. This capability enables more advanced filtering logic in the SQL/JSON language. For example, one of the final sections of our business report provides statistics on the popularity of Parmesan cheese in custom pizza types.

Listing 5.10 Chaining multiple filter expressions

```
SELECT count(*)
FROM pizzeria.order_items
WHERE jsonb_path_exists(
    pizza,
    '$ ? (@.type == "custom") .toppings.cheese[*].parmesan ? (@ == "extra")'
);
```

The path expression is evaluated from left to right as follows:

- `$` refers to the `pizza` object of the current table row, which stores order details in JSON format.
- The `? (@.type == "custom")` condition checks whether the `type` field of the current `pizza` object is set to `custom`.
- If the pizza is indeed of the custom type, then the `.toppings.cheese[*].parmesan` part of the expression navigates to the array of `cheese` objects and retrieves the `parmesan` object (if present).
- The final `? (@ == "extra")` condition checks whether the retrieved `parmesan` object has the volume amount set to `extra`.

The result of this query appears as follows, showing how often customers request an extra amount of Parmesan on their pizzas:

```
 count
-------
    64
(1 row)
```

The business owners are glad to see the results of the current report. And we've learned how to use JSON path expressions in Postgres to query and access JSON data of various complexity in a compact form.

5.5 Modifying JSON data

So far, we've learned how to query JSON data using Postgres's built-in capabilities. Now, let's explore how to modify JSON objects already stored in the database.

The easiest way to modify an existing JSON object is to query it from Postgres to the application layer, perform the required changes, and then store the updated version back in the database using an `UPDATE` statement. For instance, the application can fetch a particular `pizza` recipe by passing its `order_id` as the first parameter and `order_item_id` as the second parameter:

```
SELECT pizza
FROM pizzeria.order_items
WHERE order_id = $1 and order_item_id = $2;
```

Then the contents of the `pizza` JSON object are modified by the application and written back to the database using the following statement:

```
UPDATE pizzeria.order_items
SET pizza = new_pizza_order_json
WHERE order_id = $1 and order_item_id = $2;
```

However, the easiest way isn't always the most efficient. If a JSON object has a complex structure with many fields and nested objects, but we only need to modify a single field, it's often more practical to update the field directly in the database without transferring the entire object between the application and database layers.

Postgres provides the `jsonb_set` function for updating existing JSON objects and the `#-` operator for removing specific data from a JSON structure. Let's see how these features can be used in our pizza chain application.

Imagine that one of the pizza chain's customers has placed an order, which can be retrieved with the following query:

```
SELECT jsonb_pretty(pizza)
FROM pizzeria.order_items
WHERE order_id = 20 and order_item_id = 5;
```

Currently, the order details look as follows:

```
{
     "size": "extra_large",
     "type": "custom",
     "crust": "stuffed",
     "sauce": "alfredo",
     "toppings": {
         "meats": [
             {
                 "sausage": "light"
             }
         ],
         "cheese": [
             {
                 "parmesan": "regular"
             }
         ],
         "veggies": [
             {
                 "tomato": "light"
             }
         ]
     }
 }
```

Five minutes later, the customer decides to change a few ingredients and goes back to the app. Luckily, the kitchen crew has not started working on the order yet, so the application allows the customer to modify the pizza's style and toppings.

The customer changes the crust type from stuffed to regular and clicks the Save button. The application executes the following query save the changes to the database.

Listing 5.11 Updating JSON fields with `jsonb_set`

```
UPDATE pizzeria.order_items
SET pizza = jsonb_set(pizza,'{crust}', '"regular"', false)
WHERE order_id = 20 and order_item_id = 5;
```

The query updates the `pizza` column with the new version of the pizza order, generated using the `jsonb_set` function. This function accepts four arguments, each serving a specific purpose:

- *The original JSON object*—The first argument is the JSON object to be updated. In our case, it is the value stored in the `pizza` column.
- *The path to the target field*—The second argument specifies the path to the JSON field or nested object that needs modification. The path is written in the `{a, b, c}` format, where `c` is the target field located in the `b` object, which is itself nested under the `a` object. In our query, we are updating the `crust` field, so the path is simply `{crust}`.

- *The new value*—The third argument is the new value to set for the target field. In our case, we set the `crust` field to `"regular"`, which is a JSON string and must be enclosed in double quotes (").
- *Behavior when the field is missing*—The fourth argument determines what to do if the target field is missing in the JSON structure. If this argument is set to `true` and the field is missing, the field will be added to the JSON object. In our query, we pass `false` because the `crust` field always exists in pizza orders.

Right after saving the changes, the customer glances at the veggie ingredients and notices that they mistakenly requested a "light" amount of tomatoes. The customer edits the order again, this time asking for extra tomatoes and some spinach. The application executes the next query in Postgres.

Listing 5.12 Updating JSON arrays with `jsonb_set`

```
UPDATE pizzeria.order_items
SET pizza = jsonb_set(
    pizza,
    '{toppings,veggies}',
   '[{"tomato":"extra"}, {"spinach":"regular"}]',
   false
)
WHERE order_id = 20 and order_item_id = 5;
```

This time, the `jsonb_set` function navigates to the `veggies` array using the `{toppings, veggies}` path and updates the `veggie` ingredients to `[{"tomato": "extra"}, {"spinach": "regular"}]`.

> **TIP** If you need to update a specific object in a JSON array and you know its index, you can include the index in the path. For example, the `{toppings, veggies, 0, tomato}` path allows the `jsonb_set` function to locate the `tomato` object, which is the first element of the `veggies` array (hence, the index is 0), and then set it to a new value, such as `"extra"`.

The customer is about to close the pizza app and put their phone aside when their teenager notices that they added sausage to the pizza—again. The teen suggests removing meats from the order, and the dad makes one final change, removing this topping. The application executes the query against the database.

Listing 5.13 Deleting JSON fields with the `#-` operator

```
UPDATE pizzeria.order_items
SET pizza = pizza #- '{toppings,meats}'
WHERE order_id = 20 AND order_item_id = 5;
```

The query uses the `#-` operator to remove the `meats` topping located under the `{toppings, meats}` path from the JSON object stored in the `pizza` column.

After making so many changes, the customer hopes the kitchen crew has been notified about the changes in time. Once the crew starts working on the customer's order, the order management system retrieves the order details from Postgres with the following query:

```
SELECT jsonb_pretty(pizza)
FROM pizzeria.order_items
WHERE order_id = 20 and order_item_id = 5;
```

The kitchen crew now sees the up-to-date order with all the modifications made by the customer—regular crust, no sausage, extra tomato, and spinach:

```
{
    "size": "extra_large",
    "type": "custom",
    "crust": "regular",
    "sauce": "alfredo",
    "toppings": {
        "cheese": [
            {
                "parmesan": "regular"
            }
        ],
        "veggies": [
            {
                "tomato": "extra"
            },
            {
                "spinach": "regular"
            }
        ]
    }
}
```

TIP We've explored how to query and modify existing JSON data using Postgres's built-in capabilities. In addition, Postgres lets you create JSON objects from scratch using `json_build_object`, `row_to_json`, `jsonb_agg`, and other JSON creation functions. These functions are useful when you want to generate JSON data at the database layer instead of in the application. You can learn more about JSON creation functions in the official documentation: https://www.postgresql.org/docs/current/functions-json.html.

5.6 Indexing JSON data

Let's now learn how to index JSON data in Postgres to speed up searches with the JSON operators and functions supported by the database.

5.6.1 Using an expression index with a B-tree

Imagine that several components of our application for the pizza chain need to calculate the total number of pizza orders of a particular type. For instance, the following query returns the total number of pizzas of the custom type:

```
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza ->> 'type' = 'custom';
```

The query reports that customers ordered more than 560 custom pizzas. But this raises the next question. How efficient is the execution of this query on the database? Let's add the `EXPLAIN (analyze, costs off)` statement and check the execution plan Postgres uses to retrieve the data:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza ->> 'type' = 'custom';
```

According to the generated plan, Postgres performs a full table scan (`Seq Scan`) over the entire table of pizza orders:

```
                                QUERY PLAN
---------------------------------------------------------------------------
 Aggregate (actual time=1.135..1.136 rows=1 loops=1)
   ->  Seq Scan on order_items (actual time=0.034..1.062 rows=563 loops=1)
         Filter: ((pizza ->> 'type'::text) = 'custom'::text)
         Rows Removed by Filter: 2375
 Planning Time: 0.174 ms
 Execution Time: 1.185 ms
(6 rows)
```

This particular query can easily be optimized with an expression index on the data returned by the `pizza ->> 'type'` statement. The following query creates such an index.

Listing 5.14 Creating an expression index on a JSON field

```
CREATE INDEX idx_pizza_type
ON pizzeria.order_items ((pizza ->> 'type'));
```

Once the index is created, we can check the execution plan for the query again:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza ->> 'type' = 'custom';
```

Now Postgres performs a `Bitmap Index Scan` over the JSON data, optimizing the search to be nearly four times faster (1.185 ms with a `Seq Scan` versus 0.376 ms with a `Bitmap Index Scan`):

```
                               QUERY PLAN
-----------------------------------------------------------------------
 Aggregate (actual time=0.299..0.300 rows=1 loops=1)
   ->  Bitmap Heap Scan on order_items
 ➥      (actual time=0.170..0.257 rows=563 loops=1)
         Recheck Cond: ((pizza ->> 'type'::text) = 'custom'::text)
         Heap Blocks: exact=101
         ->  Bitmap Index Scan on idx_pizza_type
 ➥            (actual time=0.068..0.068 rows=563 loops=1)
               Index Cond: ((pizza ->> 'type'::text) = 'custom'::text)
 Planning Time: 0.164 ms
 Execution Time: 0.376 ms
(8 rows)
```

However, a tradeoff of expression indexes is that they index the exact expression or function call specified in the `CREATE INDEX` statement. If a query deviates even slightly from the indexed expression, the database will no longer use the index. For example, if our pizza application changes its logic to use the `->` operator instead of `->>`, the `idx_pizza_type` index will no longer be used, as shown in the execution plan for the following query:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza -> 'type' = '"custom"';
```

The execution plan shows that Postgres defaults to a `Seq Scan`, iterating through the entire table:

```
                               QUERY PLAN
---------------------------------------------------------------------------
 Aggregate (actual time=2.039..2.040 rows=1 loops=1)
   ->  Seq Scan on order_items (actual time=0.040..1.991 rows=563 loops=1)
         Filter: ((pizza -> 'type'::text) = '"custom"'::jsonb)
         Rows Removed by Filter: 2375
 Planning Time: 0.134 ms
 Execution Time: 2.089 ms
(6 rows)
```

Although it's possible to establish rules at the application layer, asking the development team to always use the `pizza ->> 'type'` operator to take advantage of the index, what happens if the application needs to count the total number of pizzas of a particular size? Let's analyze the execution plan for the following query, which counts the total number of large pizzas using the `pizza ->> 'size' = 'large'` condition:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza ->> 'size' = 'large';
```

The execution plan for this query shows that the database performs a full table scan (`Seq Scan`) because there is no index for the `pizza ->> 'size'` expression:

```
                              QUERY PLAN
---------------------------------------------------------------------------
 Aggregate (actual time=1.234..1.235 rows=1 loops=1)
   ->  Seq Scan on order_items (actual time=0.031..1.181 rows=733 loops=1)
         Filter: ((pizza ->> 'size'::text) = 'large'::text)
         Rows Removed by Filter: 2205
 Planning Time: 0.126 ms
 Execution Time: 1.281 ms
(6 rows)
```

We can optimize this query by creating an index for the `pizza ->> 'size'` expression. However, this approach doesn't scale well if the application requires indexes on additional fields in the JSON structure. The more indexes Postgres has, the more time it takes to maintain and update them, which can affect the latency of certain queries and overall application performance.

But why do we even use expression indexes? Why can't we just create a single index for the entire JSON object stored in the `pizza` column of the table?

Currently, we create expression indexes for our JSON data because we rely on the B-tree data structure, which is used by Postgres by default to store and access indexed data internally. The next query confirms this by showing that our `idx_pizza_type` expression index is backed by the B-tree data structure.

Listing 5.15 Checking index details

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE schemaname = 'pizzeria'
  AND tablename = 'order_items'
  AND indexdef LIKE '%idx_pizza_type%';
```

The output for this query shows that Postgres used a B-tree for the expression index (`USING btree`):

```
   indexname    |                      indexdef
----------------+------------------------------------------------------------
 idx_pizza_type | CREATE INDEX idx_pizza_type
                  ON pizzeria.order_items
                  USING btree (((pizza ->> 'type'::text)))
(1 row)
```

Although a B-tree is an excellent general-purpose index type, it has specific limitations. If we attempted to use a B-tree to index the contents of the entire `pizza` column, the B-tree would index the JSON data as a single value. This means we wouldn't benefit from the index when applying `->>`, `@>`, or other JSON-specific operators to fields in our JSON pizza orders. Instead, the application would have to generate an exact JSON object—with all fields, values, and even whitespace—and compare it to the contents of the `pizza` column using the equality (`=`) operator. This approach would require creating and comparing entire JSON objects even if we only needed to retrieve pizzas of a certain type. That's why we use expression indexes with B-trees that let us easily compare the contents of specific fields in the JSON structure, like `pizza ->> 'type'`.

However, the B-tree isn't the only data structure in Postgres that determines how indexed data is stored and accessed internally by the database. Postgres also offers more specialized indexes that persist and handle data differently. One such index is GIN, which is much better suited for JSON data.

5.6.2 Using GIN indexes

By using GIN as the underlying data structure, an application can create a regular single-column index on JSON data to enable efficient searches across the nested JSON structure. Postgres supports two types of GIN indexes for JSON objects, each storing indexed data differently and supporting distinct operator classes for data access. The first type is the default GIN index, which extracts all the keys, values, and array elements from the original JSON object and adds them as distinct items to the index structure. The second type indexes only paths from the root of the JSON document down to each value and array element. Let's start with the default GIN index and later compare it to the one that indexes the paths of the JSON structure.

USING THE DEFAULT GIN INDEX

Imagine that we have a pizza order with the following details stored in the database:

```
{
 "size": "large",
 "type": "three cheese",
 "crust": "thin",
 "sauce": "marinara",
 "toppings": {
   "cheese": [
     {"cheddar": "regular"},
     {"mozzarella": "extra"},
     {"parmesan": "light"}
   ]
 }
}
```

When we create a default GIN index on the `pizza` column, which includes this order, the index will extract the following keys and values from the JSON object and store them as separate index items:

- *Keys*—`size`, `type`, `crust`, `sauce`, `toppings`, `cheese`, `cheddar`, `mozzarella`, `parmesan`
- *Values*—`large`, `three cheese`, `thin`, `marinara`, `regular`, `extra`, `light`

Each extracted index entry references the corresponding table row that stores the original JSON pizza order. Figure 5.1 shows a simple representation of a GIN index with a subset of the discussed index entries.

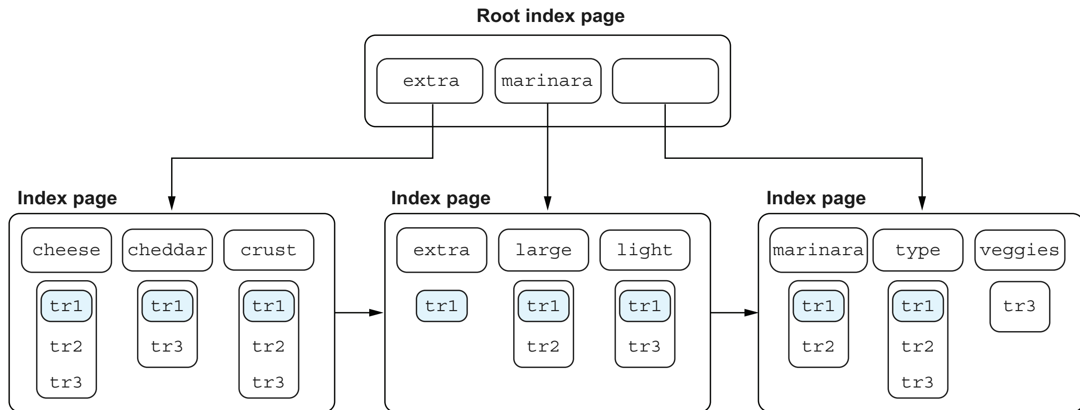


Figure 5.1 A sample default GIN index holding references to the table row `tr1` with our sample pizza order

The index entries are stored in ascending lexicographical (alphabetical) order and distributed across several index pages. The search starts with the root index page, which contains three index entries:

- `extra` points to an index page containing entries lexicographically less than `extra`.
- `marinara` points to another index page with entries that are lexicographically greater than or equal to `extra` but less than `marinara`.
- The blank key points to a third index page that holds all other entries greater than or equal to `marinara`.

The search continues with the second-level index pages, which store index entries that reference actual table rows. For example, the index entry for `cheese` references three table rows (`tr1`, `tr2`, and `tr3`), indicating that the `pizza` column in those rows includes `cheese` in its JSON structure.

If we imagine that `tr1` is a reference to our example pizza order, we can see that all the index entries (except `veggies`) reference that row. The `veggies` entry doesn't point to `tr1` because there are no vegetable ingredients in that particular order.

After the index is created, the application can use the default `jsonb_ops` operator class to benefit from the index when querying the nested JSON structure. This operator

class supports several key-exists operators, including `?`, the containment operator `@>`, and the `jsonpath` match operators `@?` and `@@`.

Let's see how the default GIN index works in practice by creating a GIN index on the `pizza` column using the following query.

Listing 5.16 Creating a default GIN index

```
CREATE INDEX idx_pizza_orders_gin
ON pizzeria.order_items
USING GIN(pizza);
```

Once the index is created, let's generate an execution plan for the following query that calculates the total number of pizzas of the custom type:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"type": "custom"}';
```

Because the containment operator `@>` is supported by the GIN index, the execution plan indicates that Postgres used the newly created `idx_pizza_orders_gin index` for query execution:

```
                                QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate (actual time=0.784..0.785 rows=1 loops=1)
   ->  Bitmap Heap Scan on order_items
➥        (actual time=0.139..0.747 rows=563 loops=1)
         Recheck Cond: (pizza @> '{"type": "custom"}'::jsonb)
         Heap Blocks: exact=101
         ->  Bitmap Index Scan on idx_pizza_orders_gin
➥              (actual time=0.109..0.110 rows=563 loops=1)
               Index Cond: (pizza @> '{"type": "custom"}'::jsonb)
 Planning Time: 1.444 ms
 Execution Time: 0.830 ms
(8 rows)
```

Queries using GIN indexes rely on the `Bitmap Index Scan` access method because the index stores partial data, such as distinct keys and values, rather than the original JSON objects. Here's how GIN uses the `Bitmap Index Scan` for this and comparable queries:

1. Postgres analyzes the `Index Cond` and extracts distinct values from it. In our case, the values are `type` and `custom`.
2. The database performs a `Bitmap Index Scan` on the `idx_pizza_orders_gin` index, locating the index entries for `type` and `custom` in the index structure. During the index traversal, Postgres generates a bitmap with pointers to table rows that have `type` or `custom` in the structure of the original JSON object.

3 Postgres visits the rows listed in the bitmap (`Bitmap Heap Scan`) and evaluates them. Only rows where the `pizza` column satisfies the exact condition `{"type": "custom"}` with `type` as a top-level key and `custom` as its value are added to the query output. This ensures, for example, that the rows with the reverse structure `{"custom": "type"}` are not added to the output.

If the application queries other keys and values in the JSON structure, Postgres can still utilize the GIN index. For example, let's examine the execution plan for a query that calculates the total number of orders for small pizzas with pesto sauce:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"sauce": "pesto"}' and pizza @> '{"size": "small"}';
```

The execution plan confirms that Postgres continues to use the previously created `idx_pizza_orders_gin` index:

```
                                 QUERY PLAN
---------------------------------------------------------------------------
 Aggregate (actual time=0.330..0.331 rows=1 loops=1)
   ->  Bitmap Heap Scan on order_items
 ➥        (actual time=0.208..0.318 rows=35 loops=1)
          Recheck Cond: ((pizza @> '{"sauce": "pesto"}'::jsonb)
 ➥        AND (pizza @> '{"size": "small"}'::jsonb))
          Heap Blocks: exact=28
          ->  Bitmap Index Scan on idx_pizza_orders_gin
 ➥              (actual time=0.172..0.172 rows=35 loops=1)
                Index Cond: ((pizza @> '{"sauce": "pesto"}'::jsonb)
 ➥              AND (pizza @> '{"size": "small"}'::jsonb))
 Planning Time: 0.393 ms
 Execution Time: 0.386 ms
(8 rows)
```

As with the previous query, Postgres performed a `Bitmap Index Scan` over the GIN index, locating all pages with rows that include `sauce`, `pesto`, `size`, or `small` in their JSON pizza order structure. Following this, the database executed a `Bitmap Heap Scan` on the `order_items` table, visiting the rows listed in the bitmap and adding only those rows to the query output where the JSON structure fully satisfied the original condition (`pizza @> '{"sauce": "pesto"}' AND pizza @> '{"size": "small"}'`).

Because the GIN index stores distinct keys and values from the original JSON object, the database can also use it for more complex conditions involving nested JSON arrays. Let's examine the execution plan for the following query, which calculates the total number of pizza orders with a regular amount of cheddar cheese:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"toppings":{"cheese":[{"cheddar":"regular"}]}}';
```

The execution plan confirms that, regardless of the complexity of the condition in the `WHERE` clause, Postgres is still able to use the GIN index:

```
                                QUERY PLAN
------------------------------------------------------------------------
 Aggregate (actual time=1.290..1.291 rows=1 loops=1)
   ->  Bitmap Heap Scan on order_items
➥        (actual time=0.284..1.243 rows=638 loops=1)
         Recheck Cond: (pizza @>
➥          '{"toppings": {"cheese": [{"cheddar": "regular"}]}}'::jsonb)
         Rows Removed by Index Recheck: 90
         Heap Blocks: exact=101
         ->  Bitmap Index Scan on idx_pizza_orders_gin
➥              (actual time=0.248..0.248 rows=728 loops=1)
               Index Cond: (pizza @>
➥           '{"toppings": {"cheese": [{"cheddar": "regular"}]}}'::jsonb)
 Planning Time: 0.390 ms
 Execution Time: 1.351 ms
(9 rows)
```

Even though the condition `{"toppings":{"cheese":[{"cheddar":"regular"}]}}` appears more complex, Postgres simply extracted four distinct items—`toppings`, `cheese`, `cheddar`, and `regular`—and performed a `Bitmap Index Scan` over the GIN index, locating rows with these items in their JSON structure. As we've seen before, the database then executed a `Bitmap Heap Scan` on the table data, returning only the rows where the pizza orders fully satisfied the original condition, `pizza @> '{"toppings": {"cheese": [{"cheddar": "regular"}]}}'`. This includes matching the exact order and hierarchy of the keys and their values from the condition.

Overall, a single GIN index on JSON data is sufficient to enable fast searches over nested JSON structures. However, in addition to the default GIN index, Postgres offers another GIN index type that indexes only the paths of the JSON structure, providing even more efficient lookups at the cost of some flexibility.

Using nondefault GIN indexes with jsonb_path_ops

The second, nondefault GIN index type for JSON data in Postgres indexes the paths from the root of a JSON object down to each value and array element. Let's revisit this sample pizza order from our dataset:

```
{
 "size": "large",
 "type": "three cheese",
 "crust": "thin",
 "sauce": "marinara",
 "toppings": {
   "cheese": [
     {"cheddar": "regular"},
     {"mozzarella": "extra"},
     {"parmesan": "light"}
   ]
 }
}
```

If we create the nondefault GIN index type over the object, the following seven paths will be added to the index structure:

- Four simple paths from top-level keys to their string values:
  - `size.large`
  - `type.three cheese`
  - `crust.thin`
  - `sauce.marinara`
- Three paths from the toppings key down to each value in its nested JSON arrays:
  - `toppings.cheese.cheddar.regular`
  - `toppings.cheese.mozzarella.extra`
  - `toppings.cheese.parmesan.light`

Plus, instead of storing these seven paths as text values, the GIN index applies a hash function to each path's value and stores the hash codes in the index structure. This optimization reduces the index size and improves lookup efficiency by storing and comparing fixed-size integers rather than variable-sized text values. Figure 5.2 shows a simple representation of such a GIN index with a subset of the discussed index items.

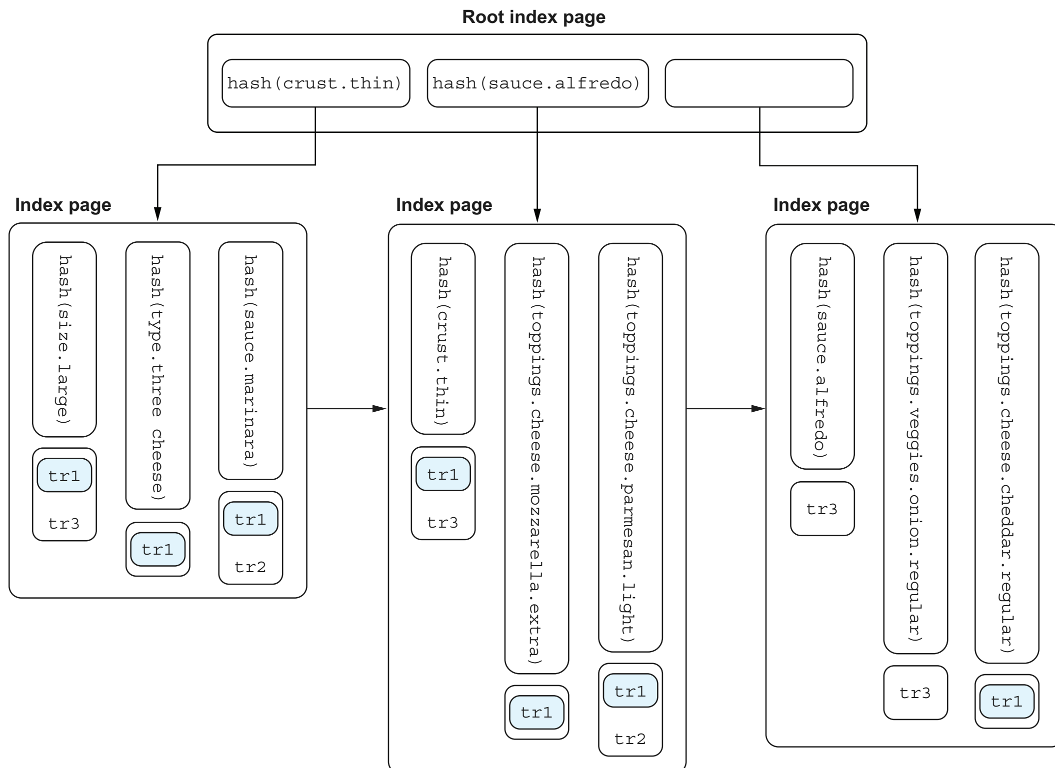


Figure 5.2 A sample nondefault GIN index holding references to the table row `tr1` with our sample pizza order

The search begins at the root index page, which contains three index entries pointing to index pages on the second level:

- `hash(crust.thin)`points to an index page with paths whose hash codes are smaller than the one returned by `hash(crust.thin)`.
- `hash(sauce.alfredo)` points to another index page holding index items with hash codes greater than or equal to the hash code of `hash(crust.thin)` but smaller than `hash(sauce.alfredo)`.
- The blank key points to the last index page that holds all other hash codes greater than or equal to `hash(sauce.alfredo)`.

The second-level index pages store index items with hash codes calculated for specific paths, which reference actual table rows. For example, the index entry for `hash(size.large)` references two table rows (`tr1` and `tr3`), indicating that the `pizza` column in those rows should have `size.large` in its JSON structure.

If we assume that `tr1` refers to our example pizza order, we can see that nearly all index entries, except for `hash(toppings.veggies.onion.regular)` and `hash(sauce.alfredo)`, reference that row. These two entries do not point to `tr1` because `onion` is not included in the toppings list and `alfredo` is not the sauce selected by the customer.

Once the index is created, the application can use the `jsonb_path_ops` operator class. This operator class supports the containment operator `@>` and the jsonpath match operators `@?` and `@@`. However, it doesn't support the key-exists operators including `?` because the index no longer stores distinct keys as well as values in its structure.

Let's see how the nondefault GIN index works in practice by creating it on the `pizza` column using the following query.

Listing 5.17 Creating a GIN index with the `jsonb_path_ops` class

```
CREATE INDEX idx_pizza_orders_paths_ops_gin
ON pizzeria.order_items
USING GIN (pizza jsonb_path_ops);
```

The `USING GIN (pizza jsonb_path_ops)` clause specifies that the index on the `pizza` column will be a GIN index with the `jsonb_path_ops` operator class. This is what we refer to as the nondefault GIN index, which indexes paths of JSON objects.

Once the index is created, let's examine the execution plan for the query calculating the total number of pizzas with custom recipes:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"type": "custom"}';
```

The execution plan looks as follows:

```
                                    QUERY PLAN
----------------------------------------------------------------------
 Aggregate (actual time=0.681..0.682 rows=1 loops=1)
   ->  Bitmap Heap Scan on order_items
➥         (actual time=0.106..0.644 rows=563 loops=1)
         Recheck Cond: (pizza @> '{"type": "custom"}'::jsonb)
         Heap Blocks: exact=101
         ->  Bitmap Index Scan on idx_pizza_orders_paths_ops_gin
➥               (actual time=0.070..0.070 rows=563 loops=1)
               Index Cond: (pizza @> '{"type": "custom"}'::jsonb)
 Planning Time: 0.544 ms
 Execution Time: 0.736 ms
(8 rows)
```

Postgres chose to use the newly created `idx_pizza_orders_paths_ops_gin` index, which also relies on the `Bitmap Index Scan` access method:

1. Postgres analyzes the `Index Cond` (`{"type": "custom"}`) and transforms it into the path `"type.custom"`. It then calculates the hash code for this path.
2. Using the calculated hash code, the database performs a `Bitmap Index Scan` over the GIN index to locate the index entry for the hash code. These entries point to table rows that might satisfy the search condition, and pointers to these rows are added to the bitmap.
3. Postgres performs a `Bitmap Heap Scan` on the table data, visiting all the rows from the bitmap and adding only those rows to the query output where the JSON `pizza` column fully satisfies the search condition. During this final verification step, the database can also handle potential hash collisions. Other paths in the JSON structure might produce the same hash code as `"type.custom"`, and this additional check ensures that only rows matching the exact condition are included in the result.

As we can see, Postgres already favors the nondefault GIN index with the `jsonb_path_ops` operator class for queries using the containment (`@>`) operator over the default GIN index created in the previous section. This preference is due to the fact that hash-code-based lookups on fixed-size integers are faster than searches over variable-size text data.

In addition to performance benefits, the nondefault GIN index is generally more compact because it stores hash codes in its structure. Let's execute the next query to compare the sizes of the two existing GIN indexes.

Listing 5.18 Comparing the size of GIN indexes

```
SELECT
    c.relname AS index_name,
    pg_size_pretty(pg_relation_size(c.oid)) AS index_size
FROM pg_class c
JOIN pg_index i ON c.oid = i.indexrelid
WHERE c.relname IN ('idx_pizza_orders_paths_ops_gin',
                    'idx_pizza_orders_gin');
```

According to the query output, the nondefault GIN index (`idx_pizza_orders_paths_ops_gin`) currently uses half the space of the default GIN index (`idx_pizza_orders_gin`):

```
           index_name           | index_size
--------------------------------+------------
 idx_pizza_orders_gin           | 112 kB
 idx_pizza_orders_paths_ops_gin | 56 kB
(2 rows)
```

However, although the nondefault GIN index uses less storage space and offers more efficient lookups for the containment `@>` operator, this efficiency comes at the cost of flexibility. The nondefault GIN index does not support lookups for keys in the JSON structure because it doesn't index the keys, and the `jsonb_path_ops` operator class does not support key-exists operators.

For example, imagine that the application needs to find all pizza orders with special instructions from customers. Such orders would have a top-level `special_instructions` key in their JSON structure, and the application can locate those orders using the key-exists `?` operator. Let's examine the execution plan for such a query:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza ? 'special_instructions';
```

This time, Postgres selected the default GIN index (`idx_pizza_orders_gin`), which stores distinct keys and values in the index structure, providing greater flexibility for the application:

```
                                QUERY PLAN
------------------------------------------------------------------------
 Aggregate (actual time=0.095..0.096 rows=1 loops=1)
   ->  Bitmap Heap Scan on order_items
 ➥       (actual time=0.089..0.089 rows=0 loops=1)
         Recheck Cond: (pizza ? 'special_instructions'::text)
         ->  Bitmap Index Scan on idx_pizza_orders_gin
 ➥             (actual time=0.077..0.077 rows=0 loops=1)
               Index Cond: (pizza ? 'special_instructions'::text)
 Planning Time: 0.641 ms
 Execution Time: 0.150 ms
(7 rows)
```

With that, we've learned how to store, query, modify, and index JSON data in Postgres while working on the application for the pizza chain.

Summary

- Postgres introduced initial support for JSON in 2012. Since then, JSON-specific capabilities and improvements have been added on a regular basis.
- The database supports two JSON-specific data types: `json` and `jsonb`. The `jsonb` data type should be used by default because it stores JSON objects in an internal binary format, which improves search performance.
- Postgres supports various operators and functions for querying JSON structures, such as the key-exists (`?`) and containment (`@>`) operators.
- In addition to the basic JSON functions and operators, Postgres offers a powerful JSON path language engine capable of executing expressions written in the SQL/JSON language.
- The database provides the `jsonb_set` function for updating existing JSON objects and the `#-` operator for removing specific data from a JSON structure.
- Postgres supports two types of GIN indexes for JSON objects, each storing indexed data differently and supporting distinct operator classes for data access.

6 Postgres for full-text search

This chapter covers

- Understanding how Postgres supports full-text search
- Converting textual data into lexemes
- Storing lexemes in the database
- Performing full-text search using built-in functions and operators
- Optimizing full-text search performance with GIN and GiST indexes

Postgres has supported various text data types and search operators for years. For instance, we can store text values in a column of the `TEXT` type and then use the equality operator (`=`), pattern-matching operators (`LIKE` and `ILIKE`), regular-expression-matching operators (`~` and `~*`), and other operators to query the text data. Although these capabilities work well for simple text data and queries, they are less effective when searching through large text documents that require understanding linguistic features like word variations or relevance ranking.

Full-text search is another core feature of Postgres that lets us perform advanced searches over textual data of varying size and complexity. Unlike basic text comparison or regex matching, it uses linguistic techniques to tokenize, normalize, rank, and index text data, making searches faster and more relevant.

Let's explore the full-text search capabilities of Postgres as we build a movie recommendation service that helps users find movies they like or might want to watch next. We'll learn how to preprocess and store movie descriptions in a format suited for full-text search and then query that data, returning movies that match the text users type in the search form.

6.1 Basics of full-text search in Postgres

Before our application can execute full-text search queries over the database, Postgres needs to preprocess the original text data by converting it into a format suitable for such queries. Once the data is preprocessed, we should store and, preferably, index the transformation results in the database to make it ready for application queries. After that, the application can execute full-text search queries over the database. This is how full-text search works in Postgres in a nutshell. Now, let's break this process down into pieces.

Overall, full-text search in Postgres works as follows:

1. *Tokenization*—The original text data, usually referred to as a *document*, is parsed into *tokens*, which are smaller text components such as words or phrases. For instance, the sentence "5 explorers are traveling" is tokenized into "5," "explorers," "are," and "traveling."
2. *Normalization*—The database converts tokens into *lexemes*, which are the basic units of meaning in a language. During normalization, Postgres removes case sensitivity, stems tokens to their root forms, and removes stop words such as articles and prepositions. For example, after normalizing the four tokens from the previous step, the database can produce the following three lexemes: "5," "explor," and "travel." The token "traveling" is stemmed to its root form, "travel," and "explorers" is converted to the lexeme "explor." The token "are" is removed because it is a stop word with no significant meaning. The token "5" remains unchanged.

NOTE The reason the word "explorers" is stemmed to "explor" rather than "explore" is that the default English stemmer in Postgres removes common suffixes like "-er" and "-s." As a result, "explorers" is normalized to "explor" by the stemmer.

3. *Storing and indexing*—The lexemes are stored in the database to avoid repeating the tokenization and normalization steps. Postgres provides a special data type called `tsvector`, specifically designed to store lexemes. Additionally, lexemes can be indexed to improve the performance of full-text search queries.

4. *Searching*—The application executes full-text search queries on the lexemes stored in the database.

In this section, we'll dive deeper into the tokenization and normalization processes. The storing, searching, and indexing aspects will be explored in the subsequent sections of the chapter.

NOTE If you'd like to gain practical experience while reading the chapter, connect to your Postgres instance started in chapter 1 using the `docker exec -it postgres psql -U postgres` command.

6.1.1 Tokenization and normalization

Imagine that a user of our movie recommendation service wants to find a movie matching the following description: "5 explorers are traveling to a distant galaxy." Let's use the following query to see how the database would tokenize and normalize this sentence.

Listing 6.1 Analyzing tokenization and normalization with `ts_debug`

```
SELECT token, description, lexemes, dictionary
FROM ts_debug('5 explorers are traveling to a distant galaxy');
```

The query makes a call to the `ts_debug` function used for testing and debugging full-text search configurations in Postgres. In this case, it provides the following details about the tokens and lexemes generated for the sentence:

```
   token    |   description    |  lexemes  |  dictionary
------------+------------------+-----------+--------------
 5          | Unsigned integer | {5}       | simple
            | Space symbols    |           |
 explorers  | Word, all ASCII  | {explor}  | english_stem
            | Space symbols    |           |
 are        | Word, all ASCII  | {}        | english_stem
            | Space symbols    |           |
 traveling  | Word, all ASCII  | {travel}  | english_stem
            | Space symbols    |           |
 to         | Word, all ASCII  | {}        | english_stem
            | Space symbols    |           |
 a          | Word, all ASCII  | {}        | english_stem
            | Space symbols    |           |
 distant    | Word, all ASCII  | {distant} | english_stem
            | Space symbols    |           |
 galaxy     | Word, all ASCII  | {galaxi}  | english_stem
(15 rows)
```

The `token` column lists all the tokens the sentence was split into, and the `description` column matches each token to a token type. The tokens from the sentence belong to the following three token types:

- `Unsigned integer`—Corresponds to the token "5."
- `Word, all ASCII`—Seven tokens fall into this group: "explorers," "are," "traveling," "to," "a," "distant," and "galaxy."
- `Space symbols`—Seven space symbols are shown as a blank value in the `token` column.

> **Postgres full-text search parser**
>
> Postgres includes a built-in parser that is used by default for all full-text search configurations. This parser is named `pg_catalog.default`, and you can find it by executing the following query:
>
> ```
> SELECT prsname FROM pg_ts_parser;
>  prsname
> ---------
>  default
> (1 row)
> ```

The `lexemes` column from the query output lists all the lexemes that the tokens were converted into during the normalization phase. Postgres uses dictionaries to perform this conversion, and the `dictionary` column shows which dictionary was used to transform each token into a lexeme. The following dictionaries were used during the normalization phase:

- The `simple` dictionary treats tokens as is without applying linguistic rules like stemming or stop-word removal. This dictionary left the token "5" untouched, converting it to the lexeme `{5}` because it's a simple numeric value.
- The `english_stem` dictionary was used for all tokens of the `Word, all ASCII` type. This dictionary applies stemming and stop-word removal based on English language rules. For example, the token "traveling" is transformed into the lexeme `{travel}`, and the token "explorers" is stemmed to `{explor}`. For stop-word removal, Postgres filters out common words such as articles or prepositions that are not useful for full-text search. For instance, the tokens "are," "to," and "a" are all removed by being mapped to the empty lexeme `{}`. Postgres maps stop words to `{}` instead of removing them entirely to preserve their position information within the original sentence. This is useful for the `<->` (followed by) operator, which we'll discuss in the chapter.
- The `dictionary` column is blank for tokens of the `Space symbols` type because these tokens are not mapped to any lexeme, and their position within the original sentence is not relevant for full-text search.

6.1.2 Full-text search configurations

Although Postgres has a single predefined full-text search parser that transforms a text document into tokens, it comes with multiple built-in dictionaries to convert tokens

into lexemes. These dictionaries are not selected by the database randomly. Instead, Postgres provides predefined *full-text search configurations* for many human languages, and these configurations determine which dictionary to use for a particular token type.

The default configuration is set during the database installation process and can be retrieved at runtime using the `default_text_search_config` parameter. If you started Postgres in Docker using the instructions from chapter 1, the default configuration is set to `english`. Execute the following query to confirm:

```
show default_text_search_config;

 default_text_search_config
----------------------------
 pg_catalog.english
(1 row)
```

Many full-text search functions depend on a configuration that can be passed as a special argument. This argument is usually optional, and if it is not provided, Postgres uses the configuration stored in the `default_text_search_config` parameter. For example, the `ts_debug` function, which we used in the previous section, accepts the following two arguments:

```
ts_debug([ config regconfig, ] document text)
```

The second argument is the text document we want to tokenize and normalize. The first argument (`regconfig`) allows us to specify one of the available full-text search configurations, and the square brackets `[]` indicate that this argument is optional.

When we execute the `ts_debug('5 explorers are traveling to a distant galaxy')` function without providing the `regconfig` argument, Postgres uses the default full-text search configuration. You can confirm this by explicitly passing the `english` configuration to the `ts_debug` function and verifying that it produces the same result as the query from listing 6.1, which does not specify any configuration:

```
SELECT token, description, lexemes, dictionary
FROM ts_debug('english', '5 explorers are traveling to a distant galaxy');
```

The output should look as follows:

```
   token    |   description    |  lexemes  |  dictionary
------------+------------------+-----------+--------------
 5          | Unsigned integer | {5}       | simple
            | Space symbols    |           |
 explorers  | Word, all ASCII  | {explor}  | english_stem
            | Space symbols    |           |
 are        | Word, all ASCII  | {}        | english_stem
            | Space symbols    |           |
 traveling  | Word, all ASCII  | {travel}  | english_stem
            | Space symbols    |           |
```

```
 to          | Word, all ASCII  | {}        | english_stem
             | Space symbols    |           |
 a           | Word, all ASCII  | {}        | english_stem
             | Space symbols    |           |
 distant     | Word, all ASCII  | {distant} | english_stem
             | Space symbols    |           |
 galaxy      | Word, all ASCII  | {galaxi}  | english_stem
(15 rows)
```

Next, if we want to see which dictionaries a configuration uses, we can execute the `\dF+ <configuration_name>` meta-command in the psql tool. For example, the following command lists all the dictionaries used by the `english` configuration:

```
\dF+ english
```

The output shows that the configuration uses either the `simple` or the `english_stem` dictionary, depending on the token type:

```
Text search configuration "pg_catalog.english"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | english_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | english_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | english_stem
```

Finally, we can use the `\dF` meta-command to view a complete list of predefined full-text search configurations. Here's what a truncated output for the `\dF` command looks like:

```
\dF
```

```
               List of text search configurations
   Schema   |    Name    |              Description
------------+------------+---------------------------------------
 pg_catalog | arabic     | configuration for arabic language
 pg_catalog | armenian   | configuration for armenian language
```

```
 pg_catalog | basque     | configuration for basque language
 pg_catalog | catalan    | configuration for catalan language
 pg_catalog | danish     | configuration for danish language
 pg_catalog | dutch      | configuration for dutch language
 pg_catalog | english    | configuration for english language
 pg_catalog | finnish    | configuration for finnish language
 pg_catalog | french     | configuration for french language
 pg_catalog | german     | configuration for german language
 pg_catalog | greek      | configuration for greek language
 pg_catalog | hindi      | configuration for hindi language
 pg_catalog | hungarian  | configuration for hungarian language
 pg_catalog | indonesian | configuration for indonesian language
 pg_catalog | irish      | configuration for irish language
 pg_catalog | italian    | configuration for italian language
...truncated output
```

Even though there is always one default configuration stored in the `default_text_search_config` parameter, we can still use other configurations at runtime. For example, the sentence "5 explorers are traveling to a distant galaxy" translates to "5 исследователей путешествуют к далёкой галактике" in Russian. The following query demonstrates how to use the `russian` full-text search configuration to tokenize and normalize the translated sentence:

Listing 6.2 Using the `russian` configuration for full-text search

```
SELECT token, description, lexemes, dictionary
FROM ts_debug('russian',
'5 исследователей путешествуют к далёкой галактике.');
```

The query returns the following result:

```
     token      |    description    |    lexemes      |  dictionary
----------------+-------------------+-----------------+--------------
 5              | Unsigned integer  | {5}             | simple
                | Space symbols     |                 |
 исследователей | Word, all letters | {исследовател}  | russian_stem
                | Space symbols     |                 |
 путешествуют   | Word, all letters | {путешеств}     | russian_stem
                | Space symbols     |                 |
 к              | Word, all letters | {}              | russian_stem
                | Space symbols     |                 |
 далёкой        | Word, all letters | {далек}         | russian_stem
                | Space symbols     |                 |
 галактике      | Word, all letters | {галактик}      | russian_stem
 .              | Space symbols     |                 |
(12 rows)
```

NOTE If none of the predefined full-text search configurations work for your needs, Postgres allows you to create custom configurations and use them for full-text search.

6.2 Preparing data for text search

After understanding how text tokenization and normalization work in Postgres, let's learn how to prepare text data for full-text search by first converting it into lexemes and then storing the lexemes in the database. Suppose we are part of the development team for our movie recommendation service, responsible for the search module. This module is used whenever end users or other application components request movie recommendations, and the job of the module is to return the most relevant results. Because Postgres is widely used in the company, we decided to take advantage of its full-text search capabilities for the search module. And our first goal is to prepare the company's movie dataset for text search.

6.2.1 Generating lexemes with the to_tsvector function

Postgres provides the `to_tsvector` function, which converts raw text into a list of lexemes. The function is defined as follows:

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

The function accepts two arguments:

- `regconfig`—This optional argument lets us specify one of the existing full-text search configurations discussed in the previous section. If not provided, the database uses the default configuration stored in the `default_text_search_config` parameter. In our case, the default configuration is `english`.
- `document`—This parameter can be any text string or document from our application. For the search module of our movie recommendation service, we will pass a movie title and its description.

When executed, the function returns a list of lexemes stored in the `tsvector` data type.

Let's see how the `to_tsvector` function works if one of our users searches for a movie with the following description:

The explorers must save the fragile peace between Earth and the aliens.

The following query shows how to convert this description into a `tsvector` representation of its lexemes.

Listing 6.3 Using the `to_tsvector` function

```
SELECT * FROM to_tsvector(
    'The explorers must save the fragile peace between Earth and the aliens.');
```

We omitted the `regconfig` argument, allowing Postgres to use the default `english` configuration. The database processed the sentence and returned the following `tsvector` representation of the lexemes:

```
                               to_tsvector
-----------------------------------------------------------------------
 'alien':12 'earth':9 'explor':2 'fragil':6 'must':3 'peac':7 'save':4
(1 row)
```

As we can see, the `to_tsvector` function used Postgres's full-text search parser and dictionaries to normalize the words to their root forms and remove stop words like the article "the" and the preposition "and." Each generated lexeme includes information about the position of the original word within the sentence.

For example, the lexeme "explor" corresponds to the word "explorers," which is the second word in the sentence. It follows the stop word "The," which begins the sentence. Notice that although stop words are removed from the output, their positional information is preserved. If this were not the case, the position of "explor" would be set to 1 instead of 2. This positional information is important for functions and operators that calculate the distance between lexemes, such as the `<->` (followed by) operator.

Next, our search module allows users to provide both the title and description of a movie. If both values are provided, we may want to use the string concatenation operator `||` to merge the title and description into a single text string before passing it to the `to_tsvector` function.

Suppose the user clarifies their original request and adds the title *Space Explorers* to the search. The following query shows how to use `to_tsvector` to generate lexemes for a concatenated text string that combines the movie title and description.

Listing 6.4 Using the `to_tsvector` function for a concatenated text string

```
SELECT * FROM to_tsvector(
'Space Explorers' ||
' ' ||                                  <- Adds the space symbol
'The explorers must save the fragile peace between Earth and the aliens.');
```

The query produces the following lexemes:

```
                                to_tsvector
-------------------------------------------------------------------------
 'alien':14 'earth':11 'explor':2,4 'fragil':8 'must':5 'peac':9
 ➥   'save':6 'space':1
(1 row)
```

Notice that because the word "explorers" appears twice—once in the title and once in the description—the "explor" lexeme stores positional information for both occurrences (`'explor':2,4`).

If we ever need to generate `tsvector` lexemes for a movie title and description separately, we can easily do so with a query like the one shown next.

Listing 6.5 Generating lexemes for the title and description separately

```
SELECT title_lexemes, description_lexemes FROM
 to_tsvector('Space Explorers') as title_lexemes,
 to_tsvector(
 'The explorers must save the fragile peace between Earth and the aliens.'
 ) as description_lexemes;
```

This query applies the `to_tsvector` function separately to the title and description, producing the following result:

```
    title_lexemes     |                  description_lexemes
----------------------+----------------------------------------------------
 'explor':2 'space':1 | 'alien':12 'earth':9 'explor':2 'fragil':6 'must':3
                      | 'peac':7 'save':4
(1 row)
```

As we can see, using the `to_tsvector` function to convert original text data into a list of lexemes is straightforward. Now, let's explore how to store the generated `tsvector` lexemes in the database.

6.2.2 Storing tsvector lexemes in the database

Once `tsvector` lexemes are generated, our application has several options for what to do with them:

- *Generate lexemes on the fly.* The application can run a full-text search query on the generated lexemes and then discard them. With this approach, the application uses the `to_tsvector` function to generate lexemes "on the fly" from table columns storing text data for every full-text search query. Although this approach works, it is not the most efficient, as lexeme generation must be repeated for every query, and the generated lexemes are discarded after the query completes.
- *Store lexemes in a column.* The application adds a column of the `tsvector` type to a table and stores the generated lexemes there. This column can then be indexed using a specialized generalized inverted index (GIN) or generalized search tree (GiST) index to improve search performance.
- *Index lexemes directly.* The application creates a GIN or GiST index directly from the generated `tsvector` lexemes without storing them in a table column. However, this approach requires specifying the full-text search configuration in the index creation statement, which may not work for some applications. But if it does work for an application use case, you'll save storage space by not storing copies of lexemes in the table column.

In this book, we'll focus on the second option. We'll store lexemes in a `tsvector` column in the `movies` table and then use Postgres's indexing capabilities to optimize the performance of full-text search queries on this data.

LOADING THE MOVIE DATASET

Our movie recommendation service uses a dataset derived from the free Open Media Database (OMDB). The derived dataset consists of a single table with the following structure, which is sufficient for our experiments with full-text search in Postgres:

```
omdb.movies (
    id BIGINT PRIMARY KEY,
    name TEXT NOT NULL,
    description TEXT NOT NULL,
    release_date DATE,
    runtime INT,
    budget NUMERIC,
    revenue NUMERIC,
    vote_average NUMERIC,
    votes_count BIGINT
);
```

This table stores essential information about a movie, including the title (`name`), `description`, `release_date`, `budget`, and other useful details.

> **What is the OMDB database?**
>
> OMDB (Open Media Database) is a free, community-driven database for movie data. You can explore it here: www.omdb.org.
>
> The database is maintained and expanded by a community of volunteers and is available under both the GNU Free Documentation License and the Creative Commons License. Anyone, including you, can join the community to add or modify existing information on OMDB.
>
> In this book, we use a subset of the OMDB data. However, if you want to pull the latest and complete dataset from OMDB and load it into Postgres, you can use the `omdb-postgresql` module: https://github.com/df7cb/omdb-postgresql.

If you'd like to gain practical experience while reading the chapter, follow these steps to preload the dataset into your Postgres instance started in chapter 1:

1. Clone the book's repository with listings and sample data:

   ```
   git clone https://github.com/dmagda/just-use-postgres-book
   ```

2. Copy the movie dataset to your Postgres container:

   ```
   cd just-use-postgres-book/
   docker cp data/movie/. postgres:/home/.
   ```

3. Preload the dataset by connecting to the container and using the `\i` meta-command of psql to apply the copied SQL scripts:

```
docker exec -it postgres psql -U postgres -c "\i /home/omdb_movies_ddl.sql"
docker exec -it postgres psql -U postgres -c "\i /home/omdb_movies_data.sql"
```

The table is created under the `omdb` schema, and you can confirm this by connecting to Postgres and executing the `\dt omdb.*` command:

```
docker exec -it postgres psql -U postgres
\dt omdb.*
```

The output will be as follows:

```
         List of relations
 Schema |  Name   | Type  |  Owner
--------+--------+-------+----------
 omdb   | movies | table | postgres
(1 row)
```

Next, confirm that slightly more than 4,200 movies are in the table:

```
SELECT count(*) FROM omdb.movies;

 count
-------
  4225
(1 row)
```

Storing lexemes in the movies table

The search module of our movie recommendation service will support full-text search on the data stored in the `name` and `description` columns of the `movies` table. For example, the following query retrieves data for a movie whose title contains "Space Odyssey":

```
SELECT id, name, description
FROM omdb.movies
WHERE name LIKE '%Space Odyssey%';
```

The output looks as follows:

```
-[ RECORD 1 ]-----------------------------------------------------------
id          | 62

name        | 2001: A Space Odyssey

description | The highly respected and often cited milestone in
 ➥   Science Fiction cinema. Director Stanley Kubrick takes us into
 ➥   the future, into outer space and aboard a space station that's
 ➥   controlled by a seemingly trustworthy computer named HAL.
```

TIP Use the `\x on` meta-command in psql to display data in a format similar to that from the previous output. This command enables expanded display mode, which shows each row vertically. It's helpful for results with many columns or large column values, making the data easier to read. To revert to the default tabular display mode, use the `\x off` command, which displays query results in a tabular format with rows and columns aligned horizontally.

Next, let's execute the `ALTER TABLE .. ADD COLUMN` command to add a column named `lexemes` of the `tsvector` type. This column will store lexemes generated from the contents of the `name` and `description` columns.

Listing 6.6 Adding a stored generated column for lexemes

```
ALTER TABLE omdb.movies
ADD COLUMN lexemes tsvector
GENERATED ALWAYS AS (
  to_tsvector(
    'english', coalesce(name, '') ||
    ' ' ||
    coalesce(description, ''))) STORED;
```

The command uses several statements and functions serving the following purpose:

- The `lexemes` column is defined as a stored generated column (`GENERATED ALWAYS ... STORED`), meaning its data is automatically updated whenever the source data changes. In essence, if the movie `name` or `description` changes, the contents of the `lexemes` column are automatically regenerated.
- We explicitly pass the `english` configuration to the `to_tsvector` function because Postgres requires the expression used for a generated column to be immutable. If the configuration is not explicitly provided, the query will fail. This is because `to_tsvector` would then rely on the value of the `default_text_search_config` setting, which can change at the session or database level.
- We can use the `coalesce(column, '')` function to handle scenarios where a column's value is set to `NULL`. If a value is `NULL`, `coalesce` will pass an empty string to `to_tsvector`. This step is necessary because `to_tsvector(NULL)` returns `NULL`.

NOTE In our case, the `name` and `description` columns can't store `NULL` values due to the `NOT NULL` constraints defined in the `CREATE TABLE` statement, so the `coalesce` function isn't really needed in the query. However, it's still worth understanding how to use it for columns that can store `NULL` values, which is why the function is used in code listings of the chapter.

Once the `lexemes` column is added, Postgres will automatically generate and store lexemes in the `movies` table. After that, we can run the following query to see the generated lexemes for the movie with "Space Odyssey" in the title:

```
SELECT id, name, description, lexemes
FROM omdb.movies
WHERE name LIKE '%Space Odyssey%';
```

The output looks as follows:

```
-[ RECORD 1 ]----------------------------------------------------------------
id          | 62

name        | 2001: A Space Odyssey

description | The highly respected and often cited milestone in Science
➥   Fiction cinema. Director Stanley Kubrick takes us into the future,
➥   into outer space and aboard a space station that's controlled by
➥   a seemingly trustworthy computer named HAL.

lexemes     | '2001':1 'aboard':28 'cinema':15 'cite':10 'comput':39
➥   'control':34 'director':16 'fiction':14 'futur':23 'hal':41
➥   'high':6 'kubrick':18 'mileston':11 'name':40 'odyssey':4
➥   'often':9 'outer':25 'respect':7 'scienc':13 'seem':37
➥   'space':3,26,30 'stanley':17 'station':31 'take':19
➥   'trustworthi':38 'us':20
```

With all the required data in place, we're now ready to explore how to use Postgres's full-text search queries for our movie recommendation service.

6.3 Performing full-text search

The full-text search queries that our application plans to execute on the generated `tsvector` lexemes have to be transformed into the `tsquery` data type. This transformation is necessary because application queries also need to be normalized by removing stop words and converting the remaining words into lexemes. Also, the lexemes in the `tsquery` representation are combined using special operators like `&` (logical AND) and `|` (logical OR), which lets us add filter expressions in full-text search queries.

Once we have the `tsvector` lexemes generated from our dataset and the application queries in the `tsquery` format, we can use Postgres's match operator (`@@`) to perform the full-text search. This operator compares the `tsvector` data to the `tsquery` search term and evaluates to `true` if there is a match or `false` otherwise. Let's put everything together and see how it works in action.

6.3.1 Using plainto_tsquery for simple queries

Imagine that one of our users wants to watch an animated movie and types the phrase "a computer animated film" into the movie recommendation service user interface (UI). When the application passes this phrase to Postgres for full-text search, the database needs to convert it into the `tsquery` format.

One way to achieve this is by using `plainto_tsquery`, which is defined as follows:

```
plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

The function takes the raw text version of a query via the `querytext` argument and converts it into the `tsquery` type. Like many other full-text search functions, `plainto_tsquery` accepts an optional `regconfig` argument if we need to specify a full-text search configuration other than the default.

Let's execute the following query to see how `plainto_tsquery` transforms raw text into a `tsquery`:

```
SELECT plainto_tsquery('a computer animated film');
```

Spell check and word similarity with trigrams

Even though the phrase "computer animated" is not grammatically correct and should be written as "computer-animated," let's assume this is the exact phrase the user typed in the application UI. At the end of the day, our application should be prepared to handle minor grammatical errors and typos.

If we want our application to recognize misspelled words or support an auto-complete feature that suggests highly likely words the user might want to type, we can use *trigrams* (groups of three consecutive characters extracted from a string), which are available via the `pg_trgm` extension. Trigram matching is beyond the scope of this book, but you can explore it further in the Postgres documentation: https://www.postgresql.org/docs/current/pgtrgm.html.

The query produces the following result:

```
      plainto_tsquery
----------------------------
 'comput' & 'anim' & 'film'
(1 row)
```

As we can see, the function removed the article "a" because it is a stop word and normalized the remaining words to their root forms. Also, the `&` (AND) operator was inserted between the words, indicating that all the lexemes—"comput," "anim," and "film"—must be present in the data the query will be executed against for a match to occur.

Once the query is transformed into the `tsquery` representation, we can use the match `@@` operator to perform a full-text search by executing the `tsquery` value against the previously generated `tsvector` lexemes. The following query shows how to use the `@@` operator to find all the movies corresponding to the user's search request:

Listing 6.7 Executing our first full-text search query

```
SELECT id, name
FROM omdb.movies
WHERE lexemes @@ plainto_tsquery('a computer animated film');
```

The query works as follows:

1. *Transforms the user phrase*—We use the `plainto_tsquery` function to convert the user phrase "a computer animated film" into the `tsquery` format.
2. *Performs the full-text search*—The `@@` operator compares the `tsquery` value against the contents of the `lexemes` column that stores `tsvector` values generated from the movie titles and descriptions.
3. *Returns the result*—The query returns the `id` and `name` of each movie that satisfies the full-text search condition from the `WHERE` clause.

As a result, our first full-text search query returns the following movie recommendations to the user:

```
   id   |            name
--------+---------------------------
     12 | Finding Nemo
    280 | Terminator 2: Judgment Day
    425 | Ice Age
    585 | Monsters, Inc.
    604 | The Matrix Reloaded
    810 | Shrek the Third
    862 | Toy Story
    863 | Toy Story 2
    950 | Ice Age: The Meltdown
    953 | Madagascar
   1273 | TMNT
   2310 | Beowulf
  26301 | Avatar
  32683 | The Croods
 120862 | The Lion King
   9928 | Robots
(16 rows)
```

6.3.2 Using to_tsquery for advanced filtering

As we've seen, the `plainto_tsquery` function is useful in scenarios where all lexemes from a generated query must be present in a target `tsvector` value derived from our dataset. However, if we need more advanced filtering during a search—such as using the `&` (AND), `|` (OR), or `<->` (FOLLOWED BY) operator, or any combination of them—we can use the `to_tsquery` function.

The `to_tsquery` function is defined as follows:

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

The function returns a `tsquery` value created from `querytext`, which must include text tokens/words separated by the `&` (AND), `|` (OR), `!` (NOT), and `<->` (FOLLOWED BY) operators. It also allows grouping tokens using parentheses. Also, like other full-text search functions, `to_tsquery` accepts an optional `regconfig` argument if we need to specify a full-text search configuration different from the default one.

Imagine that after reviewing the list of suggestions produced by the query in listing 6.7, the user decides to be more specific and asks our recommendation service to suggest computer-animated movies featuring a lion, a clownfish, or a donkey. Our search module can use the next query to find the requested information.

Listing 6.8 Using a combination of AND and OR operators

```
SELECT id, name
FROM omdb.movies
WHERE lexemes @@ to_tsquery('computer & animated
      & (lion | clownfish | donkey)');
```

The expression passed to the `to_tsquery` function asks Postgres to find all movies containing the lexemes corresponding to the "computer" and "animated" tokens. Among these, the database will return only those that also include "lion," "clownfish," or "donkey" in their description.

When we execute this query, Postgres suggests the following animated movies that match the search criteria:

```
   id   |     name
--------+---------------
     12 | Finding Nemo
    808 | Shrek
 120862 | The Lion King
(3 rows)
```

Next, the user refines their search further, asking the movie recommendation service to return all movies featuring lions but excluding *The Lion King*. The following query shows how to translate this request into a full-text search query.

Listing 6.9 Using the NOT operator and filtering by phrase

```
SELECT id, name
FROM omdb.movies
WHERE lexemes @@ to_tsquery('lion & !''The Lion King''');
```

This query uses the `!` (NOT) operator to exclude any movies with the phrase "The Lion King" in their description. The phrase must be enclosed in two single quotes from each side to ensure that it is treated as a single unit.

The output for this query shows a variety of other movies featuring lions:

```
   id   |                              name
--------+-------------------------------------------------------------
    411 | The Chronicles of Narnia: The Lion, the Witch and the Wardrobe
    630 | The Wizard of Oz
   9904 | The Wild
  35066 | Four Lions
  99216 | Lion
```

```
 130325 | Les misérables
(6 rows)
```

How does Postgres check that the phrase "The Lion King" is not present in the movie description? It's simple to understand by looking into the result of the `to_tsquery('lion & !''The Lion King''')` function call:

```
SELECT * FROM to_tsquery('lion & !''The Lion King''');

           to_tsquery
---------------------------------
 'lion' & !( 'lion' <-> 'king' )
(1 row)
```

As the output shows, the `to_tsquery` function converts the phrase into two lexemes ("lion" and "king") and removes "the" because it is a stop word. Also, the function adds the `<->` (FOLLOWED BY) operator between the lexemes, indicating that "king" must immediately follow "lion" in the description.

The `<->` operator allows us to specify the distance between lexemes. For example, if we want to find a movie where "king" is the third token after "return," we can achieve this by using the `<N>` operator and setting `N` to `3`, as shown in this example:

```
SELECT id, name
FROM omdb.movies
WHERE lexemes @@ to_tsquery('return <3> king');
```

This query returns a single match:

```
 id  |                     name
-----+---------------------------------------------------
 122 | The Lord of the Rings: The Return of the King
(1 row)
```

This result satisfies the search criteria because the description includes the token "return," followed by "of," "the," and "king," with "king" being the third token after "return."

Additional functions for converting queries into the tsquery type

In this book, we've explored how to use the `plainto_tsquery` and `to_tsquery` functions to convert an application query into the `tsquery` data type. Postgres provides two additional functions that you might find useful in your applications:

- The `phraseto_tsquery` function works similarly to `plainto_tsquery`, except it inserts the `<->` operator between lexemes instead of the `&` operator. It also accounts for stop words by inserting `<N>` instead of `<->` operators whenever applicable.

(continued)

- The `websearch_to_tsquery` function is ideal for processing raw input directly from users. Users can include simple search operators (like AND, OR, or quotes for phrases) without worrying about special formatting, as the function is designed to handle such input without raising syntax errors.

With that, we've learned how to perform full-text search in Postgres using its built-in functions and operators. The next step is to explore how to rank search results.

6.4 Ranking search results

Imagine that a user is in the mood to watch a movie about ghosts. The user types "ghosts" into the search bar and clicks the Search button. In response, our movie recommendation service executes the following query against Postgres.

Listing 6.10 Returning movies containing the word "ghosts"

```
SELECT id, name, vote_average
FROM omdb.movies
WHERE lexemes @@ to_tsquery('ghosts')
ORDER BY vote_average DESC NULLS LAST LIMIT 10;
```

The database returns the following list of movies, ordered by the `vote_average` column:

```
  id   |               name                | vote_average
-------+-----------------------------------+--------------
 84904 | A Girl Walks Home Alone at Night |            9
  1548 | Ghost World                       | 8.1428575516
 25028 | Miracolo a Milano                 |            8
   426 | Vertigo                           | 7.6849312782
   745 | The Sixth Sense                   | 7.5660376549
   620 | Ghostbusters                      | 7.0500001907
 57784 | ParaNorman                        |            7
 11439 | The Ghost Writer                  | 6.5833334923
   251 | Ghost                             | 6.3333301544
 10016 | Ghosts of Mars                    |  6.222219944
(10 rows)
```

However, the question remains: Is this ranking approach the most accurate and relevant from a full-text search perspective? Should we return the result as is or order it differently?

For example, the current ranking does not account for whether "ghost" (the root form of "ghosts" produced by the `to_tsquery('ghosts')` function) appears in the title or description of the movie. Among the top five suggested movies, only one movie—*Ghost World*—has "ghost" in its name.

On top of that, this approach does not consider how frequently the "ghost" lexeme appears in the movie titles and descriptions. For instance, let's execute the following

query to compare how many times "ghost" appears in the movies *A Girl Walks Home Alone at Night* (ranked first) and *Ghost* (ranked ninth):

```
SELECT id, name, lexemes
FROM omdb.movies
WHERE id IN (84904, 251);
```

The output is generated using the extended display mode, enabled with the `\x on` meta-command, and looks as follows:

```
-[ RECORD 1 ]--------------------------------------------------------------
id      | 251

name    | Ghost

lexemes | 'award':6 'chaotic':19 'contact':24 'demi':29 'ghost':1,2,15
➥  'help':22 'love':8 'moor':30 'oscar':5 'patrick':11 'play':27
➥  'psychic':20 'star':10 'stori':9 'swayz':12 'use':17 'wife':26
➥  'win':7

-[ RECORD 2 ]--------------------------------------------------------------
id      | 84904

name    | A Girl Walks Home Alone at Night

lexemes | 'addict':28 'alon':5 'also':40 'arash':8 'bad':15 'citi':16
➥  'deal':33 'drug':27,34 'drug-addict':26 'eke':19 'exist':22
➥  'father':29 'ghost':13 'girl':2,44 'home':4,41 'iranian':12
➥  'live':9 'men':50 'night':7 'outcast':18 'prey':48 'town':14,38
➥  'vampir':46 'walk':3
```

When we locate the "ghost" lexemes in the output for both records, we can see that "ghost" appeared three times (`'ghost':1,2,15`) in the movie *Ghost* and only once (`'ghost':13`) in *A Girl Walks Home Alone at Night*. However, despite this fact, *Ghost* is still ranked much lower.

The good news is that Postgres provides two built-in functions to calculate the relevance ranking of full-text search results. The first function is called `ts_rank`, and it ranks results based on term frequency and their position in a text document. The second function, `ts_rank_cd`, is a variant of `ts_rank` that factors in term density, giving higher ranks to documents where query terms are closer together. Let's explore how to use the `ts_rank` function to improve the ranking approach in our movie recommendation service.

Using ts_rank for search result ranking

The `ts_rank` function has the following definition:

```
ts_rank([ weights float4[], ] vector tsvector,
  query tsquery [, normalization integer ]) returns float4
```

The function arguments serve the following purposes:

- The `weights` argument is optional and allows us to weigh word instances differently based on their labels. Postgres supports four labels—`A`, `B`, `C`, and `D`—with `A` typically assigned to the highest-priority parts of a document and `D` to the lowest-priority parts. Weights are passed as an array in the format `{D-weight, C-weight, B-weight, A-weight}`. If the `weights` argument is omitted, the function uses default values: `{0.1, 0.2, 0.4, 1.0}`. The closer the weight is to 1, the higher the priority.
- The `vector` argument is a list of lexemes stored in the `tsvector` data type, generated for a particular text value or document.
- The `query` argument is the search query, translated into the `tsquery` type.
- The `normalization` argument is optional and specifies how a document's length should influence its rank. The argument is a bitmask that accepts the following values:
  - `0` (default)—No normalization means the document's length is ignored.
  - `1`—Divide by the document length.
  - `2`—Divide by the mean term frequency.

  Additional values are also available for other normalization methods. Plus, the bitmask can be a combination of several values.

Now, imagine that we've updated our application logic to use the `ts_rank` function for ranking movies that match a search term. The following query demonstrates how this implementation might look.

Listing 6.11 Ranking search results with the `ts_rank` function

```
SELECT id, name, vote_average,
  ts_rank(lexemes, to_tsquery('ghosts')) AS search_rank
FROM omdb.movies
WHERE lexemes @@ to_tsquery('ghosts')
ORDER BY search_rank DESC, vote_average DESC NULLS LAST LIMIT 10;
```

The query execution flow looks as follows:

1. Postgres finds all movies where the `lexemes` column matches the `to_tsquery('ghosts')` query.
2. For each matched movie, the database calculates its `search_rank` using the `ts_rank(lexemes, to_tsquery('ghosts'))` function. This function determines the rank based on how many times the "ghost" lexeme (the root form of "ghosts") appears in the `lexemes` value.
3. The search result is sorted first by `search_rank` in descending order and then by `vote_average` in descending order.

Once the query is executed, the application will return the list of movies sorted in the following order:

```
   id   |              name              | vote_average | search_rank
--------+--------------------------------+--------------+-------------
    251 | Ghost                          | 6.3333301544 | 0.082745634
 210675 | A Most Annoying Ghost          |              | 0.082745634
   1548 | Ghost World                    | 8.1428575516 | 0.075990885
    620 | Ghostbusters                   | 7.0500001907 | 0.075990885
  57784 | ParaNorman                     |            7 | 0.075990885
  12175 | Ghosts of Goldfield            |            6 | 0.075990885
 185828 | Dhamilo Pani                   |              | 0.075990885
 166008 | Ghosts                         |              | 0.075990885
  84904 | A Girl Walks Home Alone at Night |            9 |  0.06079271
  25028 | Miracolo a Milano              |            8 |  0.06079271
(10 rows)
```

As we can see, the movies with the highest `search_rank` values are listed first. This indicates that the "ghost" lexeme (produced by the `to_tsquery('ghosts')` function) appeared more frequently in the `lexemes` list, corresponding to their titles and descriptions.

Also, some of the ranked movies have an empty value in the `vote_average` column, meaning users have not voted for them yet. But despite the missing user rank, these movies are still included in the output because they matched the full-text search criteria.

USING WEIGHTS IN SEARCH RESULT RANKING

Currently, the `ts_rank` function used in listing 6.11 does not differentiate whether the "ghost" lexeme appears in the title or description of a movie. This is because we have not assigned a higher weight to a movie title compared to its description.

For example, let's examine the lexemes generated from the title and description of *Ghost*, which was ranked first by the query in listing 6.11:

```
SELECT id, name, description, lexemes
FROM omdb.movies
WHERE id = 251;

-[ RECORD 1 ]-----------------------------------------------------------
id          | 251
name        | Ghost
description | Ghost is an Oscar award winning love story starring Patrick
➥   Swayze as a ghost who uses a chaotic psychic to help him contact
➥   his wife played by Demi Moore.
lexemes     | 'award':6 'chaotic':19 'contact':24 'demi':29
➥   'ghost':1,2,15 'help':22 'love':8 'moor':30 'oscar':5
➥   'patrick':11 'play':27 'psychic':20 'star':10 'stori':9
➥   'swayz':12 'use':17 'wife':26 'win':7
```

As you recall, the `lexemes` vector was generated from a text value created by concatenating the movie `name` and `description`. The "ghost" lexeme appears three times

('ghost':1,2,15) in this concatenated value: once in the part corresponding to the movie name (position 1) and twice in the part derived from the movie description (positions 2 and 15 after concatenating name and description).

Apart from the positional information, no additional data would allow Postgres to weigh the "ghost" occurrences in the movie title differently from those in the description. However, it's easy to assign different weights to the name and description portions of the movies using Postgres' setweight function. This function allows us to label the tsvector lexemes with a specified weight, which can be set to one of the letters A, B, C, or D. For example, the following query shows how to use the setweight function to assign different weights to the name and description portions of *Ghost* (id = 251).

Listing 6.12 Assigning weights with the setweight function

```
SELECT id, name, description,
  (setweight(to_tsvector('english', coalesce(name, '')), 'A') ||
   setweight(to_tsvector('english', coalesce(description, '')), 'B'))
     as lexemes_with_weight
FROM omdb.movies
WHERE id = 251;
```

This query assigns the A (highest) weight to the tsvector generated from the movie name and the B (lower) weight to the tsvector created from the description. The two vectors are then concatenated using the || operator, with the resulting value stored in the lexemes_with_weight column.

When we execute the query, the output will look as follows:

```
-[ RECORD 1 ]-------+----------------------------------------------------
id                  | 251

name                | Ghost

description         | Ghost is an Oscar award winning love story starring
➥  Patrick Swayze as a ghost who uses a chaotic psychic to help him
➥  contact his wife played by Demi Moore.

lexemes_with_weight | 'award':6B 'chaotic':19B 'contact':24B 'demi':29B
➥  'ghost':1A,2B,15B 'help':22B 'love':8B 'moor':30B 'oscar':5B
➥  'patrick':11B 'play':27B 'psychic':20B 'star':10B 'stori':9B
➥  'swayz':12B 'use':17B 'wife':26B 'win':7B
```

This time, the "ghost" lexeme includes weights in addition to positional information ('ghost':1A,2B,15B). The A weight indicates that the lexeme appeared in the movie name, and the B weight shows that it appeared in the movie description.

Now that we've learned how to assign weights to different portions of text, let's recreate the lexemes column in our movies table to ensure that it uses the setweight function. The following query shows how to first drop the lexemes column and then add it back, applying logic that assigns different weights to the movie name and description.

Listing 6.13 Re-creating stored generated column for lexemes

```
ALTER TABLE omdb.movies
DROP COLUMN lexemes;

ALTER TABLE omdb.movies
ADD COLUMN lexemes tsvector
GENERATED ALWAYS AS (
    setweight(to_tsvector('english', coalesce(name, '')), 'A') ||
    setweight(to_tsvector('english', coalesce(description, '')), 'B')
) STORED;
```

After re-creating the column, let's find the movies featuring ghosts one more time:

```
SELECT id, name, vote_average,
  ts_rank(lexemes, to_tsquery('ghosts')) AS search_rank
FROM omdb.movies
WHERE lexemes @@ to_tsquery('ghosts')
ORDER BY search_rank DESC, vote_average DESC NULLS LAST LIMIT 10;
```

This time, the `ts_rank` function accounts for the weights and lists all movies with the word "ghost" in the title at the top of the result:

```
   id   |         name          | vote_average | search_rank
--------+-----------------------+--------------+-------------
    251 | Ghost                 | 6.3333301544 |   0.6957388
 210675 | A Most Annoying Ghost |              |   0.6957388
   1548 | Ghost World           | 8.1428575516 |  0.66871977
  12175 | Ghosts of Goldfield   |            6 |  0.66871977
 166008 | Ghosts                |              |  0.66871977
  11439 | The Ghost Writer      | 6.5833334923 |   0.6079271
  10016 | Ghosts of Mars        |  6.222219944 |   0.6079271
    620 | Ghostbusters          | 7.0500001907 |  0.30396354
  57784 | ParaNorman            |            7 |  0.30396354
 185828 | Dhamilo Pani          |              |  0.30396354
(10 rows)
```

Using weights in tsquery search terms

Weights can also be defined directly in the `tsquery` search term to restrict the full-text search to lexemes with a specific weight label. For example, in the following query, the `to_tsquery` function in the `WHERE` clause ensures that only lexemes labeled with weight `A` are considered during the search (`to_tsquery('ghosts:A')`):

```
SELECT id, name, vote_average,
  ts_rank(lexemes, to_tsquery('ghosts')) AS search_rank
FROM omdb.movies
WHERE lexemes @@ to_tsquery('ghosts:A')
ORDER BY search_rank DESC, vote_average DESC NULLS LAST LIMIT 10;
```

(continued)

The output for this query is as follows, excluding all movies that don't have the "ghost" lexeme in their titles:

```
   id   |          name          | vote_average | search_rank
--------+------------------------+--------------+-------------
    251 | Ghost                  | 6.3333301544 |   0.6957388
 210675 | A Most Annoying Ghost  |              |   0.6957388
   1548 | Ghost World            | 8.1428575516 |  0.66871977
  12175 | Ghosts of Goldfield    |            6 |  0.66871977
 166008 | Ghosts                 |              |  0.66871977
  11439 | The Ghost Writer       | 6.5833334923 |   0.6079271
  10016 | Ghosts of Mars         |  6.222219944 |   0.6079271
(7 rows)
```

6.5 Highlighting search results

Imagine that our search module supports the option to highlight words in movie descriptions that match a search term. This feature is particularly useful for other components of our movie recommendation service, which may want to retrieve a list of description fragments matching the search and perform additional processing on them. In this case, we can again rely on Postgres' built-in full-text search capabilities to implement the result highlighter feature.

Postgres provides the `ts_headline` function, which returns fragments of a text string or document with search terms highlighted. The function is defined as follows:

```
ts_headline([ config regconfig, ] document text,
  query tsquery [, options text ]) returns text
```

The function accepts a text `document` along with a `query` containing the search terms the application needs to locate and highlight in the `document`. It returns fragments from the `document` in which the query terms are highlighted. Additionally, the function supports the optional `regconfig` parameter for specifying a full-text search configuration and the `options` parameter to customize the output of the `ts_headline` function.

Let's explore how `ts_headline` works in practice by executing the following query, which returns highlighted fragments of a movie about pirates with the highest rank.

Listing 6.14 Using `ts_headline` to highlight search results

```
SELECT id, name,description,
    ts_headline(description, to_tsquery('pirates')) AS fragments,
    ts_rank(lexemes, to_tsquery('pirates')) AS rank
FROM omdb.movies
WHERE lexemes @@ to_tsquery('pirates:B')
ORDER BY rank DESC LIMIT 1;
```

The query execution flow looks as follows:

1. Postgres finds all movies where the `lexemes` column matches the `to_tsquery('pirates:B')` query. This query searches only the lexemes derived from the `description` column that are labeled with the `B` weight.
2. For each matched movie, the database uses the `ts_headline` function to extract fragments of the `description` column, highlighting the words that satisfy the `to_tsquery('pirates')` query. Note that, in this case, we don't include the `B` weight in the query because the `ts_headline` function works with the original value of the `description` column, generating lexemes without any weights.
3. Postgres orders the movies by their `rank` in descending order and applies the `LIMIT 1` clause to return a single movie with the highest rank. The `ts_rank` function, which calculates the rank, doesn't include the `B` weight in its `to_tsquery('pirates')` query to ensure that `rank` also considers occurrences of the term "pirates" in lexemes derived from the `name` column, which are labeled with the `A` weight.

Once the query is executed, the database will return the following movie:

```
-[ RECORD 1 ]------------------------------------------------------------
id          | 58

name        | Pirates of the Caribbean: Dead Man's Chest

description | Captain Jack is back! The bizarre and infamous pirate
➥   Captain Jack is in a battle with the ocean itself. Jack knows
➥   it won't be easy and gathers friends to help him on his mission to
➥   find the heart of Davy Jones in this second and more slapstick
➥   film from the pirate trilogy.

fragments   | <b>pirate</b> Captain Jack is in a battle with the ocean
➥   itself. Jack knows it won't be easy

rank        | 0.6957388
```

If we take a look at the contents of the `fragments` column in the result, we see that `ts_headline` returns a text fragment where the word "pirate" (the root form of "pirates") is enclosed in `<b>` and `</b>` tags. These tags are the default markers Postgres uses to highlight words that match search terms.

Next, if we examine the movie `description` from the previous result, we notice that there was another occurrence of the word "pirate" in the text ("film from the pirate trilogy"), but for some reason, it wasn't highlighted or included in the `fragments` column. This is because, by default, the `ts_headline` function returns only a single text fragment, and it chose to return the fragment containing the occurrence of "pirate" at the beginning of the `description` rather than the one near the end.

WARNING The output of `ts_headline` is not guaranteed to be safe for direct inclusion in web pages and can be exploited in cross-site scripting (XSS) attacks

when processing untrusted input. In our case, both the `name` and `description` columns store plain text values and are not vulnerable to those attacks. However, if you store and query documents containing HTML markup, you should either remove all HTML tags from the input documents or sanitize the output with an HTML sanitizer to prevent XSS attacks.

The good news is that we can pass additional options to the `ts_headline` function to include missing fragments and adjust its behavior the way our application needs. The next query shows how to include several supported options to customize the function's output.

Listing 6.15 Customizing `ts_headline` to show additional fragments

```
SELECT id, name, description,
  ts_headline(description, to_tsquery('pirates'),
    'MaxFragments=3, MinWords=5, MaxWords=10,
     FragmentDelimiter=<ft_end>') AS fragments,
  ts_rank(lexemes, to_tsquery('pirates')) AS rank
FROM omdb.movies
WHERE lexemes @@ to_tsquery('pirates:B')
ORDER BY rank DESC LIMIT 1;
```

In this example, the options are passed as the third argument of the `ts_headline` function call and have the following meanings:

- The `MaxFragments` option controls the maximum number of fragments to return. Setting it to `3` ensures that up to three distinct text fragments containing the search term are included in the result.
- The `MinWords` and `MaxWords` options define the minimum and maximum number of words each fragment can have. In our case, each fragment must contain at least 5 words but no more than 10. The default values are `15` and `35`.
- The `FragmentDelimiter` option allows customization of the delimiter that separates fragments. By default, Postgres uses `...` as the delimiter, but here it is changed to `<ft_end>`, which is a delimiter agreed on for our movie recommendation service.

Once the query is executed, Postgres will produce the following output:

```
-[ RECORD 1 ]------------------------------------------------------------
id          | 58

name        | Pirates of the Caribbean: Dead Man's Chest

description | Captain Jack is back! The bizarre and infamous pirate
 ➥  Captain Jack is in a battle with the ocean itself. Jack knows it
 ➥  won't be easy and gathers friends to help him on his mission to
 ➥  find the heart of Davy Jones in this second and more slapstick
 ➥  film from the pirate trilogy.
```

```
fragments    | bizarre and infamous <b>pirate</b> Captain Jack is
 ➥  in a battle<ft_end>slapstick film from the <b>pirate</b> trilogy

rank         | 0.6957388
```

This time, the `fragments` column lists all occurrences of the word "pirate" that belong to two distinct fragments, separated by our custom `<ft_end>` delimiter.

NOTE You can customize the output of the `ts_headline` function further by using additional options. Refer to the Postgres official documentation for a complete list of supported options: https://www.postgresql.org/docs/current/textsearch-controls.html#TEXTSEARCH-HEADLINE.

6.6 Indexing lexemes

So far, we've explored how to use various Postgres full-text search capabilities for our movie recommendation service. We've learned to generate `tsvector` lexemes from movie descriptions, store them in the database, and create and execute full-text search queries over the generated `tsvector` data. The next logical step is to make the search over those lexemes more efficient by using the database's indexing capabilities.

Let's take a look at the execution plan Postgres uses to return movie recommendations. For instance, suppose a user wants to watch a historical movie where people fight for something that truly matters. The application executes the following query against Postgres:

```
SELECT id, name FROM omdb.movies
WHERE lexemes @@ to_tsquery('historical & fight');
```

The database finds three movies that match the search criteria:

```
  id   |         name
-------+---------------------
    98 | Gladiator
 10105 | Saints and Soldiers
   197 | Braveheart
(3 rows)
```

But how much effort does it take for the database to find these three records? Let's execute the following query to analyze the actual query execution plan:

```
EXPLAIN (analyze, costs off)
SELECT id, name FROM omdb.movies
WHERE lexemes @@ to_tsquery('historical & fight');
```

The plan looks as follows:

```
                              QUERY PLAN
---------------------------------------------------------------
 Seq Scan on movies (actual time=0.226..15.301 rows=3 loops=1)
   Filter: (lexemes @@ to_tsquery('historical & fight'::text))
   Rows Removed by Filter: 4222
 Planning Time: 0.179 ms
 Execution Time: 15.328 ms
(5 rows)
```

The execution plan shows that Postgres performed a full table scan, visiting every row in the movies table (`Seq Scan on movies`). The `to_tsquery('historical & fight'::text)` query was evaluated against the `lexemes` column for each row. As we already know, only three rows matched the search criteria, and more than 4,000 rows were discarded (`Rows Removed by Filter: 4222`). Let's explore how to make these searches more efficient by using GIN and GiST indexes.

6.6.1 Using GIN indexes

The GIN index is the preferred index type for full-text search queries. In chapter 5, we explored how to use GIN in practice by indexing and optimizing searches over JSON data. Similarly, GIN is equally effective for full-text search queries, where it indexes `tsvector` lexemes instead of JSON fields and attributes. If we create a GIN index on a `tsvector` column, the index will store each unique lexeme in its own index entry, with each entry referencing table rows that contain the corresponding lexeme in their text documents.

Suppose we've already created a GIN index on the `lexemes` column of the `movies` table. Figure 6.1 shows a simple representation of the index with a subset of lexemes derived from the descriptions of the movies *Gladiator*, *Saints and Soldiers*, and *Braveheart*.

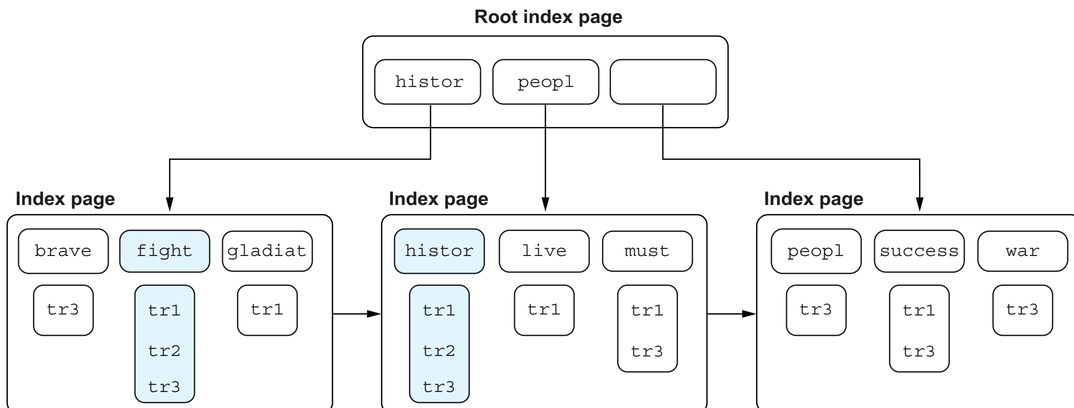


Figure 6.1 A sample GIN index showing that the "fight" and "histor" lexemes appear in three distinct movies

The index entries are stored in ascending order and distributed across several index pages. The search starts with the root index page, which contains three index entries:

- "histor" points to an index page containing items smaller than "histor."
- "peopl" points to another index page holding index items greater than or equal to "histor" and smaller than "peopl."
- The blank key points to a third index page that holds all other items greater than or equal to "peopl."

The search continues through the second-level pages, which store index entries referencing actual table rows. For instance, the index entries for the "fight" and "histor" lexemes reference three table rows (`tr1`, `tr2`, and `tr3`), showing that these lexemes appear in the descriptions of all three movies: *Gladiator*, *Saints and Soldiers*, and *Braveheart*.

> **NOTE** The GIN index stores only lexemes (normalized words), not their positional information within documents or assigned weight labels. This will require Postgres to retrieve such information from the corresponding table rows if any full-text search query needs it. If many of your full-text search queries rely on the positional information of lexemes, consider using the RUM index, which works similarly to GIN but also stores lexeme positions within the index entries. The RUM index is available as a Postgres extension.

Let's see how GIN works by creating a GIN index on the `lexemes` column using the following query.

Listing 6.16 Creating a GIN index over `tsvector` lexemes

```
CREATE INDEX idx_movie_lexemes_gin
ON omdb.movies
USING GIN (lexemes);
```

Once the index is created, let's execute the following query again to analyze its updated execution plan:

```
EXPLAIN (analyze, costs off)
SELECT id, name FROM omdb.movies
WHERE lexemes @@ to_tsquery('historical & fight');
```

The plan shows that Postgres now uses the GIN index, making the search hundreds of times faster (15.328 ms with a `Seq Scan` before the index was created, compared to 0.150 ms with the GIN index):

```
                                 QUERY PLAN
--------------------------------------------------------------------------
 Bitmap Heap Scan on movies (actual time=0.088..0.099 rows=3 loops=1)
   Recheck Cond: (lexemes @@ to_tsquery('historical & fight'::text))
   Heap Blocks: exact=3
   ->  Bitmap Index Scan on idx_movie_lexemes_gin
```

```
➥       (actual time=0.067..0.067 rows=3 loops=1)
        Index Cond: (lexemes @@ to_tsquery('historical & fight'::text))
 Planning Time: 0.283 ms
 Execution Time: 0.150 ms
(7 rows)
```

Postgres first uses the `Bitmap Index Scan` access method to quickly locate all index entries with lexemes matching the search condition (`Index Cond`). By the end of this phase, the database prepares a bitmap specifying table rows satisfying the search criteria. After that, Postgres proceeds with the `Bitmap Heap Scan` phase, visiting each row listed in the bitmap to retrieve the required information, such as the movie `id` and `name`.

6.6.2 Using GiST indexes

A GiST index is a flexible, balanced tree that is frequently used for indexing complex data types that support containment, proximity, and similarity check operations. With GiST, we can easily index geometric shapes, full-text search lexemes, network addresses, range-based types, and more.

The way GiST works depends on the data type it indexes. For instance, when indexing two-dimensional geometric points, GiST organizes them into a hierarchical structure. It starts by grouping points into large bounding rectangles, which are stored in the root pages of the index. These rectangles are then split into smaller ones and stored at the next level of the index. The process continues until GiST reaches the leaf nodes, which store the actual points. If Postgres needs to check whether a point exists in the index, it moves through the bounding rectangles that contain the point until it reaches its leaf node or determines that the point isn't in the index.

When GiST is used to index `tsvector` lexemes, Postgres creates a signature tree, where each index entry stores a *signature* representing a text document. The signature is a bit string with each bit set to either 0 or 1. It's constructed by combining the signatures of the document's lexemes using the bitwise OR operator.

A signature of an individual lexeme has only one bit set to 1, with the rest set to 0. Suppose that the signature length is chosen to be 1 byte (8 bits). Then the signatures of the lexemes might look like this:

- "histor"—`00000001`
- "film"—`10000000`
- "gladiat"—`01000000`

Once GiST assigns the signatures to lexemes, it can easily generate signatures for text documents containing those lexemes. For example, the signature for "Gladiator is a historical film" would be `11000001`, which is the result of applying the bitwise OR operation to the signatures of "histor," "film," and "gladiat."

NOTE The default signature length is 124 bytes, and it can be set to a different value when creating a GiST index. The maximum signature length is 2,024

bytes. Longer signatures make searches more precise (fewer false positives) when scanning the index, but this comes at the cost of a larger index size.

Figure 6.2 shows an example of a sample signature tree. As with any index type, the search starts at the root page and continues until the index reaches the leaf nodes, which are located at level 3. The leaf nodes store the signatures of actual text documents along with references to the corresponding table rows (note that table row references are not shown in the figure). The signatures at the upper levels 1 and 2 are formed by applying the bitwise OR operator to the signatures from the lower level.

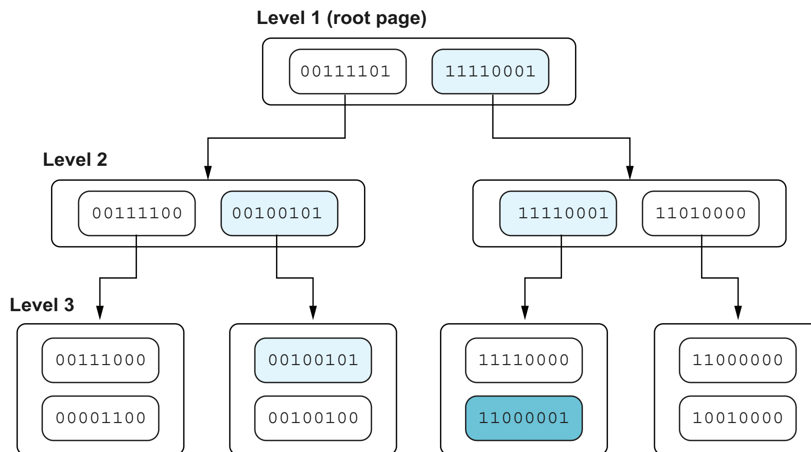


Figure 6.2 A sample GiST index highlighting all the index entries visited during a search for documents containing the "histor" lexeme (`00000001`)

Figure 6.2 also highlights the index entries visited during a search for documents containing the "histor" lexeme (`00000001`). As shown, both signatures in the root page have the eighth bit set to 1, causing the search to continue in both directions at level 2, where two other index entries meet this condition. Eventually, the search reaches level 3, where it finds two documents containing the "histor" lexeme. The first document's signature (`11000001`) corresponds to "Gladiator is a historical film," and the second signature (`00100101`) belongs to another document.

Let's now see how the GiST index works in practice by creating it on the `lexemes` column of our `movies` table using the following query.

Listing 6.17 Creating a GiST index over `tsvector` lexemes

```
CREATE INDEX idx_movie_lexemes_gist
ON omdb.movies
USING GIST (lexemes);
```

We'll also remove the previously created GIN index (`idx_movie_lexemes_gin`) to ensure that Postgres always selects the GiST index during our experiments:

```
DROP INDEX omdb.idx_movie_lexemes_gin;
```

Now, let's look again at the execution plan for a query that finds all movies with "historical" and "fight" in their descriptions:

```
EXPLAIN (analyze, costs off)
SELECT id, name FROM omdb.movies
WHERE lexemes @@ to_tsquery('historical & fight');
```

Postgres selects the following execution plan:

```
                                 QUERY PLAN
---------------------------------------------------------------------------
 Index Scan using idx_movie_lexemes_gist on movies
 ➥   (actual time=0.077..0.363 rows=3 loops=1)
   Index Cond: (lexemes @@ to_tsquery('historical & fight'::text))
 Planning Time: 0.985 ms
 Execution Time: 0.395 ms
(4 rows)
```

This time, the database traversed the created GiST index (`Index Scan using idx_movie_lexemes_gist`) to locate all documents satisfying the `Index Cond`.

Note that although the execution plan for GiST appears simpler than for GIN, which relies on the `Bitmap Index Scan` access method, the execution time with GIN was two times faster for the same query (0.150 ms with GIN versus 0.395 ms with GiST). One reason for this difference is that after GiST finds documents whose signatures match the search criteria, the database still needs to visit the corresponding table rows to confirm that the queried lexemes actually exist in the original documents. This recheck is necessary because multiple lexemes can have 1 bits set at the same positions, leading to signature collisions. The number of collisions can be reduced by increasing the signature size from the default 124 bytes up to the maximum of 2,024 bytes when creating the GiST index.

GIN vs. GiST for full-text search

Because GIN indexes actual lexemes, it usually performs faster, especially for queries that match a large number of documents. However, this comes at a cost: GIN indexes take longer to build and update due to their more complex structure. GiST indexes, on the other hand, consume less space and are faster to build and update, but index lookups can be slower because GiST relies on signature-based matching, which requires additional checks against table rows.

Overall, use GIN if you need fast lookups and can accept higher index maintenance costs. Use GiST if index size or fast build and update times matter most.

With that, we've learned how to perform full-text search in Postgres while working on the movie recommendation service. In chapter 8, we'll explore how to improve the search capabilities of the service further by using the generative AI capabilities of the database.

> **TIP** If the Postgres core full-text search capabilities we've explored are not enough for your use case, consider looking into the `pg_search` extension. It provides advanced features such as the BM25 (best matching 25) scoring algorithm for relevance ranking, which is widely used in modern search engines: https://github.com/paradedb/paradedb/tree/main/pg_search.

Summary

- Postgres comes with built-in full-text search capabilities for applications that need to analyze and query the contents of text documents.
- Like other full-text search engines, Postgres performs tokenization and normalization of original documents to produce `tsvector` lexemes.
- The database provides various functions and operators that let us easily query `tsvector` lexemes, with the ability to rank and highlight search results.
- GIN and GiST are the two primary index types used to speed up full-text search queries in Postgres.

Part 3

Extensions and the broader ecosystem

This part of the book explores what makes PostgreSQL one of a kind: its ecosystem of extensions and Postgres-compatible solutions. The motto "Just use Postgres!" emerged largely because of this rich extension ecosystem, which lets us use the database well beyond the scenarios covered earlier in the book.

We start with a quick overview of the Postgres extension ecosystem and learn how to install and use extensions. From there, we dive into three that are widely used: `pgvector` for generative AI and similarity search, TimescaleDB for time-series data, and PostGIS for geospatial applications. Finally, we explore how to implement message queues directly in Postgres, either from scratch using built-in capabilities or with the `pgmq` extension. After finishing this part of the book, we'll be ready to take advantage of the Postgres extensions ecosystem to design and build applications that handle generative AI, time-series, geospatial, and other types of workloads.

7 Postgres extensions

This chapter covers

- Understanding the basics of Postgres extensibility
- Installing and using Postgres extensions
- Exploring types of extensions useful for developers
- Postgres-compatible solutions

Postgres extensions are one of the main reasons the database has gained so much popularity and adoption. In fact, the motto "Just use Postgres" emerged largely due to its rich ecosystem of extensions, which allow us to use the database well beyond the use cases covered in the earlier chapters of the book. Let's do a quick overview of the ecosystem of Postgres extensions and practice using one of the extensions that comes preinstalled with the standard Postgres distribution.

7.1 The roots of Postgres extensibility

Postgres originated as a research project at the University of California, Berkeley, under the leadership of Michael Stonebraker, a computer scientist and Turing

Award winner specializing in database systems. During his Turing Award interview in 2017, Michael Stonebraker shared a story about why extensibility was one of the founding principles of Postgres. He mentioned that, in the early 1980s, there were dozens of research papers, all expressing a similar sentiment. A paper would state that relational databases were supposed to be terrific, but in reality, they didn't work well—or at all—for a particular scenario. The paper would then explain what was invented to fix the problem.

As a result, it became clear to Stonebraker that relational databases at the time didn't work well outside of business data processing (they didn't work for geographic management systems at all), and there ought to be a better way to enable an RDBMS for a particular use case rather than following a long list of suggestions from various research papers. Eventually, his research group set out to build Postgres, defining extensibility as one of the main requirements for the database. Since then, extensibility support in Postgres has evolved and has remained one of the database's key features, allowing us to adapt it for modern workloads and use cases.

7.2 Getting started with extensions

Postgres has hundreds of extensions serving a wide range of purposes. If core Postgres doesn't have a capability you're looking for, there's a good chance that an extension exists that does what you need.

The extension ecosystem is so broad and rich because Postgres is extensible by design and provides the following foundation, which significantly simplifies the integration process:

- *Catalog-driven operations*—Postgres stores information about tables, columns, data types, functions, and other database objects in system catalogs. These catalogs are similar to regular tables, except they store internal metadata used by the database to handle various operations and requests. An extension can access and modify system catalogs, adding new data types and functions to Postgres.
- *Database hooks*—The Postgres codebase defines several predefined hook points where extensions can inject custom logic. Hooks act like callback functions or event listeners that extensions can attach to. For example, extensions can intercept hooks related to query planning, execution, authentication, and more. When Postgres executes these hooks, extensions can process specific events or modify database behavior.
- *Dynamic loading of extension logic*—Depending on its complexity, an extension's logic can be written in pure SQL, a procedural language like PL/pgSQL, or a compiled language such as C or Rust. The dynamic loading is not required for the logic written in SQL or PL/pgSQL, as it is interpreted directly by the database engine and does not require external shared libraries. However, if the extension is written in a compiled language, Postgres loads the extension's shared library dynamically at runtime, eliminating the need to recompile the core database engine.

With this foundation, Postgres has already gained hundreds of extensions that introduce new capabilities, improve database behavior in certain scenarios, simplify administrative and development tasks, and offer much more beyond what's available in the core database engine.

Before we can start using a Postgres extension, it needs to be installed on the database server and then enabled for a particular database created with the `CREATE DATABASE` command. Some extensions have become so widespread and instrumental that they are already shipped with the standard Postgres distribution. These extensions are stored in a dedicated database directory and can be easily enabled when needed. Other extensions can be added to the database server in one of the following ways:

- *Extension registries and repositories*—Many extensions can be discovered and installed from extension repositories such as PGXN (https://pgxn.org).
- *Prebuilt container images*—Many extension creators provide Docker images that ship standard Postgres with the required extension, allowing for quick deployment.
- *Linux package managers*—Popular extensions can often be installed via a package manager like APT (Debian/Ubuntu) or DNF/YUM (RedHat/CentOS), making them available system-wide.
- *Installing from binaries or building from source*—If none of the other options work, extensions can be compiled from the source following the installation instructions provided by the extension maintainers. Plus, some extension providers can already offer binaries for particular operating systems.

7.2.1 Exploring available extensions

Postgres provides the `pg_available_extensions` view, which stores information about extensions available on the database server. Even if we don't add any specific extensions to Postgres, the database comes with some preinstalled extensions that we can view by querying `pg_available_extensions`.

For example, our Postgres instance from chapter 1 includes 45 bundled extensions:

```
SELECT count(*) FROM pg_available_extensions;

 count
-------
    45
(1 row)
```

> **NOTE** If you'd like to gain practical experience while reading the chapter, connect to your Postgres instance started in chapter 1 using the `docker exec -it postgres psql -U postgres` command.

We can query the `pg_available_extensions` view, which stores information about an extension's name, description, and its available and enabled versions:

```
SELECT * FROM pg_available_extensions
ORDER BY name
LIMIT 5;
```

Here's what those details look like for the first five extensions when the output is generated using the extended display mode of psql, enabled with the `\x on` meta-command:

```
-[ RECORD 1 ]-----+---------------------------------------------------
name              | amcheck
default_version   | 1.4
installed_version |
comment           | functions for verifying relation integrity
-[ RECORD 2 ]-----+---------------------------------------------------
name              | autoinc
default_version   | 1.0
installed_version |
comment           | functions for autoincrementing fields
-[ RECORD 3 ]-----+---------------------------------------------------
name              | bloom
default_version   | 1.0
installed_version |
comment           | bloom access method - signature file-based index
-[ RECORD 4 ]-----+---------------------------------------------------
name              | btree_gin
default_version   | 1.3
installed_version |
comment           | support for indexing common datatypes in GIN
-[ RECORD 5 ]-----+---------------------------------------------------
name              | btree_gist
default_version   | 1.7
installed_version |
comment           | support for indexing common datatypes in GiST
```

TIP Use the `\x on` meta-command in psql to display data in a format similar to the one from the previous output. This command enables expanded display mode, which shows each row vertically. It's helpful for results with many columns or large column values, making the data easier to read. To revert to the default tabular display mode, use the `\x off` command, which displays query results in a tabular format with rows and columns aligned horizontally.

The `default_version` column refers to the version of an extension available on the database server, which will be installed (enabled) by default unless a different version is explicitly specified. The `installed_version` column shows the version currently installed and enabled in the database we're connected to. If it is blank, the extension is available on the server but has not been enabled in the current database.

In our case, the information about installed extensions is provided for the `postgres` database because that's the database we're connected to. We can confirm this by executing the `\c` meta-command:

```
\c

You are now connected to database "postgres" as user "postgres".
```

If we'd like to see which extensions are already installed and enabled in the `postgres` database, we can query the `pg_available_extensions` view as follows:

```
SELECT * FROM pg_available_extensions
WHERE installed_version IS NOT NULL;
```

Postgres reports that there is one installed extension, `plpgsql`, which we used in chapter 2 while creating a custom stored procedure.

```
  name   | default_version | installed_version |         comment
---------+-----------------+-------------------+-------------------------
 plpgsql | 1.0             | 1.0               | PL/pgSQL procedural
         |                 |                   | language
(1 row)
```

TIP To find information about installed extensions, query the `pg_extension` view or execute the `\dx` meta-command in the psql tool.

Now, let's learn how to install and use extensions that are already available on the database server.

7.2.2 Installing and using extensions

The extension installation process consists of two steps:

1. Install the extension on the Postgres database server. As we've discussed, some extensions come bundled with a Postgres distribution, and others can be added via extension registries, package managers, or other methods. In the following chapters, we'll learn to use Docker container images that package Postgres with an extension enabling the database for a particular use case.
2. Install (or, more accurately, "enable") the extension for a specific database that an application is intended to work with. This could be the default `postgres` database (as in our case) or another one created using the `CREATE DATABASE` command.

TIP Refer to chapter 2 if you'd like to learn how to create and connect to custom databases created with the `CREATE DATABASE` command.

In this section, we'll explore the second step of the extension installation process by enabling and using the `pgcrypto` extension, which comes bundled with the Postgres distribution. The `pgcrypto` extension provides various cryptographic functions, allowing us to securely store and process sensitive data in the database.

ENABLING THE PGCRYPTO EXTENSION

Let's run the following query to check the version of `pgcrypto` available on our Postgres server running in the Docker container.

Listing 7.1 Checking the `pgcrypto` extension version

```
SELECT name, default_version, installed_version
FROM pg_available_extensions
WHERE name = 'pgcrypto';
```

According to the output, `pgcrypto` version 1.3 is available on the database server (`default_version`), but it hasn't been enabled for the postgres database we're connected to (`installed_version` is empty):

```
   name   | default_version | installed_version
----------+-----------------+-------------------
 pgcrypto | 1.3             |
(1 row)
```

Enabling the extension for the current database is straightforward. All we need to do is use the `CREATE EXTENSION` command as follows:

```
CREATE EXTENSION pgcrypto;
```

Once the extension is enabled, we can rerun the query from listing 7.1 to verify that `installed_version` is now set to 1.3:

```
   name   | default_version | installed_version
----------+-----------------+-------------------
 pgcrypto | 1.3             | 1.3
(1 row)
```

We can also query the `pg_extension` view, which tracks only the extensions enabled for the current database:

```
SELECT extname, extversion FROM pg_extension;
```

The output should be as follows:

```
 extname  | extversion
----------+------------
 plpgsql  | 1.0
 pgcrypto | 1.3
(2 rows)
```

Now, let's try out `pgcrypto` by using its cryptographic functions to securely store user passwords.

Using pgcrypto

Imagine that our application stores user account names and hashed passwords in the `accounts` table.

Listing 7.2 Creating an `accounts` table

```
CREATE TABLE accounts (
    id SERIAL PRIMARY KEY,
    username TEXT NOT NULL,
    password_hash TEXT NOT NULL
);
```

Next, suppose a new user creates an account. The application stores the account details in Postgres by executing the following query.

Listing 7.3 Encrypting the password for a new user

```
INSERT INTO accounts (username, password_hash)
VALUES ('ahamilton', crypt('SuperSecret123', gen_salt('bf')));
```

The query uses two functions from the `pgcrypto` extension to securely store the user's password (`SuperSecret123`):

- The `gen_salt()` function generates a random salt using the Blowfish (`bf`) algorithm. It needs to be executed for every new password to ensure that random salts are used for different passwords. Random salting is better than using a single fixed salt string because if the latter becomes available to bad actors, they could attempt to crack all passwords at once. With random salts, if a salt is exposed, only the password associated with that salt will be at risk.
- The `crypt()` function combines the raw user password with the generated salt and computes a secure hash. Because we use random salts, the function produces unique hashes even for identical passwords. The resulting hash stores the salt as part of its value, enabling secure password verification later.

If bad actors ever gain access to the database, they won't be able to see the actual passwords used by users:

```
SELECT * FROM accounts;

 id | username  |                      password_hash
----+-----------+-------------------------------------------------------
  1 | ahamilton | $2a$06$NzaPtzIucejO/LVefQRHuOAmW9OLGKZf..l70dv48.
  ➥ vO9p/kSx/Qi
(1 row)
```

Now, suppose that the user `ahamilton` returns to our application the next day and signs in via the authentication form. The user enters the username and raw password

(`SuperSecret123`), and the application runs the following query to verify whether the username/password combination exists in the database.

Listing 7.4 Authenticating a user with `pgcrypto`

```
SELECT username FROM accounts
WHERE username = 'ahamilton'
AND password_hash = crypt('SuperSecret123', password_hash);
```

This time, the application passes the raw password entered by the user along with the stored `password_hash` to the `crypt` function. The function extracts the salt from the stored `password_hash` and uses it to generate a hash for the provided password (`SuperSecret123`). If the calculated hash matches the one in the database, Postgres returns the username, indicating that the user has been successfully authenticated.

The output for the query looks as follows:

```
 username
-----------
 ahamilton
(1 row)
```

If the user misspells the password by entering `SuperSecret1243`, the application will execute the following query:

```
SELECT username FROM accounts
WHERE username = 'ahamilton'
AND password_hash = crypt('SuperSecret1243', password_hash);
```

Because the hashed value won't match the stored `password_hash`, the database will return an empty result, indicating that authentication has failed. The application can then display a message to the user that the password is incorrect.

```
 username
----------
(0 rows)
```

DISABLING PGCRYPTO

If we ever decide to disable `pgcrypto` or any other extension for a particular database, we can use the `DROP EXTENSION` command. Let's go ahead and remove `pgcrypto` from the current database we're connected to:

```
DROP EXTENSION pgcrypto;
```

After that, we can query the `pg_extension` view to confirm that `pgcrypto` is no longer listed among the enabled extensions:

```
SELECT extname, extversion FROM pg_extension;

 extname | extversion
---------+------------
 plpgsql | 1.0
(1 row)
```

Without `pgcrypto`, we can no longer use its cryptographic functions to handle user passwords. For instance, let's try to authenticate our user again:

```
SELECT username FROM accounts
WHERE username = 'ahamilton'
AND password_hash = crypt('SuperSecret123', password_hash);
```

Because the `pgcrypto` extension has been uninstalled, the query fails with the following error:

```
ERROR:  function crypt(unknown, text) does not exist
LINE 3: AND password_hash = crypt('SuperSecret123', password_hash);
                            ^
HINT:  No function matches the given name and argument types.
➥   You might need to add explicit type casts.
```

However, if we ever change our mind, we can easily reenable the `pgcrypto` extension using the following familiar command and continue securely handling user passwords in Postgres:

```
CREATE EXTENSION pgcrypto;
```

7.3 Essential extensions for developers

After learning how to install and use extensions, let's do a quick overview of some extensions that we, as developers, can take advantage of in our applications. Even though there is no official categorization of existing Postgres extensions, we can still classify them based on our own criteria. Let's define and discuss five groups of extensions that can help expand your developer toolbox.

The first group of extensions can be called "Postgres beyond relational" because they extend Postgres far beyond the traditional use cases relational databases are known for. As we've seen in previous chapters, core Postgres already supports JSON processing and full-text search, but that's just the tip of the iceberg. Most of its nonrelational capabilities come from its rich ecosystem of extensions. Here are a few examples:

- `pgvector`, `pg_ai`, and `pgvectorscale` enable Postgres for generative AI and other AI/ML workloads, effectively turning it into a vector database. We'll explore Postgres's gen AI capabilities in detail in chapter 8.

- `TimescaleDB` is a well-known extension that makes it easy to store and analyze large volumes of time-series and event data. We'll learn to use Postgres for time-series workloads in chapter 9.
- `PostGIS` transforms Postgres into a database capable of storing, indexing, and querying geospatial data. We dedicate chapter 10 to `PostGIS` and its use cases.
- `pgmq` allows Postgres to function as a lightweight message queue, capable of persisting and delivering application-specific messages and events. We'll explore message queuing in Postgres in chapter 11.
- `pg_duckdb` enables high-performance analytical workloads for Postgres. It does so by embedding DuckDB's columnar-vectorized storage engine.

The second category of extensions that we, as developers, can benefit from is "programming and procedural languages." In chapter 2, we learned how to create stored procedures using the PL/pgSQL procedural language. However, you don't need to be a PL/pgSQL expert to create custom database functions or procedures that run inside Postgres. In most cases, you'll likely find an extension that allows you to write database functions in your preferred programming language. Here are some examples for major languages:

- `PLV8` embeds the V8 JavaScript engine into Postgres, allowing us to write JavaScript-based database logic and call it from SQL. Additionally, we can use `PLV8` together with the `PgCompute` client-side library to execute JavaScript functions directly on the database from our application logic, bypassing SQL.
- `PL/Java`, `PL/Python`, and `PL/Rust` enable us to create database functions, procedures, and triggers using Java, Python, or Rust, respectively.

The third category of extensions falls under "connectors and foreign data wrappers" because they allow Postgres to connect to external data sources and query remote data transparently using standard SQL, as if it were stored in regular Postgres tables. With this type of extension, Postgres can act as a unified data layer, pulling data from various sources—some of which might not even support SQL natively. Here are a few examples from this category:

- `file_fdw` is a foreign data wrapper (FDW) that allows querying data from the file system.
- `postgres_fdw`, `mysql_fdw`, `oracle_fdw`, and `sqlite_fdw` let Postgres connect to and query other SQL databases, such as a remote Postgres server, MySQL, Oracle, or SQLite.
- `redis_fdw`, `parquet_s3_fdw`, and `kafka_fdw` enable Postgres to retrieve data from non-SQL data sources such as Redis, S3, and Kafka.

The fourth category of extensions, "query and performance optimization," helps analyze query execution statistics and improve performance when necessary. Here are a few extensions in this category:

- `pg_stat_statements` helps track the planning and execution statistics of all SQL statements executed by Postgres.
- `auto_explain` automatically logs execution plans for slow queries, making it easier to diagnose performance problems without manually running `EXPLAIN ANALYZE`.
- `hypopg` adds support for hypothetical indexes, allowing us to test different indexing strategies without creating real indexes.

The fifth category, "tools and utilities," includes extensions that simplify common Postgres tasks, automate operations, and enhance database usability. Here are a few examples:

- `pg_cron` is a simple cron-based scheduler that runs in Postgres, helping automate routine tasks associated with the database.
- `PostgreSQL Anonymizer` lets users anonymize or mask personal and sensitive data using techniques like dynamic masking and pseudonymization.
- `pgaudit` provides detailed session and object-level audit logging through Postgres's standard logging facility.
- `pg_partman` simplifies the creation and management of both time-based and number-based table partitions.

Although this list of categories and extensions is far from exhaustive, it highlights how significantly Postgres extensions push the boundaries of the database. This became possible because Postgres was designed to be extensible, striking the right balance between a relatively conservative core engine and an ecosystem of extensions that keep it at the forefront of innovation.

7.4 Postgres-compatible solutions

Postgres adoption and popularity have grown so rapidly that its communication protocol, capabilities, and source code are now being used by other solutions that don't fall under the category of extensions. These Postgres-compatible solutions introduce new capabilities that are neither available in Postgres nor provided by any of its extensions.

Postgres-compatible solutions typically address a specific class of problems that cannot be easily or effectively solved through an extension. In some cases, Postgres may lack the necessary hooks for a solution to inject its logic. In others, the architecture of the solution may differ so drastically from Postgres that it's not feasible to implement the solution as an extension. As a result, a solution is either built from scratch, supporting the Postgres wire-level protocol (communication protocol) and a subset of its features, or built on the Postgres source code with modifications to its original logic.

NOTE The Postgres wire-level protocol defines a set of message formats and commands used for communication between a Postgres client and server. Every SQL command sent by an application is transmitted and processed according to this protocol.

A distinct type of Postgres-compatible solution is databases that benefit from Postgres' growing adoption while also helping it reach new boundaries and use cases. For instance, Google Spanner and CockroachDB are distributed Postgres-compatible databases built from scratch with support for the Postgres wire-level protocol, data manipulation language (DML) and data definition language (DDL) syntax, and a subset of Postgres features. As a result, we can reuse some of the existing Postgres third-party tools and frameworks while building scalable and highly available applications on these databases.

Neon (a serverless database) and YugabyteDB (a distributed database) are examples of Postgres-compatible databases built on the Postgres source code with modifications and enhancements at the storage level. Both reuse most of the Postgres source code, allowing us to run Postgres applications as is without modifications or migrate with minimal effort. As a result, we can continue using most Postgres libraries, tools, and frameworks while building scalable and reliable applications on these databases.

Thus, if you can't solve a particular task or problem with Postgres and its extensions, consider exploring Postgres-compatible solutions. They might offer what you need while allowing you to continue using Postgres capabilities and applying your Postgres skills.

Summary

- Extensibility was one of the main requirements for Postgres.
- Postgres enables extensibility through catalog-driven operations, database hooks, and dynamic loading of external logic.
- Postgres extensions can be installed via extension registries/repositories, package managers, or prebuilt container images or built from source.
- Postgres ships dozens of extensions in its standard distribution.
- Application developers can use a wide range of extensions to go beyond relational workloads, query data from remote sources, simplify query optimization, and more.
- Postgres-compatible solutions address specific problems that cannot be easily or effectively solved through an extension. They either support the Postgres wire-level protocol and a subset of its features or are built on the Postgres source code with modifications to its original logic.

8 Postgres for generative AI

This chapter covers

- Exploring Postgres's capabilities for generative AI
- Using the `pgvector` extension to store and query vector embeddings
- Optimizing the similarity search with HNSW and IVFFlat indexes
- Implementing RAG with Postgres

Generative artificial intelligence (gen AI) uses specialized models to produce text, images, videos, and other types of data. These generative models are trained on vast amounts of data and can generate new content based on patterns learned during training. For example, a large language model (LLM) is trained on diverse text data that can produce coherent text in response to user prompts in a natural language such as English or Japanese. An LLM can answer questions, engage in conversation, and perform various tasks by trying to understand the intent behind the prompts.

Let's explore Postgres's capabilities for generative AI as we continue building the movie recommendation service that lets users find movies they like or might want

to watch. We'll learn to turn movie descriptions into vector embeddings, store them in Postgres, and use vector similarity search and retrieval-augmented generation (RAG) to provide users with more sophisticated and personalized recommendations.

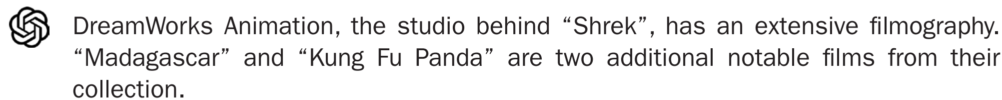
8.1 How to use Postgres with gen AI

Our movie recommendation service can use gen AI with Postgres in various ways. At a minimum, the service can store vector embeddings for movie descriptions in the database and let users find movies of interest by typing natural language prompts. Also, the service can use an LLM to create an AI agent that provides recommendations based on user preferences and the existing movie catalog stored in Postgres. Beyond that, such an agent can manage user watchlists by adding and removing movies based on user requests. However, before we learn to implement some of these capabilities, let's take a closer look at how LLMs and embedding models can be integrated with Postgres.

8.1.1 Postgres and LLMs

Because LLMs are trained on large volumes of text data, they are generally good at answering general-purpose questions across various topics. Imagine that a user visits the ChatGPT website and asks the following question:

ChatGPT is an example of a gen AI web application that relies on an underlying LLM to understand and answer user prompts. The application has several models at its disposal, and if the user selects the model GPT-4, a truncated version of the response might be as follows:



LLM providers typically offer access to their models via APIs. For example, our applications can interact with OpenAI's GPT models or Anthropic's Claude models through simple REST API calls. Additionally, free models like Meta's Llama and DeepSeek by High-Flyer can be installed and function in our private infrastructure.

With this capability, as an option, we can introduce a chatbot for our movie recommendation service that engages users in conversations about movies. The chatbot will interact with an LLM programmatically to fulfill its role.

However, LLMs in gen AI applications are not limited to chatbot-like use cases. Because they can understand intent from a user prompt, their capabilities go far beyond answering questions or engaging in conversations on specific topics.

LLMs can also use various tools, services, protocols, and APIs to perform tasks on our behalf. Figure 8.1 shows some example tools that an LLM can use.

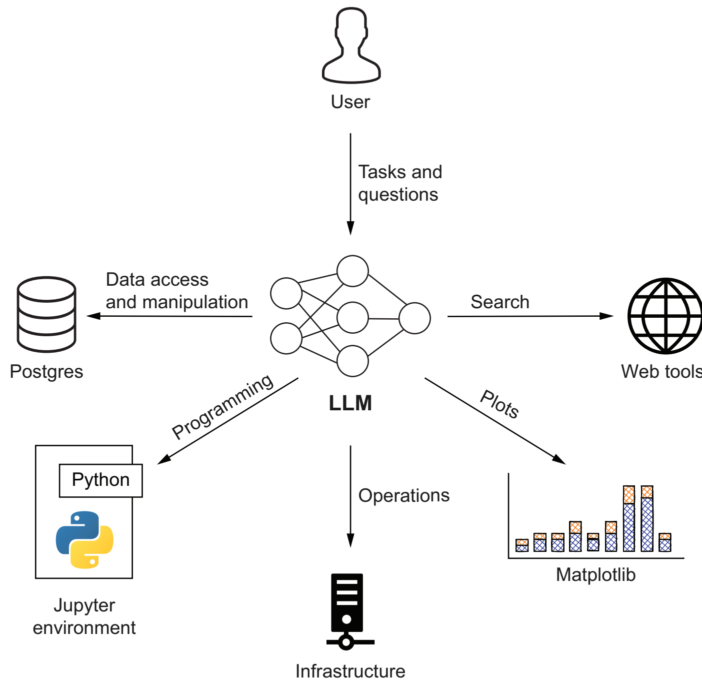


Figure 8.1 The LLM uses various tools to perform tasks following user prompts.

Imagine that a user sends a prompt to the LLM that requires it to use one or several listed tools to complete the task before it generates a response:

- *Web tools*—Every model has a *cutoff date*, which is the point in time when the data used for training was last gathered or updated. Because of this, an LLM's internal knowledge base can be outdated by several years, meaning it cannot answer questions about events that happened after the cutoff date. However, the model can retrieve the latest information from the internet using various web tools.
- *Matplotlib*—If we want the model to generate a plot or other visualization, it can produce a Python script that creates graphics using tools like Matplotlib.
- *Infrastructure*—The LLM can connect to our public cloud or on-premises infrastructure to spin up new application instances, run benchmarks, and perform other operational tasks.
- *Jupyter environment*—Not only can the model generate code in various programming languages and share it with us, but it can also write, test, and debug Python code on its own using a Jupyter-like environment. This enables the LLM to both create and validate application logic.
- *Postgres*—The LLM can access the database for multiple purposes. It may need to retrieve private data to assist with specific business tasks or expand its knowledge. The model can also connect to the database and manipulate data on our behalf.

In this chapter, we'll explore how to use an open source LLM running on our own laptops with Postgres to expand the model's knowledge base with RAG.

8.1.2 Postgres and embedding models

As we've learned, LLMs are a special class of gen AI models that take user questions or other text data as input and generate text as output. Before producing a response, LLMs may also use additional external tools and services to perform requested tasks.

However, LLMs are not the only type of AI model used in gen AI applications. Another widely used class of models is *embedding models*, which can take text or other data as input and generate high-dimensional numeric vectors called vector embeddings.

A *vector embedding* is a numeric representation of words, sentences, and other data that captures their meanings and relationships. Essentially, it is an *n*-dimensional coordinate representing a word or a longer sentence in *n*-dimensional space. For example, an embedding for the movie title *Kung Fu Panda* might be represented as a two-dimensional vector, such as `[-1.5,-0.7]`, which encodes some aspect of its meaning. This embedding can be easily plotted in two-dimensional space, as shown in figure 8.2.

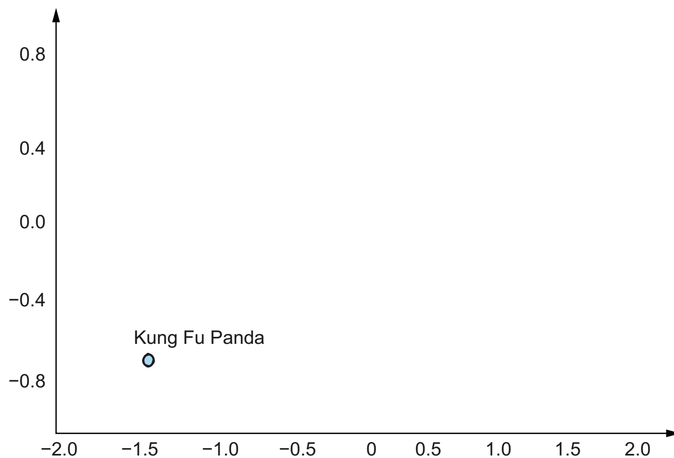


Figure 8.2 Positioning of the phrase "Kung Fu Panda" in a two-dimensional vector space

In practice, the embedding size typically ranges from hundreds to thousands of dimensions. Generally, the more dimensions an embedding model has, the better it captures a word's meaning in context, which is usually defined by the surrounding text. For instance, the word "panda" can refer to the black-and-white bear native to southwest China that eats bamboo or to the animated movie character who masters kung fu and loves dumplings. An embedding model operating in vector space with more dimensions can usually better capture such nuances, producing vector embeddings that encode the meaning of "panda" and other words in their diverse contexts.

In addition to capturing meanings, embedding models are trained to recognize relationships between words, phrases, and larger chunks of text data. For example, *Kung Fu Panda* is one of the movies produced by DreamWorks Animation, the same studio behind *Shrek* and *Madagascar*. As a result, the vector embeddings for these three movies can develop strong associations and form a distinct cluster in high-dimensional space. Figure 8.3 shows how this cluster might appear when projected onto two dimensions. The figure also shows that embeddings for *Toy Story*, *Finding Nemo*, and *Monsters, Inc.* form a separate cluster because they were produced by Pixar and Disney.

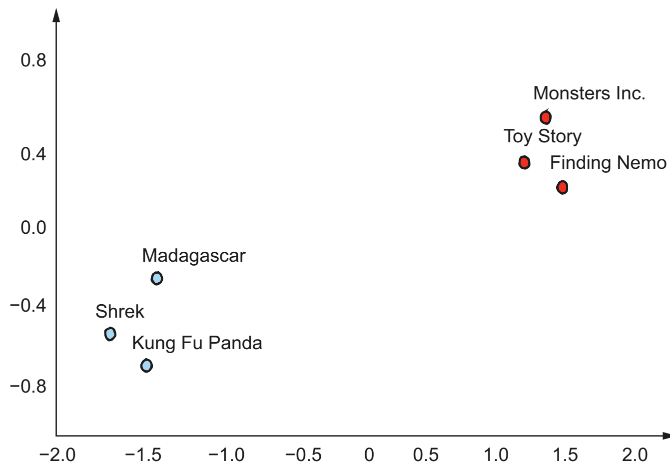


Figure 8.3 Clusters of movies related by animation studio

NOTE High-dimensional embedding models can encode different relationships and associations between movies. For example, embeddings may capture connections based on release year, budget, actors, or category (for example, animated versus horror). This allows movies to be clustered in high-dimensional space based on various contexts and criteria.

How does this relate to Postgres? First, we can generate embeddings for our private data stored in the database, as well as in other documents or data sources. Once generated, these embeddings can be stored in Postgres and used for vector similarity searches that allow us to find data that is related to a provided search term.

For example, if we generate embeddings for movie descriptions, our recommendation service can support the following use cases:

- *Semantic search*—Find movies that match a user-provided description.
- *Clustering*—Identify movies related by plot, genre, or other criteria.
- *Recommendations*—Suggest movies similar to those the user has recently watched.

- *RAG*—Retrieve relevant context for an LLM like GPT-4 from our private data so that the LLM can generate a more relevant response or perform some actions based on the internal knowledge base.

Let's learn how this works in practice by enabling vector similarity search for our movie recommendation service. The first step is to start Postgres with the `pgvector` extension.

8.2 Starting Postgres with pgvector

The `pgvector` extension provides Postgres with fundamental capabilities for storing and querying vector embeddings. It introduces the `vector` data type, enables operators for vector similarity searches, and provides specialized indexes for vectorized data. Essentially, `pgvector` turns Postgres into a vector database.

We'll explore `pgvector`'s capabilities gradually throughout the chapter. We begin by starting a Postgres instance with `pgvector` and enabling the extension for our movie recommendation service.

Like many other Postgres extensions, `pgvector` can be installed from Postgres extension repositories, Linux package managers, or binaries. Additionally, it's available as a Docker image that packages the extension with the standard Postgres distribution. We'll use this last option by running Postgres with `pgvector` in Docker.

Stopping the Postgres container used in previous chapters

In previous chapters, we used the Postgres container that doesn't include the `pgvector` extension. The container listens for client connections on port 5432, which we mapped directly to the same port on the host. This is done to allow clients and applications running from the host to connect to the database using its default port. If you still have any Postgres containers running, you need to stop them to free up port 5432.

Confirm that a Postgres container is up and running:

```
docker ps --filter "name=postgres" --filter "publish=5432"
```

The command might output a result similar to the following, indicating that a container is currently running:

```
IMAGE            STATUS         PORTS                      NAMES
postgres:latest  Up 50 minutes  0.0.0.0:5432->5432/tcp     <container-name>
```

Stop the container by providing its name in the `<container-name>` placeholder:

```
docker container stop <container-name>
```

Use the following command to start a Postgres container with `pgvector` on a Unix operating system such as Linux or macOS.

Listing 8.1 Starting Postgres with `pgvector` on Unix

```
docker volume create postgres-pgvector-volume

docker run --name postgres-pgvector \
    -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password \
    -p 5432:5432 \
    -v postgres-pgvector-volume:/var/lib/postgresql/data \
    -d pgvector/pgvector:0.8.0-pg17
```

If you're a Windows user, use the next command in PowerShell instead. If you use Command Prompt (CMD), replace each backtick (`) with a caret (^) at the end of each line.

Listing 8.2 Starting Postgres with `pgvector` on Windows

```
docker volume create postgres-pgvector-volume

docker run --name postgres-pgvector `
    -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password `
    -p 5432:5432 `
    -v postgres-pgvector-volume:/var/lib/postgresql/data `
    -d pgvector/pgvector:0.8.0-pg17
```

Both commands start the `postgres-pgvector` container using the `pgvector/pgvector:0.8.0-pg17` Docker image that includes the `pgvector` extension. The container listens for incoming connections on port `5432` and stores Postgres data in the Docker-managed volume named `postgres-pgvector-volume`.

NOTE The way we deploy Postgres in Docker works well for development and for exploring the database's capabilities. However, if you plan to use this deployment option in production, be sure to review security and other deployment best practices.

Once the container is started, connect to it using the psql client:

```
docker exec -it postgres-pgvector psql -U postgres
```

Confirm that the `pgvector` extension exists in the Postgres container by executing the following command.

Listing 8.3 Checking the available `pgvector` version

```
SELECT * FROM pg_available_extensions
WHERE name = 'vector';
```

The output should look similar to the following:

```
  name   | default_version | installed_version |          comment
--------+-----------------+-------------------+----------------------------
 vector | 0.8.0           |                   | vector data type and ivfflat
        |                 |                   | and hnsw access methods
(1 row)
```

Because the `installed_version` column is blank, it means the extension is not yet enabled for our database. Let's install it using the following `CREATE EXTENSION` command:

```
CREATE EXTENSION vector;
```

Finally, execute the query from listing 8.2 again to confirm that version 0.8.0 of `pgvector` is now installed for your database:

```
  name   | default_version | installed_version |           comment
--------+-----------------+-------------------+----------------------------
 vector | 0.8.0           | 0.8.0             | vector data type and ivfflat
        |                 |                   | and hnsw access methods
(1 row)
```

With that, we're ready to move on to the next step by generating vector embeddings for movie descriptions and storing them in the `vector` type provided by `pgvector`.

8.3 Generating embeddings

At a minimum, the embedding generation process for our private text data requires us to think about the following items:

1. *Target data*—The text for which we want to generate embeddings. It can come from various text documents, existing database records, or other sources. We should select data relevant to semantic search, RAG, and other gen AI use cases.
2. *Embedding model*—A model that converts text into vector embeddings. There are many options to choose from: proprietary models from OpenAI, Anthropic, and other providers that are available as a service, and open source models that we can run in our own infrastructure. Another key factor is the number of dimensions a model supports. Some models generate embeddings with hundreds of dimensions, whereas others support thousands. Generally, more dimensions lead to more accurate and relevant results in vector similarity searches, but they also require more memory, storage, and compute resources.
3. *Chunking*—The process of splitting large text into smaller chunks that can be fed into the embedding model. Chunks are usually measured in tokens (words, subwords, or symbols), and models often provide recommendations for chunk size. We can follow these recommendations or use other approaches, like splitting text by sentences or paragraphs.

4. *Vector database*—A database capable of storing generated embeddings and supporting vector similarity searches. It also needs to support specialized index types for embeddings that can speed up searches over this vectorized data.

Figure 8.4 shows the embedding generation process once we have decided on the target data, embedding model, and vector database.

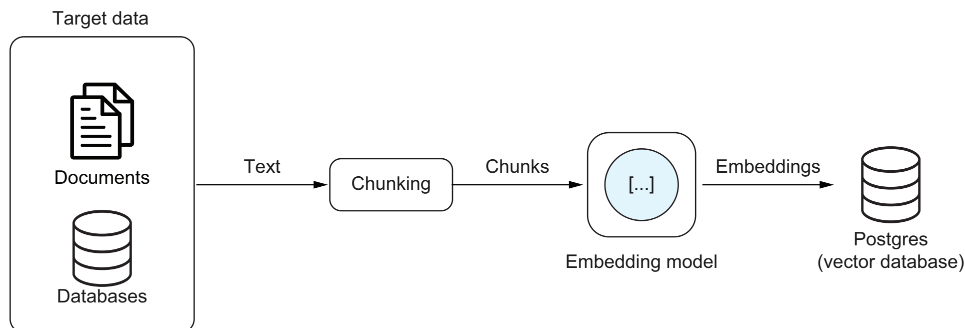


Figure 8.4 Generating and storing embeddings for target text data

Overall, we can create application logic that reads the target text data, chunks it if necessary, and sends the chunks to the embedding model. The model then outputs embeddings that we can store in Postgres, which can function as a vector database with the help of the `pgvector` extension. Let's see how we can apply this approach to generate embeddings for our movie recommendation service.

8.3.1 Generating embeddings for movies

Our new version of the movie recommendation service will store the movie catalog and embeddings for movie descriptions in the following table, which we'll create later when loading the final dataset:

```
omdb.movies (
    id BIGINT PRIMARY KEY,
    name TEXT NOT NULL,
    description TEXT NOT NULL,
    movie_embedding VECTOR(1024),
    release_date DATE,
    runtime INT,
    budget NUMERIC,
    revenue NUMERIC,
    vote_average NUMERIC,
    votes_count BIGINT
);
```

The table structure is similar to the one from chapter 6, with one key difference. In chapter 6, the table had the `lexemes` column of type `tsvector`, storing lexemes

generated from the `name` and `description` columns. This time, we introduce the `movie_embedding` column of type `vector` to store embeddings generated from the concatenation of the movie `name` and `description`.

The `vector` data type is provided by the `pgvector` extension and allows us to specify the number of dimensions for our vector embeddings. We'll be using the open source `mxbai-embed-large` embedding model, which we can run locally and use for free. The model produces 1,024-dimensional vectors, which is why the `movie_embedding` column is defined as `vector(1024)`. Figure 8.5 shows how the embedding generation process will look like for our movies dataset.

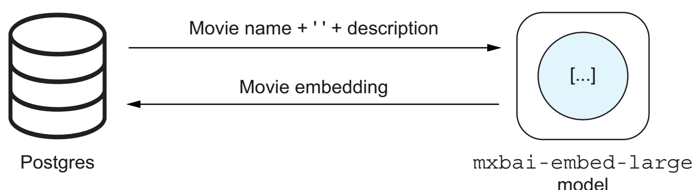


Figure 8.5 Generating embeddings for the movies dataset

In a nutshell, our application logic will retrieve the `name` and `description` of each movie from the database, concatenate them with a space separator (`name + ' ' + description`), and send the result to the `mxbai-embed-large` model. The model will generate 1,024-dimensional embeddings, which we'll store back in the database in the `movie_embedding` column. The following listing shows the pseudo-code for this process.

Listing 8.4 Pseudo-code for generating movie embeddings

```
# Fetch all movies from the database
movies = database.execute("SELECT id, name, description FROM omdb.movies")

# Iterate over each movie and generate the embedding
for each movie in movies:
    id = movie.id
    name = movie.name
    description = movie.description

    # Combine name and description for embedding
    combined_text = name + " " + description

    # Generate embedding using the model
    embedding = embedding_model.generate_embedding(combined_text)

    # Update the database with the generated embedding
    database.execute("
UPDATE omdb.movies SET movie_embedding = ? WHERE id = ?", embedding, id)
```

We don't apply any chunking strategy here because the combined movie description (`combined_text`) is small enough and already captures the essential information about each movie. Splitting it further into chunks could result in a loss of context, potentially affecting the accuracy and relevance of the vector similarity searches we'll use for our movie recommendation service.

The pseudo-code from listing 8.4 can be implemented in any major programming language that supports client-side drivers for Postgres and has gen AI libraries for working with embedding models. For example, JavaScript developers can use the node-postgres driver to interact with the database and the LangChain framework for embedding models. Java developers can use the Postgres JDBC driver for database interactions and the Spring AI framework for gen AI tasks, including embedding generation.

> **TIP** If you'd like to generate or update embeddings directly from the database, you can explore the `pgai` extension for Postgres. This extension allows you to work with embedding models purely in SQL, performing more gen AI–related tasks in the database.

Because this book doesn't assume prior knowledge of JavaScript, Java, Python, or any other programming language, we'll simply load the final dataset with the generated movie embeddings in the next section.

The final dataset was generated using a Python Jupyter notebook named embeddings_generator.ipynb, located in the book's GitHub repository: https://github.com/dmagda/just-use-postgres-book/tree/main/ai_samples. This repository also provides instructions on how to deploy the `mxbai-embed-large` model locally in an Ollama container, allowing you to generate embeddings from scratch or experiment with a different embedding model.

For now, let's proceed with loading the final dataset into Postgres. That way, you don't have to work with a programming language you might not be familiar with or run the Ollama container on your machine while you explore the basic capabilities of Postgres for gen AI applications.

8.3.2 Loading the final dataset into Postgres

If you'd like to gain practical experience while reading the chapter, follow these steps to preload the dataset into the Postgres container with the `pgvector` extension:

1. Make sure that the book's repository is already cloned on your machine:

```
git clone https://github.com/dmagda/just-use-postgres-book
```

2. Copy the movie dataset with the embeddings to your Postgres container:

```
cd just-use-postgres-book/
docker cp data/movie_pgvector/. postgres-pgvector:/home/.
```

3. Preload the dataset by connecting to the container and using the `\i` meta-command of psql to apply the copied SQL scripts:

```
docker exec -it postgres-pgvector psql -U postgres -c "\i /home/schema.sql"
docker exec -it postgres-pgvector psql -U postgres -c "\i /home/movies.sql"
docker exec -it postgres-pgvector psql -U postgres -c "\i /home/phrases.sql"
```

The tables are created under the `omdb` schema, and you can confirm this by connecting to Postgres and checking the created table with the `\dt omdb.*` command:

```
docker exec -it postgres-pgvector psql -U postgres
\dt omdb.*
```

The output is as follows:

```
               List of relations
 Schema |        Name        | Type  |  Owner
--------+--------------------+-------+----------
 omdb   | movies             | table | postgres
 omdb   | phrases_dictionary | table | postgres
(2 rows)
```

Next, confirm that slightly more than 4,200 movies are in the `omdb.movies` table:

```
SELECT count(*) FROM omdb.movies;

 count
-------
  4224
(1 row)
```

And make sure that every movie has an associated movie embedding:

```
SELECT count(*)
FROM omdb.movies
WHERE movie_embedding IS NOT NULL;
```

The output is as follows:

```
 count
-------
  4224
(1 row)
```

Finally, use the `\x on` command to switch to an extended display mode, and then execute the following query to check the embedding for the move *Jurassic Park*:

```
SELECT name, description, movie_embedding
FROM omdb.movies
WHERE id = 329;
```

The truncated output is as follows:

```
-[ RECORD 1 ]---+------------------------------------------------------------
name            | Jurassic Park

description     | After successfully pulling DNA from a petrified mosquito,
➥    a dinosaur park is created on an island. Yet before the dinosaur
➥    theme park opens to the public a group of scientists must tour the
➥    park and determine if it's ready for the public. The visit doesn't
➥    go as planned and the dinosaurs attempt to free themselves from
➥    the park.

movie_embedding | [0.008344619,0.0430743,0.04397416,-0.018851468,
➥    0.008879144,0.00067602337,-0.06890293,-0.002350036,
➥    0.039374635,0.02740375 truncated output
```

This 1,024-dimensional embedding was generated from the values in the `name` and `description` columns (`name + ' ' + description`). The output displays only the first 10 dimensions of the vector.

> **TIP** Use the `\x on` meta-command in psql to display data in a format similar to the one from the previous output. This command enables expanded display mode, which shows each row vertically. It's helpful for results with many columns or large column values, making the data easier to read. To revert to the default tabular display mode, use the `\x off` command, which displays query results in a tabular format with rows and columns aligned horizontally.

The preloaded dataset also has the `omdb.phrases_dictionary` table with the following structure:

```
\d omdb.phrases_dictionary

                 Table "omdb.phrases_dictionary"
      Column      |     Type     | Collation | Nullable | Default
------------------+--------------+-----------+----------+---------
 phrase           | text         |           | not null |
 phrase_embedding | vector(1024) |           |          |
```

The table stores several phrases that users might use to search for movie recommendations on our service. Each phrase has a corresponding vector embedding (`phrase_embedding`), generated using the same `mxbai-embed-large` model.

Finally, the following `omdb.get_embedding(input_phrase)` function is created during the dataset loading phase. It looks up the phrase in the `omdb.phrases_dictionary` table and returns its embedding if found. The complete implementation is as follows:

```
CREATE OR REPLACE FUNCTION omdb.get_embedding(input_phrase TEXT)
RETURNS VECTOR(1024) AS $$
DECLARE
    embedding VECTOR(1024);
BEGIN
```

```
    SELECT phrase_embedding INTO embedding
    FROM omdb.phrases_dictionary
    WHERE LOWER(phrase) = LOWER(input_phrase);

    IF NOT FOUND THEN
        RAISE EXCEPTION
            'The seach phrase does not exist in the dictionary table.';
    END IF;

    RETURN embedding;
END;
$$ LANGUAGE plpgsql;
```

We'll use this function and the pregenerated phrases in the next section as we learn how to perform vector similarity searches in Postgres.

8.4 Performing vector similarity search

Vector similarity search, also known as *nearest neighbor search* or *semantic search,* is a method for finding similar or related items in a dataset based on their vector representations. Gen AI applications often use vector similarity search to retrieve data related to a user's query. The user types a question in natural language, and the application generates an embedding for the query. It then performs a similarity search, comparing the query's embedding to those stored in the database. Finally, the application returns the most relevant data based on the query. Let's use the flow diagram in figure 8.6 to break this down for our movie recommendation service:

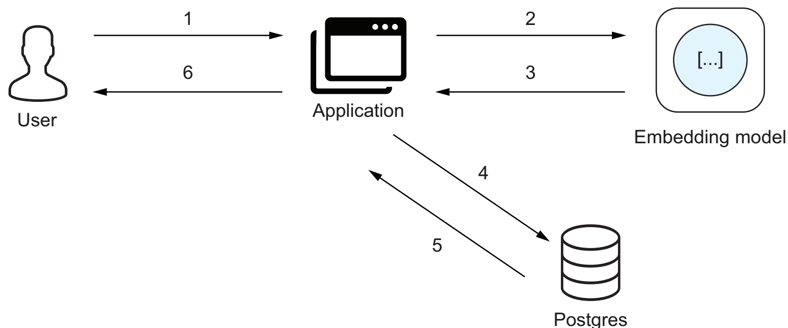


Figure 8.6 Performing vector similarity search with Postgres

1. The user connects to our movie recommendation service and enters a question or prompt, which is sent to the application backend.
2. The application backend receives the user's question and transforms it into a vector embedding using the same embedding model used for generating movie embeddings. In our case, this is the `mxbai-embed-large` model.

3. The model returns the embedding.
4. The application backend connects to Postgres and performs a vector similarity search, comparing the question's embedding to the movie embeddings stored in the `movie_embedding` column.
5. The database returns the movies most related to the user's question—these are the ones whose embeddings are closest to the question's embedding in the high-dimensional vector space.
6. The application backend formats and returns the list of recommendations to the user.

NOTE In our case, we don't generate embeddings dynamically for user prompts (steps 2 and 3 in figure 8.6). Instead, we use the `omdb.get_embedding` function, which looks up embeddings for preconfigured phrases in `omdb.phrases_dictionary`. However, the embeddings for those phrases were generated using the `mxbai-embed-large` model.

The vector similarity search relies on a distance function that, as the name suggests, calculates the distance between vectors. Vectors positioned close to each other in high-dimensional space will have the shortest distance between them and can be considered the most-related nearest neighbors.

The `pgvector` extension provides several distance operators, including cosine distance (`<=>`), Euclidean distance (`<->`), and inner product (`<#>`). Cosine distance is a common choice for gen AI applications that compare embeddings generated from text data, especially when the embeddings are L2-normalized—meaning their length is always 1. This method calculates the cosine of the angle between two vector embeddings:

- The smaller the angle between the embeddings, the smaller the cosine distance, meaning the embeddings are more related.
- The larger the angle, the greater the distance, indicating that the embeddings are less related.

For example, figure 8.7 illustrates how the angle might vary between more- and less-related vector embeddings in a two-dimensional space.

As we can see, angle *a* between the embeddings for the "May the force be with you" phrase and the "*Star Wars* …" description (where "…" indicates that the description continues) is smaller than angle *b* between the same phrase and the "*Jurassic Park* …" movie description. This is because the phrase appears frequently in *Star Wars* but never appears in *Jurassic Park*.

8.4.1 Using cosine distance for similarity search

Let's explore how vector similarity search works in action for our movie recommendation service. We can use cosine distance because the `mxbai-embed-large` embedding model produced normalized embeddings for the movies dataset.

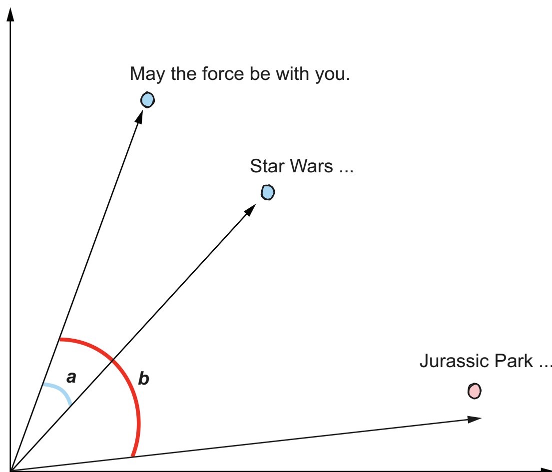


Figure 8.7 Angle difference between more- and less-related vector embeddings.

Imagine that a user visits our movie recommendation website and searches for movies related to the phrase "May the force be with you." Our application backend can quickly return the most relevant movie recommendations by executing the following query in Postgres.

Listing 8.5 Performing vector similarity search with cosine distance

```
SELECT id, name, description
FROM omdb.movies
ORDER BY movie_embedding <=> omdb.get_embedding('May the force be with you')
LIMIT 3;
```

The query execution flow works as follows:

- The query scans all movies, ordering them by the result of the `movie_embedding <=> omdb.get_embedding('May the force be with you')` operation. This operation calculates the cosine distance (`<=>`) between the embeddings stored in the `movie_embedding` column and the embedding corresponding to the search phrase.
- The `get_embedding(input_phrase)` function looks up the phrase in the `omdb.phrases_dictionary` table and returns its embedding if found.
- The result is ordered in ascending order, ensuring that movies with the smaller cosine distance appear first in the output.
- The query returns the top three most relevant movies (`LIMIT 3`), meaning their embeddings are the nearest neighbors to the search phrase's embedding in the 1,024-dimensional vector space.

The query output looks as follows:

```
-[ RECORD 1 ]-------------------------------------------------------------
id          | 1892
```

```
name        | Star Wars: Episode VI - Return of the Jedi
description | Return of the Jedi is the third and final chapter in the
➥   original wondrous Star Wars saga. Luke (Mark Hamill) must save
➥   Han Solo (Harrison Ford) from the clutches of the monstrous
➥   Jabba the Hut and bring down the newly reconstructed -and even
➥   more powerful- Death Star.

-[ RECORD 2 ]------------------------------------------------------------
id          | 204710
name        | The Acolyte
description | Star Wars series that takes viewers into a galaxy of shadowy
➥   secrets and emerging dark-side powers in the final days of the
➥   High Republic era.

-[ RECORD 3 ]------------------------------------------------------------
id          | 10840
name        | On Your Mark
description |
```

The first two movies in the output are indeed related to the search phrase, as they belong to the *Star Wars* saga. But neither the `name` nor the `description` column of these movies contains the phrase "May the force be with you." However, this phrase likely appeared frequently in *Star Wars*–related data used during the training of the `mxbai-embed-large` embedding model. As a result, the embeddings of the phrase and the first two movies from the output show a strong association.

The third movie in the output doesn't seem relevant to the search phrase, highlighting two important points:

- Like LLMs, embedding models are prone to *hallucination*, meaning they can generate inaccurate results.
- The effect of hallucination can be minimized by providing more context during embedding generation.

In this case, the problem with the third movie is that its `description` column is empty, meaning its embedding was generated solely from the title `On Your Mark`. This lack of context makes it difficult for the model to establish a meaningful connection between the movie and the data the model was trained on.

Vector similarity search in the application layer

We use the `omdb.get_embedding` function that retrieves embeddings from a predefined phrase dictionary so that we can explore vector similarity search in Postgres with minimal effort—without running an embedding model or implementing logic in the application layer. However, when implementing vector similarity search in an application, the pseudo-code can be as simple as this:

```
# Generating an embedding for the phrase
embedding = embedding_model.generate_embedding("May the force be with you")
```

(continued)

```
# Creating a SQL query
query = query.create(
   "SELECT id, name, description FROM omdb.movies
   ORDER BY movie_embedding <=> ::phrase_embedding
   LIMIT 3;")

# Passing the phrase's embedding as a parameter
query.set_param("phrase_embedding", embedding)

# Performing the vector similarity search
movies = database.execute(query)
```

For a full implementation in Python, refer to the ai_samples/similarity_search.ipynb Jupyter notebook from the book's GitHub project.

8.4.2 Changing the search phrase for better results

Let's see how the quality and accuracy of the output improve when the user refines the search phrase to be more specific. First, let's execute the following query to see the cosine distance between the embeddings of the phrase "May the force be with you" and the movies returned in the output.

Listing 8.6 Adding the cosine distance to the query output

```
WITH phrase AS (
  SELECT omdb.get_embedding('May the force be with you') AS embedding
)
SELECT id, name, description, m.movie_embedding <=> p.embedding AS distance
FROM omdb.movies m CROSS JOIN phrase p
ORDER BY distance LIMIT 3;
```

The query evaluates the common table expression (CTE) named `phrase`, which returns the `embedding` for the search phrase. This embedding from the CTE is then used to calculate the cosine distance for each movie record (`m.movie_embedding <=> p.embedding AS distance`), and the query produces the following output:

```
-[ RECORD 1 ]------------------------------------------------------------
id          | 1892
name        | Star Wars: Episode VI - Return of the Jedi
description | Return of the Jedi is the third and final chapter in the
 ➥  original wondrous Star Wars saga. Luke (Mark Hamill) must save
 ➥  Han Solo (Harrison Ford) from the clutches of the monstrous
 ➥  Jabba the Hut and bring down the newly reconstructed -and even
 ➥  more powerful- Death Star.
distance    | 0.39259877160780154

-[ RECORD 2 ]------------------------------------------------------------
```

```
id          | 204710
name        | The Acolyte
description | Star Wars series that takes viewers into a galaxy of
➥   shadowy secrets and emerging dark-side powers in the final days
➥   of the High Republic era.
distance    | 0.4038085330102721

-[ RECORD 3 ]--------------------------------------------------------
id          | 10840
name        | On Your Mark
description |
distance    | 0.4116991022170934
```

The cosine distances between our embeddings ranges from 0 to 1. The closer the distance is to 0, the more related a movie is to the search phrase. In this case, the cosine distance between the search phrase and the most relevant movie is approximately 0.39.

Now, let's change the search phrase to "A movie about a Jedi who fights against the dark side of the force" and execute the following query.

Listing 8.7 Changing the search phrase to improve result accuracy

```
WITH phrase AS (
  SELECT omdb.get_embedding(
    'A movie about a Jedi who fights against the dark side of the force')
     AS embedding
)
SELECT id, name, description, m.movie_embedding <=> p.embedding AS distance
FROM omdb.movies m CROSS JOIN phrase p
ORDER BY distance LIMIT 3;
```

This time, Postgres returns results that include only movies from the *Star Wars* saga, demonstrating how a more specific search phrase can improve the result of the vector similarity search:

```
-[ RECORD 1 ]---------------------------------------------------------
id          | 1892
name        | Star Wars: Episode VI - Return of the Jedi
description | Return of the Jedi is the third and final chapter in the
➥   original wondrous Star Wars saga. Luke (Mark Hamill) must save
➥   Han Solo (Harrison Ford) from the clutches of the monstrous
➥   Jabba the Hut and bring down the newly reconstructed -and even
➥   more powerful- Death Star.
distance    | 0.2742586520642718

-[ RECORD 2 ]---------------------------------------------------------
id          | 204710
name        | The Acolyte
description | Star Wars series that takes viewers into a galaxy of shadowy
➥   secrets and emerging dark-side powers in the final days of the
➥   High Republic era.
distance    | 0.3004889488218425
```

```
-[ RECORD 3 ]-----------------------------------------------------------
id          | 1891
name        | Star Wars: Episode V - The Empire Strikes Back
description | The Empire Strikes Back is considered the most morally
➥     and emotionally complex of the original Star Wars trilogy,
➥     continuing creator George Lucas's epic saga where Star Wars:
➥     Episode IV - A New Hope left off. Masterful storytelling weaves
➥     together multiple, archetypal plotlines that pit Vader against Han
➥     and Leia as he desperately attempts to capture Luke for political
➥     and personal reasons.
distance    | 0.31210668375643624
```

Now the cosine distance between the search phrase and the most relevant movie is around 0.27, which is significantly closer than 0.39, the distance from the previous search phrase ("May the force be with you") to its most related movie.

> **TIP** If you'd like to experiment with different custom phrases, run the ai_samples/similarity_search.ipynb Jupyter notebook from the book's GitHub project. The notebook also includes instructions on how to start the embedding model locally using Ollama.

8.5 Indexing embeddings

After learning how to generate embeddings for target data and perform similarity search, we're ready to explore how to make the search more efficient using specialized index types for vector data. Let's start by having a look at the current execution plan Postgres selects for similarity search queries. Execute the `\x off` command to have the psql tool show output in the default tabular format, and then run the following query, which shows a plan for a typical query in our movie recommendation service.

Listing 8.8 Execution plan for a similarity search query

```
EXPLAIN (analyze, costs off)
SELECT id, name, description
FROM omdb.movies
ORDER BY movie_embedding <=>
  omdb.get_embedding('May the force be with you')
LIMIT 3;
```

The database selects the following execution plan:

```
                              QUERY PLAN
---------------------------------------------------------------------
 Limit (actual time=52.421..52.422 rows=3 loops=1)
   ->  Sort (actual time=52.420..52.420 rows=3 loops=1)
         Sort Key: ((movie_embedding <=>
➥       omdb.get_embedding('May the force be with you'::text)))
         Sort Method: top-N heapsort  Memory: 27kB
         ->  Seq Scan on movies
➥            (actual time=0.803..51.398 rows=4224 loops=1)
```

```
 Planning Time: 0.694 ms
 Execution Time: 52.567 ms
(7 rows)
```

According to the plan, the query is executed as follows:

- Postgres performs a full table scan (`Seq Scan`), processing more than 4,000 records (`rows=4224`).
- It sorts the retrieved records based on the `Sort Key`, which calculates the cosine distance between a movie's embedding and the embedding of the search phrase.
- The database returns the top three results (`rows=3` in the `LIMIT` node).

Even though the execution plan appears simple, the vector similarity search is far from a lightweight operation.

In our case, each embedding is a 1,024-dimensional array, with each dimension storing a 4-byte floating-point number. This results in an embedding size of (1,024 × 4 bytes) + 8 bytes (header) = 4,104 bytes (4 KB).

As a result, every time a user searches for movie recommendations, the database must scan the entire movie catalog, compute the cosine distance for large 4 KB arrays, and sort the result by distance. Although this approach works for small datasets, it won't scale as the data volume grows or the number of user queries increases.

Because our movie recommendation service continues to grow in popularity, we need to ensure that similarity search over movie embeddings remains fast and efficient. One way to achieve this is by creating an index on the vector embeddings.

At the time of writing, the `pgvector` extension supports two index types for vector data: inverted file with flat compression (IVFFlat) and hierarchical navigable small world (HNSW). In addition, the `pgvectorscale` extension supports another index type called StreamingDiskANN. So, there are already several options to choose from for gen AI applications running on Postgres.

But regardless of the index type we choose, it's important to keep in mind that these indexes behave differently from B-tree, GIN, or GiST indexes, which we explored in earlier chapters. Indexes for vector data return approximate results, meaning the outcome of the index search might not always be 100% accurate.

For example, if we create a B-tree index on the `revenue` column of the `numeric` type and use it to find all movies where `revenue > 1000000`, the index will return all matching rows—no exceptions.

In contrast, if we create an IVFFlat or HNSW index on the `movie_embedding` column and use it to find movies related to the phrase "May the force be with you," the index might not return the most accurate result. That's because it doesn't compare the embedding of the search phrase against all embeddings in the dataset.

Indexes for vector data perform approximate nearest neighbor (ANN) search by comparing the query embedding to only a subset of embeddings. As a result, when indexing vector data, we need to pay attention not only to the performance and size

characteristics of an index but also to its *recall*—a metric that measures how many of the retrieved nearest embeddings are actually the true nearest neighbors.

Recall is typically measured on a scale from 0 to 1, where 1 represents perfect recall (the most accurate result). If Postgres performs a full table scan—as in the query from listing 8.8—the recall is expected to be 1, as long as the database performs an exact nearest neighbor (ENN) search by comparing the search phrase's embedding against all embeddings in the dataset and returning true nearest neighbors. With that in mind, let's explore how to use the IVFFlat and HNSW index types from the `pgvector` extension to optimize similarity search in our movie recommendation service.

8.5.1 Using IVFFlat indexes

The IVFFlat index works by dividing vector embeddings into lists, where each list stores embeddings that are similar to each other. These lists are also referred to as *clusters*.

Each list has a *centroid*, which is a data point representing the center of the cluster. The IVFFlat algorithm computes the centroids by applying the k-means clustering algorithm to the embeddings. Once the centroids are calculated, the IVFFlat index assigns each embedding from the dataset to the list whose centroid is closest to that embedding.

Figure 8.8 shows how a sample IVFFlat index might have looked for a subset of our movie embeddings in a two-dimensional space. As we can see, each list in the index represents a cluster of related movies. For instance, list 2 stores animated movies produced by DreamWorks, whereas movies in list 4 belong to a series of superhero films produced by Marvel Studios.

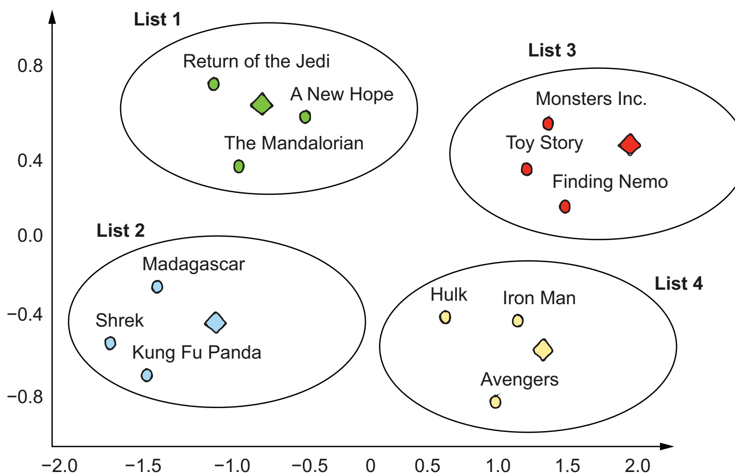


Figure 8.8 Sample IVFFlat index with movie embeddings

We use the rhombus (diamond) icon to display the centroids of the lists. The positioning of centroids is used not only during index creation—when embeddings are

assigned to one of the lists based on their proximity to a centroid—but also during vector similarity search.

For instance, if our application searches for movies related to the phrase "May the force be with you," Postgres will first calculate the distance between the embedding of the phrase and every centroid. After that, it will continue the similarity search within the list whose centroid is closest to the phrase embedding. In the case of our search phrase, the centroid of list 1, which stores *Star Wars* movies, will be the closest, so Postgres will perform the similarity search over the embeddings in that list, skipping the others.

Creating and using the index

Now, let's optimize vector similarity search for our movie recommendation service by creating the IVFFlat index shown next.

Listing 8.9 Creating an IVFFlat index

```
CREATE INDEX movie_embeddings_ivfflat_idx
ON omdb.movies
USING ivfflat (movie_embedding vector_cosine_ops) WITH (lists = 5);
```

This statement creates the index on the `movie_embedding` column using the cosine distance operator (`vector_cosine_ops`). Cosine distance is used during index creation to distribute embeddings across the lists, and the same distance calculation method must be used later during vector similarity search. If our application queries start using a distance operator different from cosine distance (`<=>`), Postgres won't use the created IVFFlat index.

The statement also defines the `WITH (lists = 5)` clause, which instructs Postgres to create the index with five lists. The `pgvector` documentation (https://github.com/pgvector/pgvector) suggests choosing the number of lists based on the following rule:

> *Choose an appropriate number of lists - a good place to start is* `rows / 1000` *for up to 1M rows and* `sqrt(rows)` *for over 1M rows.*

Once the index is created, let's check whether there are any changes in the execution plan of the following query:

```
EXPLAIN (analyze, costs off)
SELECT id, name
FROM omdb.movies
ORDER BY movie_embedding <=>
  omdb.get_embedding('May the force be with you')
LIMIT 3;
```

According to the execution plan, Postgres ignores the created index and continues with a full table scan, visiting every embedding in the movie dataset:

```
                                 QUERY PLAN
-------------------------------------------------------------------------------
 Limit (actual time=52.598..52.599 rows=3 loops=1)
   ->  Sort (actual time=52.596..52.596 rows=3 loops=1)
         Sort Key: ((movie_embedding <=>
➥        omdb.get_embedding('May the force be with you'::text)))
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Seq Scan on movies
➥            (actual time=0.635..51.893 rows=4224 loops=1)
 Planning Time: 0.477 ms
 Execution Time: 52.764 ms
(7 rows)
```

Even though the query uses cosine distance (<=>), which matches the index definition, and includes `ORDER BY` and `LIMIT` clauses to return a subset of movies, the Postgres planner still ignores the index due to the call to the `omdb.get_embedding('May the force be with you')` function. At the time of writing, the planner expects simpler queries that don't call functions like `omdb.get_embedding`.

However, if we use the following query, which replaces the `omdb.get_embedding` function with a sub-select that queries the `omdb.phrases_dictionary` table directly, the database starts taking advantage of the index.

Listing 8.10 Checking the execution plan with an IVFFlat index

```
EXPLAIN (analyze, costs off)
SELECT id, name, description
FROM omdb.movies
ORDER BY movie_embedding <=>
    (SELECT phrase_embedding
    FROM omdb.phrases_dictionary
    WHERE phrase = 'May the force be with you')
LIMIT 3;
```

This time, Postgres chooses the following execution plan:

```
                                 QUERY PLAN
------------------------------------------------------------------------------
 Limit (actual time=1.408..1.444 rows=3 loops=1)
   InitPlan 1
     ->  Seq Scan on phrases_dictionary
➥          (actual time=0.022..0.023 rows=1 loops=1)
           Filter: (phrase = 'May the force be with you'::text)
           Rows Removed by Filter: 3
   ->  Index Scan using movie_embeddings_ivfflat_idx on movies
➥        (actual time=1.400..1.434 rows=3 loops=1)
         Order By: (movie_embedding <=> (InitPlan 1).col1)
 Planning Time: 0.192 ms
 Execution Time: 1.492 ms
```

The query execution flow looks as follows:

- The database performs a full table scan (`Seq Scan`) on `phrases_dictionary` to find a pregenerated embedding for the search phrase.
- Postgres then performs an `Index Scan` using the `movie_embeddings_ivfflat_idx` index to locate movies whose embeddings are closest to the embedding of the search phrase.

The execution time for the similarity search improves significantly with the index: ~1 ms with the index versus ~50 ms when the database had to perform a full table scan. Note that the actual latency numbers may vary on your machine, but overall, you can expect the latency of an `Index Scan` to be much lower than that of the sequential full table scan.

No changes are needed in the application layer

If our application performs similarity search using logic comparable to the following pseudo-code, then Postgres will take advantage of the created IVFFlat index without requiring any changes on the application side:

```
# Generating an embedding for the phrase
embedding = embedding_model.generate_embedding("May the force be with you")

# Creating a SQL query
query = query.create(
   "SELECT id, name, description FROM omdb.movies
   ORDER BY movie_embedding <=> ::phrase_embedding
   LIMIT 3;")

# Passing the phrase's embedding as a parameter
query.set_param("phrase_embedding", embedding)

# Performing the vector similarity search
movies = database.execute(query)
```

As we can see, the application uses an embedding library to generate the embedding for the search phrase, which is then passed as a query argument to Postgres. The database planner can easily recognize that the IVFFlat index can be used to speed up the search for this application query.

Improving index recall during search

Even though we managed to optimize similarity search with the IVFFlat index, we have to remember that it performs an ANN search—visiting movie embeddings only from the list whose centroid is closest to the embedding of the search phrase. This certainly improves search performance, because the database scans only a subset of the data, but it can negatively affect the recall metric of the search if the true nearest neighbors are located in other lists that were skipped during the index scan.

Let's look at the result of the query that uses the index to find the most relevant movies for the phrase:

```
SELECT id, name
FROM omdb.movies
ORDER BY movie_embedding <=>
    (SELECT phrase_embedding
    FROM omdb.phrases_dictionary
    WHERE phrase = 'May the force be with you')
LIMIT 3;
```

The database might produce the following output:

```
  id   |           name
-------+-------------------------
 10840 | On Your Mark
  9956 | The Brave
 10268 | Je vous trouve très beau
(3 rows)
```

The current recall of the index is low—none of the returned movies has anything to do with the *Star Wars* saga. This suggests that the index performed the search within a list that didn't include embeddings associated with *Star Wars*.

> **NOTE** The results of queries using the IVFFlat index might differ on your end from what's shown in the book. This non-determinism typically stems from randomization applied during index build time, meaning that Postgres doesn't guarantee producing exactly the same index structure each time the index is built.

To improve the recall of the IVFFlat index, we can use the `ivfflat.probes` parameter, which defines how many lists should be visited during the index scan. For instance, the following query template shows how to adjust the number of probes for a specific query by using the `SET LOCAL` statement in a transaction:

```
BEGIN;
SET LOCAL ivfflat.probes = 3;
SELECT…
COMMIT;
```

Let's see how the recall improves for the following query where we set the number of probes to 2, requiring Postgres to visit two IVFFlat lists whose centroids are closest to the embedding of the search phrase.

Listing 8.11 Increasing the number of probes for an IVFFlat index

```
BEGIN;

SET LOCAL ivfflat.probes = 2;

SELECT id, name
FROM omdb.movies
```

```
ORDER BY movie_embedding <=>
    (SELECT phrase_embedding
    FROM omdb.phrases_dictionary
    WHERE phrase = 'May the force be with you')
LIMIT 3;

COMMIT;
```

This time, the query returns the three most relevant movies from our private dataset. The result is similar to what we saw in listing 8.5, when the database performed an ENN search (full table scan) by visiting every movie embedding in the table.

```
   id   |                    name
--------+-------------------------------------------
   1892 | Star Wars: Episode VI - Return of the Jedi
 204710 | The Acolyte
  10840 | On Your Mark
(3 rows)
```

However, the improved recall comes at the cost of search performance, which increases slightly because the index now needs to visit two lists of embeddings. Let's take a look at the new execution plan:

```
BEGIN;

SET LOCAL ivfflat.probes = 2;

EXPLAIN (analyze, costs off)
SELECT id, name
FROM omdb.movies
ORDER BY movie_embedding <=>
    (SELECT phrase_embedding
    FROM omdb.phrases_dictionary
    WHERE phrase = 'May the force be with you')
LIMIT 3;

COMMIT;
```

In our case, the execution time with two probes (~2 ms) remained comparable to the execution time with one probe (~1 ms), which is negligible given the substantially improved recall:

```
                               QUERY PLAN
------------------------------------------------------------------------
 Limit (actual time=2.747..2.813 rows=3 loops=1)
   InitPlan 1
     ->  Seq Scan on phrases_dictionary
➥         (actual time=0.033..0.035 rows=1 loops=1)
           Filter: (phrase = 'May the force be with you'::text)
           Rows Removed by Filter: 3
   ->  Index Scan using movie_embeddings_ivfflat_idx on movies
```

```
➥        (actual time=2.744..2.807 rows=3 loops=1)
         Order By: (movie_embedding <=> (InitPlan 1).col1)
 Planning Time: 0.402 ms
 Execution Time: 2.895 ms
(9 rows)
```

NOTE The number of probes required for good recall can vary depending on data volume, the number of IVFFlat lists, and the specifics of a query using the similarity search. Be sure to test your queries thoroughly to find the right balance between recall and the performance characteristics of the index.

The last thing to keep in mind about the IVFFlat index is that it works best when it's built on an existing dataset with embeddings and when that dataset doesn't change substantially over time. That's because the centroids of the index lists are calculated once during index creation and are not updated afterward. If the dataset grows significantly or is frequently updated, this can affect the index's recall characteristics. New embeddings will continue to be assigned to index lists based on their distance from the original centroids, which may no longer represent the optimal distribution.

If the dataset does change substantially and recall is affected, we can re-create the IVFFlat index from scratch. An alternative option is to use the HNSW index, which is more flexible and can even be created on an empty table. Let's explore how to use that index next.

8.5.2 Using the HNSW index

The HNSW index creates a multilayer graph with vector embeddings, where each node (vertex) represents a vector and edges connect similar vectors based on proximity. Higher layers of the graph contain a sparse subset of nodes, whereas the lower layers get denser with more interconnected nodes.

Figure 8.9 shows how a sample HNSW index might look for a subset of our movie embeddings in a two-dimensional space. As shown, the sample graph has three layers. The third layer (highest) is sparse, containing only a few interconnected embeddings. The lower layers become progressively denser, with more nodes connected to each other. Additionally, any node that appears in a higher layer also appears in all lower layers. For example, the node representing the movie *A New Hope* first appears in layer 3, so it also exists in layer 2 and layer 1.

The search through the index always starts at the highest layer (layer 3) and progressively moves down to the lowest layer (layer 1), traversing only the nodes closest to the embedding of the search phrase. For instance, if our application searches for movies related to the phrase "May the force be with you," Postgres begins the search at the uppermost layer 3, calculating the distance between the embedding of the search phrase and the four movie embeddings present in this layer: *Iron Man*, *Avengers*, *A New Hope*, and *The Mandalorian*. It determines that the embeddings for *A New Hope* and *The Mandalorian* are the closest (most related) and continues the search in layer 2, but only

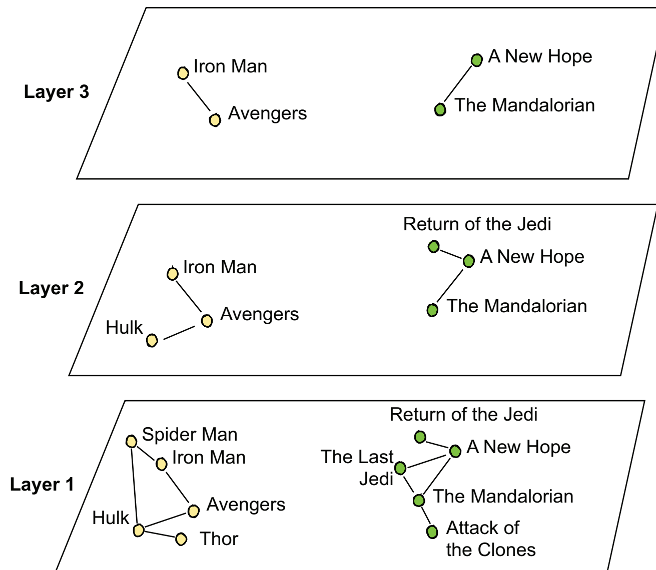


Figure 8.9 Sample HNSW index with movie embeddings

within the area of the graph containing *Star Wars* movies. The nodes for Marvel movies are no longer visited in layer 2, making the search more efficient.

Finally, the algorithm moves from layer 2 to the last layer 1, where it discovers additional embeddings for *Star Wars* movies. It completes the search by preparing a final list of the most relevant results.

CREATING AND USING THE INDEX

Now, let's optimize vector similarity search for our movie recommendation service by creating the HNSW index. First, let's drop the IVFFlat index created in the previous section because there is no need to have two indexes for vector data on the same table:

```
DROP INDEX omdb.movie_embeddings_ivfflat_idx;
```

Then, use the following query to create an HNSW index on the `movies` table.

Listing 8.12 Creating the HNSW index

```
CREATE INDEX movie_embeddings_hnsw_idx
ON omdb.movies
USING hnsw (movie_embedding vector_cosine_ops)
WITH (m = 8, ef_construction = 16);
```

This statement creates an index on the `movie_embedding` column using the cosine distance operator (`vector_cosine_ops`). Cosine distance is applied during index creation as the graph's nodes are connected by edges, and the same distance calculation method must be used later during vector similarity searches.

The statement defines two parameters in the `WITH` clause that are used during the index build time:

- `m` defines the maximum number of connections (edges) per node in the graph. A higher number of connections creates a denser graph, which improves recall and can speed up lookups by reducing the number of hops needed to find the best match. However, it also increases index build time and memory usage. The default value in `pgvector` is `16`, but we set it to `8` because our current dataset is relatively small.
- `ef_construction` controls the size of the candidate list during index construction. This list holds the closest candidates of a particular node during graph traversal. Once the traversal is completed, the list is truncated to the `m` nearest neighbors. A higher value typically results in the creation of a better index, as more close candidates are considered during construction. However, it comes at the cost of longer index build time. The default value in `pgvector` is `64`, and we set it to `16` to ensure that `ef_construction` is at least twice the value of `m`.

Once the index is created, let's execute the following query to check the new execution plan.

Listing 8.13 Checking the execution plan with the HNSW index

```
EXPLAIN (analyze, costs off)
SELECT id, name
FROM omdb.movies
ORDER BY movie_embedding <=>
    (SELECT phrase_embedding
    FROM omdb.phrases_dictionary
    WHERE phrase = 'May the force be with you')
LIMIT 3;
```

Postgres chooses the following execution plan, taking advantage of the HNSW index:

```
                            QUERY PLAN
-----------------------------------------------------------------
 Limit (actual time=1.889..1.923 rows=3 loops=1)
   InitPlan 1
     ->  Seq Scan on phrases_dictionary
➥          (actual time=0.025..0.026 rows=1 loops=1)
           Filter: (phrase = 'May the force be with you'::text)
           Rows Removed by Filter: 3
   ->  Index Scan using movie_embeddings_hnsw_idx on movies
➥        (actual time=1.887..1.918 rows=3 loops=1)
         Order By: (movie_embedding <=> (InitPlan 1).col1)
 Planning Time: 0.205 ms
 Execution Time: 1.970 ms
(9 rows)
```

This plan is comparable to the one we saw with the IVFFlat index. The database first retrieves the pregenerated embedding for the search phrase from the

`phrases_dictionary` table and then performs an `Index Scan` over the `movie_embeddings_hnsw_idx` index.

Now, let's have a look at the query result to assess the index recall characteristics:

```
SELECT id, name
FROM omdb.movies
ORDER BY movie_embedding <=>
    (SELECT phrase_embedding
    FROM omdb.phrases_dictionary
    WHERE phrase = 'May the force be with you')
LIMIT 3;
```

The query produces the following result, which is similar to the result from listing 8.5, where the database performed an ENN search (full table scan) by visiting every movie embedding in the table:

```
   id   |                     name
--------+--------------------------------------------
   1892 | Star Wars: Episode VI - Return of the Jedi
 204710 | The Acolyte
  10840 | On Your Mark
(3 rows)
```

This indicates that we managed to achieve perfect recall and significant performance improvements by creating the HNSW index with `m = 8` and `ef_construction = 16` parameters.

Improving index recall during search

Although the created HNSW index already demonstrates perfect recall for the "May the force be with you" search phrase, we should still test its accuracy more thoroughly with additional phrases that users of our movie recommendation service might use. This will help us further fine-tune the index build-time parameters (`m` and `ef_construction`), allowing us to strike a balance between index recall, performance, and size characteristics.

Imagine that we tested the index against a larger dataset with typical user phrases and confirmed that the selected parameters (`m = 8` and `ef_construction = 16`) already let us achieve recall as good as with an ENN search (full table scan). Thus, we decided to deploy the created index to production.

But what if, later in production, we notice that the index shows poor recall for some user phrases by returning inaccurate results? One option is to test the index further and re-create it with adjusted parameters. However, an additional option is to keep the existing index unchanged and instead configure the `hnsw.ef_search` parameter, which determines how many candidate nodes the algorithm considers potential nearest neighbors during graph traversal.

The `hnsw.ef_search` parameter is used during the search phase, and its default value is 40, meaning that up to 40 candidate nodes are stored as the algorithm searches for

the nearest neighbors of the vector embedding of a search phrase. Increasing this value can improve recall by allowing the algorithm to explore more potential nearest neighbors. However, this also increases search time, as more nodes in the graph need to be visited and evaluated.

The parameter can be adjusted dynamically. For instance, we can change it for all queries at the session level using the `SET hnsw.ef_search = {value}` command, or for an individual query in a transaction using the `SET LOCAL hnsw.ef_search = {value}` statement.

As an example, suppose the recall for the phrase "A movie about a Jedi who fights against the dark side of the force" is not good enough for our use case. As shown in the next listing, we can try to improve the accuracy by increasing the `hnsw.ef_search` value from 40 to 50.

Listing 8.14 Improving search accuracy with HNSW

```
BEGIN;
SET LOCAL hnsw.ef_search = 50;

SELECT id, name
FROM omdb.movies
ORDER BY movie_embedding <=>
  (SELECT phrase_embedding
  FROM omdb.phrases_dictionary
  WHERE phrase =
  'A movie about a Jedi who fights against the dark side of the force')
LIMIT 3;

COMMIT;
```

The query returns the following movie recommendations for the phrase:

```
   id   |                      name
--------+-----------------------------------------------
   1892 | Star Wars: Episode VI - Return of the Jedi
 204710 | The Acolyte
   1891 | Star Wars: Episode V - The Empire Strikes Back
(3 rows)
```

NOTE In our case, the search result is the same whether `hnsw.ef_search` is set to the default value of `40` or even to smaller values. This is because our current dataset is relatively small—just over 4,200 movies—and the selected index build-time settings (`m = 8` and `ef_construction = 16`) already allow us to achieve perfect recall, at least for the sample phrases.

With that, we've explored how to optimize vector similarity search in Postgres using IVFFlat and HNSW indexes. But which one should we use and when? The answer depends on a use case:

- Consider IVFFlat if a smaller index size or faster index build time is important. This index also works best when the data either remains unchanged since the index build time or is updated infrequently.
- Consider HNSW if search accuracy and query performance matter most and you're willing to trade off longer build times and larger index size. HNSW also offers stable recall characteristics, even as data is added or modified.

In the case of our movie recommendation service, we choose the HNSW index because the movie catalog is expected to grow over time, and we want this to have little or no effect on index recall and search performance.

8.6 Implementing RAG

The `pgvector` extension is a good example of why the extensions ecosystem plays a significant role in Postgres adoption for use cases the core database engine wasn't originally designed for. By enabling `pgvector`, we effectively turn Postgres into a vector database that can store vector embeddings, perform vector similarity search, and optimize those searches with specialized index types.

Using Postgres as a vector database allows us to build sophisticated gen AI applications that combine embedding models with Postgres to help users find relevant information or analyze private data using natural language. For example, our movie recommendation service can now suggest movies based on user prompts and questions.

But what if we want to go beyond simply retrieving data using vector similarity search and displaying it to the user? What if we want to use the retrieved data as context for an LLM—letting it augment its response or perform a specific task? This is where RAG comes into play.

With RAG, our application can treat the LLM as an application component that interacts with the user while taking our private data and knowledge base into account. Once the application receives a user prompt, it can use RAG in the following ways:

- *Context retrieval*—Use the user prompt to find relevant information in the private dataset.
- *LLM augmentation*—Provide the retrieved context to the LLM to augment its knowledge or behavior.
- *Response generation*—Let the LLM perform a task for the user if necessary and generate a final response.

Figure 8.10 shows in more detail how we can implement RAG for our movie recommendation service.

We can use this RAG-based solution to introduce an AI agent for our movie recommendation service that provides users with suggestions in a conversational style. As shown in figure 8.10, this implementation works as follows:

1. The user opens a chatbot and asks a question. The question is sent from the application frontend to the application backend.

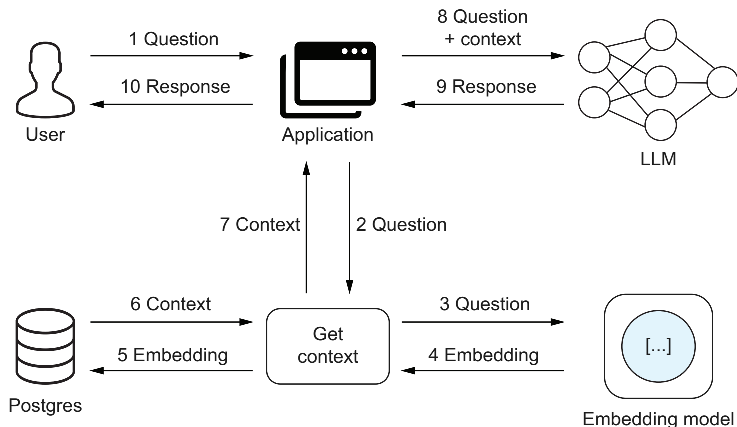


Figure 8.10 Implementing RAG with an LLM, an embedding model, and Postgres

2. The backend receives the question and uses a "get context" module to retrieve context from the private data stored in Postgres.
3. The context retrieval module uses an embedding model to generate an embedding for the user's question.
4. The embedding model returns the embedding.
5. The module uses the generated embedding to perform a vector similarity search in Postgres to find movies relevant to the user's prompt.
6. Postgres returns the result, which serves as the context we can pass to the LLM.
7. The context retrieval module returns the context to the backend logic that requested it.
8. The backend passes the user's question and the retrieved context to the LLM, asking it to generate movie recommendations. If needed, we can also include the history of the earlier conversation at this stage so the LLM can take it into account as well.
9. The LLM generates the response.
10. The response is sent back to the application frontend and displayed to the user in the chatbot window.

Now, let's learn how to create a simple working prototype of this RAG-based solution for the movie recommendation service.

8.6.1 *Preparing the environment for the prototype*

Our simple working prototype is implemented in Python and can be easily tested by running the steps in the following Jupyter notebook from the book's GitHub repository: https://github.com/dmagda/just-use-postgres-book/blob/main/ai_samples/rag.ipynb. The advantage of using a Jupyter notebook is that you don't need to know

Python to see how our RAG-based solution works in practice. All you need is Python installed on your machine and an IDE like Visual Studio Code to execute the notebook. From there, just follow the notebook's instructions to start an Ollama container, which deploys an LLM and an embedding model on your machine, and then step through the rest of the notebook.

The simple prototype continues using the `mxbai-embed-large` embedding model we introduced earlier to generate embeddings for our movies dataset and perform vector similarity search in Postgres. For the LLM, we use the TinyLlama model—a compact, open source LLM trained on 1.1 billion parameters. It consumes around 640 MB of memory and doesn't require much computational power, so it should work on most developer machines. If needed, you can always swap this model for a more advanced one, such as Llama 3.3 by Meta (which requires 43 GB of RAM) or GPT-4o by OpenAI (which requires an OpenAI subscription).

So if you'd like to follow along with the rest of the chapter, open the notebook and complete the prerequisites section—this includes deploying the Ollama container and installing Python. Alternatively, feel free to just read the remaining sections, as they still clearly explain how the RAG-based solution works in the selected environment.

8.6.2 Interacting with the LLM

The Jupyter notebook defines the `answer_question(question, context)` function that uses the `tinyllama` LLM to answer a user question using the provided context.

Listing 8.15 Answering user questions using the LLM and provided context

```
def answer_question(question, context):
    # Connecting to the tinyllama model with the LangChain Ollama interface
    llm = OllamaLLM(model="tinyllama", temperature=0.6)

    # Generating a final prompt for the LLM based on considering
    # the provided context
    prompt = f"""
    You're a movie expert and your task is to answer questions about movies
    based on the provided context.

    This is the user's question: {question}
    Consider the following context to provide a detailed and
    accurate answer: {context}

    The context includes the following details for each movie:
    - "Title" of the movie
    - "Vote Average" - the average rating of the movie
    - "Budget" - the budget allocated for the movie in US dollars
    - "Revenue" - the total revenue generated by the movie in US dollars
    - "Release Date" - the date the movie was released

    Respond in an engaging style that inspires the user to watch the movies.
    """
```

```
    # Invoke the LLM passing the prompt. The LLM will generate a response.
    response = llm.invoke(prompt)
    return response
```

Here's how the function works:

1. The logic uses the LangChain framework to connect to the TinyLlama model via LangChain's `OllamaLLM` interface.
2. The function prepares the `prompt` object that provides the instructions and data we want to pass to the LLM. It uses the `{question}` directive to include the user's original question and the `{context}` placeholder to pass in the context retrieved from Postgres. The `prompt` also specifies how the LLM should use this information and what kind of response to generate.
3. The function calls `llm.invoke(prompt)` to generate a response.

TIP Because LLMs are *stateless*—meaning they don't retain the history of the interaction—if we want the LLM to consider earlier conversation history, we need to store it separately and pass it to the `prompt` object using the `{history}` directive.

8.6.3 Retrieving context for LLM

The Jupyter notebook defines the `retrieve_context_from_postgres(question)` function to retrieve context from Postgres for a given user question. The implementation is shown next.

Listing 8.16 Retrieving context from Postgres

```
def retrieve_context_from_postgres(question):
    # Connect to the Postgres instance with the pgvector extension
    db_params = {
    "host": "localhost",
    "port": 5432,
    "dbname": "postgres",
    "user": "postgres",
    "password": "password"
    }
    conn = psycopg2.connect(**db_params)
    cursor = conn.cursor()

    # Connect to the embedding model using the OllamaEmbeddings interface
    embedding_model = OllamaEmbeddings(model="mxbai-embed-large:335m")

    # Generate the embedding for the user's question
    embedding = embedding_model.embed_query(question)

    # Perform vector similarity search to find relevant movies
    query = """
    SELECT name, vote_average, budget, revenue, release_date
```

```
    FROM omdb.movies
    ORDER BY movie_embedding <=> %s::vector LIMIT 3
    """

    cursor.execute(query, (embedding, ))

    context = ""

    # Generate context from the retrieved rows
    for row in cursor.fetchall():
        context += f"Movie title: {row[0]}, Vote Average: {row[1]}"
        context += f", Budget: {row[2]}, Revenue: {row[3]}"
        context += f", Release Date: {row[4]}\n"

    cursor.close()
    conn.close()

    return context
```

Here's how the function works:

- It connects to the locally running Postgres instance with the `pgvector` extension.
- It uses the `OllamaEmbeddings` interface from the LangChain framework to connect to the `mxbai-embed-large` embedding model running in the Ollama container. Then the embedding model generates an embedding for the user's question.
- The function uses the generated `embedding` to perform a vector similarity search in Postgres and find the top three most relevant movies.
- It builds the `context` object for the LLM from the result returned by the database.

With that, we have everything we need to experiment with a simple RAG implementation for our movie recommendation service.

8.6.4 Using RAG to answer questions

The following logic shows how RAG works by retrieving context from Postgres and using the LLM to generate a final response based on that context.

Listing 8.17 Using RAG to answer user questions

```
# Prepare a sample question.
question = "I'd like to watch the best movies about pirates." + \
    "Any suggestions?"

# Retrieve context from Postgres based on the user's question
context = retrieve_context_from_postgres(question)

print("Context from Postgres:")
print(context)

# Use the context to answer the user's question using the LLM
```

```
answer = answer_question(question, context)

print("LLM's answer:")
print(answer)
```

Here's how it works:

1. Create a sample user `question` or phrase you'd like to test. Feel free to change it and see how the response varies.
2. Use the `retrieve_context_from_postgres(question)` function to retrieve context from Postgres for the given sample question.
3. Use the `answer_question(question, context)` function to augment the LLM's knowledge with the provided context and have it generate the final response.

Once we execute this logic for the phrase "I'd like to watch the best movies about pirates. Any suggestions?" Postgres should return the following context after performing the vector similarity search:

```
Context from Postgres:
Movie title: Pirates of  the Caribbean: The Curse of the Black Pearl,
➥ Vote Average: 6.8974356651, Budget: 140000000, Revenue: 655011224,
➥ Release Date: 2003-07-09

Movie title: Pirates of the Caribbean: Dead Man's Chest,
➥ Vote Average: 6.6551699638, Budget: 110000000, Revenue: 1065659812,
➥ Release Date: 2006-07-07

Movie title: Pirates of the Caribbean: At World's End,
➥ Vote Average: 6.482758522, Budget: 110000000, Revenue: 401055000,
➥ Release Date: 2007-05-19
```

Then the TinyLlama LLM uses this context to generate a response that might look like this:

Certainly!

Consider watching "Pirates of the Caribbean: The Curse of the Black Pearl" and "Pirates of the Caribbean: Dead Man's Chest," both directed by Gore Verbinski.

The first movie is a classic adventure story with an impressive cast, including Johnny Depp as Captain Jack Sparrow, Orlando Bloom as Will Turner, and Geoffrey Rush as Captain Hector Barbossa. The film grossed over $1 billion worldwide and won numerous awards, including two Academy Awards for Best Costume Design and Best Makeup and Hairstyling.

The second movie, "Pirates of the Caribbean: Dead Man's Chest," is set a decade after the events of the first film and follows Jack Sparrow as he embarks on a dangerous mission to retrieve a lost treasure from the notorious pirate Blackbeard. The plot thickens when Jack's former crew members, including Will Turner, come back from the dead to challenge him for his leadership position.

Both movies are filled with action, adventure, and humor, making them enjoyable even for those who don't typically watch pirates-themed films. "Pirates of the Carib-

> bean: Dead Man's Chest" has a runtime of two hours and 20 minutes, but it's worth it for the epic battles, thrilling chases, and memorable characters.
>
> Don't miss out on this exciting and fun-filled experience!

The LLM followed our instructions and recommended the first two movies from the context retrieved from Postgres. It responded in an engaging way, adding extra details that weren't included in the context—such as the movie plots and the fact that Gore Verbinski directed both *The Curse of the Black Pearl* and *Dead Man's Chest*. These details were part of the LLM's training data.

> **WARNING** TinyLlama is a very compact LLM, requiring only around 640 MB of RAM, which makes it a great option for quick prototyping on general-purpose developer machines. However, the trade-off is that its responses are more prone to hallucinations, and it may not always follow instructions or produce accurate results. For example, if you take a closer look at the description of the second movie, *Dead Man's Chest*, you'll notice that it's completely inaccurate. It refers to the pirate Blackbeard, who appears only in *Pirates of the Caribbean: On Stranger Tides*. You can improve the output of the model by experimenting with the prompt message or adjusting the temperature parameter. Alternatively, you can deploy a more advanced LLM in your Ollama container—one trained with more parameters—but doing so will require more system resources.

With that, we've learned how to use the `pgvector` extension to turn Postgres into a vector database and start building gen AI applications. What should you do next? First, choose the programming language you want to use for your gen AI app, and pick one of the gen AI frameworks available in that language's ecosystem. Then take a deeper look at how that framework integrates with Postgres to build your app as efficiently as possible.

For example, in this section, we used Python with LangChain, which natively supports Postgres and `pgvector`. LangChain is also a default choice for many JavaScript developers. If you're a Java developer, consider using Spring AI.

> **TIP** Explore the `pgai` extension if you'd like to implement the RAG workflow purely in SQL and execute it entirely within the database—in case you prefer not to use an application-level gen AI framework.

In the meantime, if you'd like to see more use cases for using Postgres with gen AI applications, check out the videos in the curated list available in the book's GitHub repository: https://github.com/dmagda/just-use-postgres-book/blob/main/ai_samples/more_samples.md. For example, one video walks through creating an AI agent that uses a text-to-SQL interface to retrieve or modify data in Postgres. Another video demonstrates how to build a custom plugin for the ChatGPT marketplace that provides users with movie recommendations and manages their watch lists. More videos with code samples will be added along the way!

Summary

- Postgres can be easily used alongside LLMs and embedding models to build new types of gen AI applications, including recommendation services and autonomous AI agents.
- The `pgvector` extension turns Postgres into a vector database capable of storing vector embeddings, performing vector similarity searches, and optimizing those searches with specialized index types.
- `pgvector` supports IVFFlat and HNSW indexes, helping you find the right tradeoff between search performance, recall, and index size.
- Postgres can serve as a powerful vector database for implementing RAG and other gen AI use cases.

9 Postgres for time series

This chapter covers

- Exploring Postgres's capabilities for time-series workloads
- Using the TimescaleDB extension to manage and process time-series data
- Analyzing time-series data with specialized functions and aggregates
- Optimizing queries with B-tree and BRIN indexes

A *time series* is a sequence of data points collected over time, with each point representing the state of a system or object at a specific moment. By capturing time-series data over a given period, we can observe how the system has evolved or changed, which helps us identify trends, take proactive actions, or make future predictions. For instance, we work with time-series data when reviewing CPU and memory usage from a server over the past week, exploring currency exchange rate fluctuations over the last three months, or analyzing a patient's vital signs collected over the past year.

Let's learn how to use Postgres for time-series data and workloads as we build a smartwatch application that tracks user heart rate and records measurements for

further analysis. We'll learn how to efficiently store and analyze heart rate data in Postgres, keeping users informed of important health trends and changes.

9.1 How Postgres works with time-series data

If you've never worked on applications using time-series data before, all you need to know is that time-series data has two distinguishing characteristics. First, every data point includes the time associated with the measurement or event. Second, time-series data is append-only by nature, meaning that once a measurement or event is recorded and stored, it's never updated. Let's see how to take these two characteristics into account when using Postgres as a time-series database.

> **NOTE** Although it's optional, if you'd like to run a few commands from this section of the chapter, connect to one of your Postgres containers started in previous chapters. For instance, you can use the `docker exec -it postgres psql -U postgres` command to connect to the container from chapter 1—just make sure it's running first.

Imagine that we've created an application for a smartwatch that constantly monitors a user's heart rate. The application sends the collected data to our backend service at regular one-minute intervals or in bulk if the device was disconnected from the network. Once received by the backend, every record is stored in the following Postgres table for further analysis:

```
CREATE TABLE heart_rate_measurements (
  watch_id INT NOT NULL,
  recorded_at TIMESTAMPTZ NOT NULL,
  heart_rate INT NOT NULL,
  activity TEXT NOT NULL CHECK (
      activity IN ('walking', 'sleeping', 'resting', 'workout'))
);
```

As the table structure shows, every measurement received from the device includes the following information:

- `watch_id`—A unique ID of the device.
- `recorded_at`—The time the measurement took place. We use the `TIMESTAMPTZ` (timestamp with time zone) data type because it stores a point in time as the number of microseconds since January 1, 2000, in UTC. The database automatically converts the timestamp to UTC before storing it in the table. When the value is read back by the application, Postgres transforms it from UTC to the time zone associated with the database server, current database connection, or user.
- `heart_rate`—The actual heart rate in beats per minute (BPM) recorded at that moment.
- `activity`—Identifies the user's activity at the time the measurement was taken.

With this simple table structure, we already adhere to best practices for time-series applications by storing only the minimum information required for a single measurement. This simplicity is intentional, as time-series data is append-only by nature and can cause the table size to grow rapidly. A more complex table structure would only accelerate that growth, increasing both storage and processing costs.

For instance, if our smartwatch application sends heart rate measurements at one-minute intervals, a single user will generate 1,440 records per day (1 day = 1,440 minutes). If the application is used by 1,000 users, the total number of daily records will be 1,440 measurements × 1,000 users = 1,440,000 records!

> **NOTE** There are time-series systems where data is collected at a much higher rate, with intervals measured in seconds or even milliseconds, resulting in far larger volumes. At those rates, the number of daily records can easily reach billions.

Having a single large table storing all heart rate measurements isn't practical, because, most of the time, users are interested in seeing only the latest data, typically for the current or past few days. Weekly or monthly health trends are checked only occasionally.

For this reason, current and historical time-series data should be stored separately. This separation ensures that queries working with recent data remain fast with little or no effect from the growing data volume, while historical data can still be easily accessed, compressed, or evicted if needed.

Postgres offers a *table partitioning* feature that allows us to split one large logical table into smaller physical tables called *partitions*. One form of partitioning is range partitioning, where the table is divided into non-overlapping ranges defined by the values of a column or a set of columns.

For example, the following statement shows how we can enable range partitioning for our table that stores heart rate measurements:

```
CREATE TABLE heart_rate_measurements (
  watch_id INT NOT NULL,
  recorded_at TIMESTAMPTZ NOT NULL,
  heart_rate INT NOT NULL,
  activity TEXT NOT NULL CHECK (
      activity IN ('walking', 'sleeping', 'resting', 'workout'))
) PARTITION BY RANGE (recorded_at);
```

The `PARTITION BY RANGE` clause defines that the table is split into partitions based on the values of the `recorded_at` column.

The next step is to create the partitions by defining non-overlapping ranges for the `recorded_at` values. For example, here's how we can define two partitions: one to store measurements for January 2025, and the other for February 2025:

```
CREATE TABLE measurements_jan2025
    PARTITION OF heart_rate_measurements
    FOR VALUES FROM ('2025-01-01') TO ('2025-02-01');
```

```
CREATE TABLE measurements_feb2025
    PARTITION OF heart_rate_measurements
    FOR VALUES FROM ('2025-02-01') TO ('2025-03-01');
```

These partitions are regular Postgres tables created using a special form of the `CREATE TABLE` command:

- The `PARTITION OF` clause defines the parent table that the partition belongs to, which is `heart_rate_measurements`.
- The `FOR VALUES` clause specifies the value range that the partition stores.

The bounds of a partition are inclusive on the lower end and exclusive on the upper end. For instance, the clause `FOR VALUES FROM ('2025-01-01') TO ('2025-02-01')` means the partition will hold all values starting with (and including) `2025-01-01 00:00:00+00` and up to (but not including) `2025-02-01 00:00:00+00`.

Once the partitions are in place, our application can simply query the main `heart_rate_measurements` table, letting Postgres decide which partition(s) need to be accessed during query execution. For example, suppose we want to return all measurements from January 1 through January 3 (inclusive), and we want to inspect the execution plan for this query:

```
EXPLAIN
SELECT * FROM heart_rate_measurements
WHERE recorded_at BETWEEN '2025-01-01' AND '2025-01-03';
```

The query produces the following execution plan, showing that Postgres queried only the partition storing measurements for January 2025 (`Seq Scan on measurements_jan2025`):

```
                                  QUERY PLAN
---------------------------------------------------------------------------
 Seq Scan on measurements_jan2025 heart_rate_measurements
➥   (cost=0.00..26.05 rows=5 width=48)
   Filter: ((recorded_at >=
➥   '2025-01-01 00:00:00+00'::timestamp with time zone) AND
➥   (recorded_at <= '2025-01-03 00:00:00+00'::timestamp with time zone))
(2 rows)
```

If our application wants to fetch measurements between January 30 and February 2, we can still query the main table directly without worrying about how the data is distributed across partitions:

```
EXPLAIN
SELECT * FROM heart_rate_measurements
WHERE recorded_at BETWEEN '2025-01-30' AND '2025-02-02';
```

In this case, Postgres determines that both partitions—`measurements_jan2025` and `measurements_feb2025`—need to be accessed, and it uses the following execution plan:

```
                                 QUERY PLAN
----------------------------------------------------------------------------
 Append  (cost=0.00..52.15 rows=10 width=48)
   ->  Seq Scan on measurements_jan2025 heart_rate_measurements_1
➥        (cost=0.00..26.05 rows=5 width=48)
         Filter: ((recorded_at >=
➥       '2025-01-30 00:00:00+00'::timestamp with time zone) AND
➥      (recorded_at <= '2025-02-02 00:00:00+00'::timestamp with time zone))
   ->  Seq Scan on measurements_feb2025 heart_rate_measurements_2
➥        (cost=0.00..26.05 rows=5 width=48)
         Filter: ((recorded_at >=
➥      '2025-01-30 00:00:00+00'::timestamp with time zone) AND
➥      (recorded_at <= '2025-02-02 00:00:00+00'::timestamp with time zone))
(5 rows)
```

Table partitioning is one of the key Postgres capabilities for time-series workloads. It addresses the challenge of storing all time-series data in a single large table by splitting it into partitions, with each partition holding a subset of the dataset. Partitioning can also significantly improve query performance by having Postgres access only the partitions that match the search criteria.

> **TIP** In this section, we've explored the basics of table partitioning in Postgres. If you decide to dive deeper into the topic, refer to the official Postgres documentation: https://www.postgresql.org/docs/current/ddl-partitioning.html.

Even though table partitioning is transparent to our application layer—which can continue querying the main `heart_rate_measurements` table directly—someone still needs to manage the partitions. This includes creating partitions for future dates and deciding when to evict, truncate, or compress partitions with historical data.

Postgres offers extensions to simplify and automate partition management. For example, we can use a combination of `pg_partman` and `pg_cron` for this purpose. The `pg_partman` extension automates partition creation and maintenance through its built-in background worker, whereas `pg_cron` provides a cron-based scheduler in the database that, if necessary, can run `pg_partman`'s maintenance tasks at specific times or on a recurring schedule.

However, in this book, instead of building a custom solution from scratch using Postgres table partitioning along with the `pg_partman` and `pg_cron` extensions, we'll explore a simpler option. We'll use another extension called TimescaleDB, which is purpose-built for time-series use cases. As we'll see, TimescaleDB also automates partition creation and management.

9.2 Starting Postgres with TimescaleDB

The TimescaleDB extension provides Postgres with capabilities you'd expect from a typical time-series database. It automatically partitions tables with time-series data, supports a columnar storage engine optimized for analytical queries, allows you to configure data retention policies, provides specialized functions for querying

time-series data, and much more. We'll explore TimescaleDB's capabilities throughout this chapter; but first, let's start a Postgres instance with the extension in a Docker container.

Stopping the Postgres container used in previous chapters

In previous chapters, we used Postgres containers that don't include the TimescaleDB extension. If you have any of those Postgres containers running, stop them to free port 5432. First, find any running Postgres container that is listening on port 5432:

```
docker ps --filter "name=postgres" --filter "publish=5432"
```

The command might output a result similar to the following, indicating that a container is currently running:

```
IMAGE             STATUS         PORTS                       NAMES
postgres:latest   Up 50 minutes  0.0.0.0:5432->5432/tcp   <container-name>
```

Stop the container by providing its name in the `<container-name>` placeholder:

```
docker container stop <container-name>
```

If you're on a Unix operating system such as Linux or macOS, use the following command to start a Postgres container with TimescaleDB.

Listing 9.1 Starting Postgres with TimescaleDB on Unix

```
docker volume create postgres-timescale-volume

docker run --name postgres-timescale \
    -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password \
    -p 5432:5432 \
    -v postgres-timescale-volume:/var/lib/postgresql/data \
    -d timescale/timescaledb:2.19.2-pg17
```

If you're a Windows user, use the next command in PowerShell instead. If you use Command Prompt (CMD), replace each backtick (`) with a caret (^) at the end of each line:

Listing 9.2 Starting Postgres with TimescaleDB on Windows

```
docker volume create postgres-timescale-volume

docker run --name postgres-timescale `
    -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password `
    -p 5432:5432 `
    -v postgres-timescale-volume:/var/lib/postgresql/data `
    -d timescale/timescaledb:2.19.2-pg17
```

Both commands start the `postgres-timescale` container using the `timescale/timescaledb:2.19.2-pg17` image that includes the TimescaleDB extension. The container listens for incoming connections on port 5432 and stores Postgres data in the Docker-managed volume named `postgres-timescale-volume`.

> **NOTE** The way we deploy Postgres in Docker works well for development and for exploring the database's capabilities. However, if you plan to use this deployment option in production, be sure to review security and other deployment best practices.

Once the container is started, connect to it using the psql client:

```
docker exec -it postgres-timescale psql -U postgres
```

Confirm that the TimescaleDB extension exists in the Postgres container by executing the following command.

Listing 9.3 Checking the available TimescaleDB version

```
SELECT * FROM pg_available_extensions
WHERE name = 'timescaledb';
```

The output should look similar to the following:

```
    name     | default_version | installed_version |          comment
-------------+-----------------+-------------------+-----------------------
 timescaledb | 2.19.2          | 2.19.2            | Enables scalable
             |                 |                   | inserts and complex
             |                 |                   | queries for
             |                 |                   | time-series data
```

Because the `installed_version` column is set to a specific version, it means TimescaleDB is already enabled for the `postgres` database we're connected to. You can always check the current database using the `\conninfo` command of the psql tool, which returns output like this:

```
\conninfo

You are connected to database "postgres" as user "postgres" via socket in
➥ "/var/run/postgresql" at port "5432".
```

With that, we're ready to move on to the next step by preloading a sample dataset for our application that tracks user heartbeats on smartwatch devices.

9.3 Loading the time-series data

Imagine that our smartwatch application has already been used by thousands of users for more than a year and has generated a significant number of heartbeat

measurements, currently stored in another database. Now we'd like to migrate to Postgres, but we want to run a proof of concept (POC) first.

We've downloaded a subset of the existing production data and prepared it for our Postgres instance with the TimescaleDB extension. Let's follow these steps to preload the dataset into Postgres:

1. Clone the book's repository with listings and sample data:

   ```
   git clone https://github.com/dmagda/just-use-postgres-book
   ```

2. Copy the smartwatch dataset to your Postgres container:

   ```
   cd just-use-postgres-book/
   docker cp data/smartwatch/. postgres-timescale:/home/.
   ```

3. Preload the dataset by connecting to the container and using the `\i` meta-command of psql to apply the copied SQL scripts:

   ```
   docker exec -it postgres-timescale psql -U postgres -c "\i /home/schema.sql"
   docker exec -it postgres-timescale psql -U postgres -c  "\i /home/data.sql"
   ```

The table with the heartbeat measurements is created under the `watch` schema, and you can confirm this by connecting to Postgres and finding the created table with the `\dt watch.*` command:

```
docker exec -it postgres-timescale psql -U postgres
\dt watch.*

                  List of relations
 Schema |          Name           | Type  |  Owner
--------+-------------------------+-------+----------
 watch  | heart_rate_measurements | table | postgres
(1 row)
```

The created table contains over a million measurements for three users, collected from the end of 2024 through all of 2025.

```
SELECT count(*) FROM watch.heart_rate_measurements;

  count
---------
 1399415
(1 row)
```

The structure of the table is similar to what we discussed in the previous section:

```
watch.heart_rate_measurements (
  watch_id INT NOT NULL,
```

```
  recorded_at TIMESTAMPTZ NOT NULL,
  heart_rate INT NOT NULL,
  activity TEXT NOT NULL
    CHECK (activity IN ('walking', 'sleeping', 'resting', 'workout'))
);
```

Now, let's execute the following query to see how many measurements we have for each user device.

Listing 9.4 Number of measurements per user device

```
SELECT watch_id, count(*) as total_measurements
FROM watch.heart_rate_measurements
GROUP BY watch_id ORDER BY watch_id;
```

According to the output, we've gathered a comparable number of samples on each device:

```
 watch_id | total_measurements
----------+--------------------
        1 |             466497
        2 |             466409
        3 |             466509
(3 rows)
```

Finally, let's run the following query to see what a time-series sequence looks like for the period between 6:50 a.m. and 6:55 a.m. on January 1, 2025.

Listing 9.5 Time-series sequence for a specific period

```
SELECT watch_id, recorded_at, heart_rate, activity
FROM watch.heart_rate_measurements
WHERE watch_id = 1 AND
  recorded_at BETWEEN '2025-01-01 06:50:00' AND '2025-01-01 06:55:00'
ORDER BY recorded_at;
```

The query returns the following time-series sequence for the user wearing the watch with `watch_id = 1`:

```
 watch_id |      recorded_at       | heart_rate | activity
----------+------------------------+------------+----------
        1 | 2025-01-01 06:50:00+00 |         66 | resting
        1 | 2025-01-01 06:51:00+00 |         69 | resting
        1 | 2025-01-01 06:52:00+00 |         65 | resting
        1 | 2025-01-01 06:53:00+00 |         68 | resting
        1 | 2025-01-01 06:54:00+00 |         77 | walking
        1 | 2025-01-01 06:55:00+00 |         65 | resting
(6 rows)
```

The watch performed six measurements at one-minute intervals, showing how the user's heart rate changed over time. Next, let's learn how the TimescaleDB extension takes advantage of Postgres partitioning capabilities to store these millions of measurements efficiently in the database.

9.4 Exploring TimescaleDB hypertables

One of the scripts we used for data preloading in the previous section included the following statement:

```
SELECT create_hypertable(
  relation => 'watch.heart_rate_measurements',
  dimension => by_range('recorded_at', interval '1 month'),
  create_default_indexes => false
);
```

This statement converts our `watch.heart_rate_measurements` table into a TimescaleDB *hypertable*, which is a regular Postgres table that automatically creates and manages partitions with time-series data. The parameters of the `create_hypertable` function serve the following purposes:

1. The `relation` parameter defines the name of the table we want to turn into a hypertable, which is `watch.heart_rate_measurements` in our case.
2. The `dimension` parameter specifies the partitioning method. Here, we partition the data by range, with each range representing a one-month interval based on the values of the `recorded_at` column.
3. The `create_default_indexes` parameter controls whether default indexes should be created for the partitioned table. We disable this option because indexing for time-series data will be covered in more detail in the following sections.

WARNING We split the dataset into monthly partitions for the sake of simplicity and because our dataset includes measurements from only three user devices. However, once we start receiving data from thousands or even millions of devices, monthly-based partitions won't be optimal. Each partition would end up storing a large volume of data, which could jeopardize database and application performance. In such cases, we'll need to partition the data at a more granular level—by days, hours, or even device.

We can confirm that the hypertable automatically manages partitions by executing the following command, which returns details for the `watch.heart_rate_measurements` table:

```
\d+ watch.heart_rate_measurements
```

The `Child tables` section of the output lists all the partitions (child tables) that belong to the main/parent table. Here's what the truncated output looks like:

```
Child tables: _timescaledb_internal._hyper_1_10_chunk,
              _timescaledb_internal._hyper_1_11_chunk,
              _timescaledb_internal._hyper_1_12_chunk,
              _timescaledb_internal._hyper_1_1_chunk,
              _timescaledb_internal._hyper_1_2_chunk,
```

The TimescaleDB extension refers to these partitions as *chunks*, which are, in fact, regular Postgres tables. Thus, we'll use the terms *partition*, *chunk*, and *child table* interchangeably. For example, we can use the `\d` command on one of the chunks to inspect its table structure:

```
\d _timescaledb_internal._hyper_1_1_chunk
```

The output should look like this, with the `constraint_1` constraint defining the time range of the data stored in this chunk:

```
             Table "_timescaledb_internal._hyper_1_1_chunk"
   Column    |           Type           | Collation | Nullable | Default
-------------+--------------------------+-----------+----------+---------
 watch_id    | integer                  |           | not null |
 recorded_at | timestamp with time zone |           | not null |
 heart_rate  | integer                  |           | not null |
 activity    | text                     |           | not null |
Check constraints:
    "constraint_1" CHECK (recorded_at >=
➥   '2024-12-13 00:00:00+00'::timestamp with time zone AND
➥   recorded_at < '2025-01-12 00:00:00+00'::timestamp with time zone)
    "heart_rate_measurements_activity_check" CHECK (activity =
➥   ANY(ARRAY['walking'::text, 'sleeping'::text, 'resting'::text,
➥   'workout'::text]))
Inherits: watch.heart_rate_measurements
```

Postgres will query this chunk if our application selects or inserts data for the time interval between `2024-12-13 00:00:00+00` and `2025-01-12 00:00:00+00`. We can easily confirm this by looking at the execution plan for a simple query:

```
EXPLAIN (analyze, costs off)
SELECT count(*) FROM watch.heart_rate_measurements
WHERE recorded_at = '2025-01-01';
```

According to the output, the database queried just a single partition called `_hyper_1_1_chunk` and skipped all the other chunks that store data outside the specified time range:

```
                                    QUERY PLAN
------------------------------------------------------------------------
 Aggregate (actual time=4.431..4.432 rows=1 loops=1)
   ->  Seq Scan on _hyper_1_1_chunk
➥       (actual time=0.037..4.423 rows=3 loops=1)
         Filter: (recorded_at =
➥       '2025-01-01 00:00:00+00'::timestamp with time zone)
         Rows Removed by Filter: 46108
```

```
 Planning Time: 0.617 ms
 Execution Time: 4.495 ms
(6 rows)
```

The TimescaleDB extension also provides additional functions that let us learn more about existing chunks or perform actions on them. For example, we can use the `show_chunks` function to list all the partitions that have been created for our `heart_rate_measurements` table:

```
SELECT show_chunks('watch.heart_rate_measurements');
```

According to the output, there are currently 12 chunks, each storing time-series data for one of the monthly intervals:

```
               show_chunks
-----------------------------------------
 _timescaledb_internal._hyper_1_1_chunk
 _timescaledb_internal._hyper_1_2_chunk
 _timescaledb_internal._hyper_1_3_chunk
 _timescaledb_internal._hyper_1_4_chunk
 _timescaledb_internal._hyper_1_5_chunk
 _timescaledb_internal._hyper_1_6_chunk
 _timescaledb_internal._hyper_1_7_chunk
 _timescaledb_internal._hyper_1_8_chunk
 _timescaledb_internal._hyper_1_9_chunk
 _timescaledb_internal._hyper_1_10_chunk
 _timescaledb_internal._hyper_1_11_chunk
 _timescaledb_internal._hyper_1_12_chunk
(12 rows)
```

For example, the last chunk, `_timescaledb_internal._hyper_1_12_chunk`, stores data for the period between `2025-11-08 00:00:00+00` and `2025-12-08 00:00:00+00`. We can confirm this using the `\d` command:

```
\d _timescaledb_internal._hyper_1_12_chunk
```

The `Check constraints` section of the output defines the time range the last chunk is responsible for:

```
Check constraints:
    "constraint_12" CHECK (recorded_at >=
➥  '2025-11-08 00:00:00+00'::timestamp with time zone AND
➥  recorded_at < '2025-12-08 00:00:00+00'::timestamp with time zone)
```

Because this is the latest chunk, it means there are no measurements for or after 2025-12-08. Let's confirm that with the following query:

```
SELECT * FROM watch.heart_rate_measurements
WHERE recorded_at >= '2025-12-08 00:00:00';
```

And the query returns an empty result:

```
 watch_id | recorded_at | heart_rate | activity
----------+-------------+------------+----------
(0 rows)
```

Now, let's see what happens when our application backend receives the first measurement from a user device on 2025-12-08 or any time after that:

```
INSERT INTO watch.heart_rate_measurements VALUES
(1,'2025-12-08 00:25:00',57,'sleeping');
```

The TimescaleDB extension automatically creates a new partition named `_timescaledb_internal._hyper_1_13_chunk`, as shown in the truncated output of the `show_chunks` command:

```
SELECT show_chunks('watch.heart_rate_measurements');

               show_chunks
-----------------------------------------
...earlier chunks
 _timescaledb_internal._hyper_1_13_chunk
(13 rows)
```

The `Check constraints` section of the `\d` command for the new chunk shows the time range it covers:

```
\d _timescaledb_internal._hyper_1_13_chunk

Check constraints:
    "constraint_13" CHECK (recorded_at >=
 ➥  '2025-12-08 00:00:00+00'::timestamp with time zone AND
 ➥  recorded_at < '2026-01-07 00:00:00+00'::timestamp with time zone)
```

Finally, let's confirm that Postgres accesses only this new partition when our application queries for the just-inserted record:

```
EXPLAIN (analyze, costs off)
SELECT * FROM watch.heart_rate_measurements
WHERE recorded_at >= '2025-12-08 00:00:00';
```

And according to the plan, the database scanned only the newly created partition `_hyper_1_13_chunk` and skipped all the others:

```
                                  QUERY PLAN
----------------------------------------------------------------------------
 Seq Scan on _hyper_1_13_chunk (actual time=0.013..0.014 rows=1 loops=1)
   Filter: (recorded_at >=
 ➥  '2025-12-08 00:00:00+00'::timestamp with time zone)
```

```
 Planning Time: 0.208 ms
 Execution Time: 0.026 ms
(4 rows)
```

With that, our smartwatch application can rely on Postgres and its TimescaleDB extension to handle ever-growing volumes of time-series measurements. The database automatically arranges incoming data across a set of partitions and then queries only the subset that satisfies the search criteria, allowing for predictable high performance regardless of data volume.

Automatically drop data with the data retention policy

If you'd like to keep time-series data only for a specific time period, you can define a data retention policy on a hypertable. The following example shows how to create a 30-day retention policy for our table with heart rate measurements. Chunks that store data older than 30 days from the time of policy creation will be automatically discarded by the TimescaleDB extension.

Do not add this retention policy while reading the chapter; if you do, it might remove all the records from the `watch.heart_rate_measurements` table. The sample dataset is dated 2024 and 2025, so if you're reading the book after 2025, all of the data may fall outside the 30-day retention window. With that warning, here's the command:

```
SELECT add_retention_policy(
  'watch.heart_rate_measurements', INTERVAL '30 days');
```

Once the policy is defined, the extension schedules a job that runs daily and drops chunks containing only data older than 30 days. You can configure the job to run more or less frequently if necessary. For instance, this example shows how to create a retention policy with the job executed every 12 hours:

```
SELECT add_retention_policy(
  'watch.heart_rate_measurements',
  drop_after => INTERVAL '30 days',
  schedule_interval => INTERVAL '12 hours');
```

9.5 Analyzing time-series data

After loading a subset of our production data into Postgres, we're ready to explore just how flexible and powerful the database is when it comes to querying time-series data. The TimescaleDB extension provides dozens of specialized functions designed to efficiently process and analyze large volumes of time-series data while maintaining high performance. Let's explore some of these capabilities so users of our smartwatch application can better understand and monitor their health as well as receive important notifications in a timely manner.

9.5.1 Using the time_bucket function

The `time_bucket` function in TimescaleDB simplifies the aggregation of time-series data. It groups timestamps into specified intervals, or *buckets*, which are useful for roll-ups and aggregations—such as the average heart rate over the last 30 minutes, the max heart rate during 1-hour workout sessions, or the lowest heart rate at 15-minute intervals during sleep.

Depending on the method signature, the function can accept up to five arguments, with the first two always required:

```
time_bucket(bucket_width, ts, [origin, timezone, offset])
```

The meaning of the arguments is as follows:

- `bucket_width`—A time interval that defines the length of each bucket: for example, `'5 minutes'`, `'3 days'`, or `'1 week'`.
- `ts`— The timestamp value to bucket.
- `origin` (optional)—A fixed point in time from which the buckets are aligned. By default, buckets are aligned to midnight on January 1, 2000, when working with monthly, yearly, or century intervals, or to midnight on January 3 in all other cases.
- `timezone` (optional)—The timezone used for calculating bucket start and end times. Defaults to UTC.
- `offset` (optional)—A time interval by which to offset all time buckets.

Let's explore how to use the function in practice by supporting several capabilities for our smartwatch application.

One of the capabilities allows our users to keep an eye on the average and max heart rates during or after workout sessions. For example, imagine that the user wearing the watch with `watch_id = 1` had a workout session on 2025-04-23. After the session ended, the user opened our app to view charts showing their average and maximum heart rates during the workout at 10-minute intervals. Such charts can be easily populated with the data returned by the following query.

Listing 9.6 Calculating average and max heart rates during workouts

```
SELECT
  time_bucket('10 minutes', recorded_at) AS period, activity,
  AVG(heart_rate)::int AS avg_rate, MAX (heart_rate)::int AS max_rate
FROM watch.heart_rate_measurements
WHERE watch_id = 1 AND activity = 'workout'
  AND recorded_at >= '2025-04-23' AND recorded_at < '2025-04-24'
GROUP BY period, activity ORDER BY period;
```

The query works as follows:

- It runs only for the records that satisfy the condition in the `WHERE` clause.

- The `time_bucket('10 minutes', recorded_at)` function places matching records into 10-minute buckets based on their `recorded_at` value.
- The average and maximum heart rate are calculated and reported for each bucket.

Once we execute the query, it returns the following result for the user:

```
         period         | activity | avg_rate | max_rate
------------------------+----------+----------+----------
 2025-04-23 07:00:00+00 | workout  |      126 |      137
 2025-04-23 07:10:00+00 | workout  |      129 |      138
 2025-04-23 07:20:00+00 | workout  |      131 |      139
 2025-04-23 07:30:00+00 | workout  |      128 |      135
 2025-04-23 07:40:00+00 | workout  |      135 |      183
 2025-04-23 07:50:00+00 | workout  |      130 |      139
 2025-04-23 18:00:00+00 | workout  |      130 |      139
 2025-04-23 18:10:00+00 | workout  |      136 |      184
 2025-04-23 18:20:00+00 | workout  |      134 |      186
 2025-04-23 18:30:00+00 | workout  |      130 |      139
 2025-04-23 18:40:00+00 | workout  |      135 |      193
 2025-04-23 18:50:00+00 | workout  |      129 |      135
(12 rows)
```

Another useful capability allows our users to see weekly summaries for each type of activity. The next following query calculates heart rate aggregates for the same user with `watch_id = 1` over the period from April 1 through April 15, 2025.

Listing 9.7 Calculating a weekly summary for every type of activity

```
SELECT time_bucket('1 week', recorded_at) AS period, activity,
  AVG(heart_rate)::int AS avg_rate,
  MAX (heart_rate)::int AS max_rate, MIN (heart_rate)::int AS min_rate
FROM watch.heart_rate_measurements
WHERE watch_id = 1 AND recorded_at >= '2025-04-01'
  AND recorded_at < '2025-04-15'
GROUP BY period, activity ORDER BY period, activity;
```

This query puts the data satisfying the search criteria into weekly buckets as defined by the `time_bucket('1 week', recorded_at)` function and returns the following result:

```
         period         | activity | avg_rate | max_rate | min_rate
------------------------+----------+----------+----------+----------
 2025-03-31 00:00:00+00 | resting  |       67 |       69 |       65
 2025-03-31 00:00:00+00 | sleeping |       57 |       59 |       55
 2025-03-31 00:00:00+00 | walking  |       79 |       84 |       75
 2025-03-31 00:00:00+00 | workout  |      131 |      196 |      120
 2025-04-07 00:00:00+00 | resting  |       67 |       69 |       65
 2025-04-07 00:00:00+00 | sleeping |       57 |       59 |       55
 2025-04-07 00:00:00+00 | walking  |       79 |       84 |       75
 2025-04-07 00:00:00+00 | workout  |      130 |      199 |      120
```

```
 2025-04-14 00:00:00+00 | resting  |       67 |       69 |       65
 2025-04-14 00:00:00+00 | sleeping |       57 |       59 |       55
 2025-04-14 00:00:00+00 | walking  |       79 |       84 |       75
 2025-04-14 00:00:00+00 | workout  |      131 |      182 |      120
(12 rows)
```

Even though the WHERE clause of the query returns records with `recorded_at >= '2025-04-01'`, the summary for the first week starts on 2025-03-31 instead of April 1. This happens because by default `time_bucket()` aligns weekly buckets to midnight on January 3, 2000 (Monday). It creates repeating seven-day intervals from that exact point in time.

If we check the calendar, we'll see that March 31, 2025, is a Monday, and the interval starting on that day will include heart rate measurements captured from April 1 through April 6. However, if our users want to see the same reports but aligned to the beginning of the month, we can set April 1, 2025, as the origin for the weekly buckets. The following query does this by passing `'2025-04-01'::timestamptz` via the `origin` argument of the `time_bucket` function.

Listing 9.8 Changing the time origin for the weekly summary

```
SELECT time_bucket('1 week', recorded_at, '2025-04-01'::timestamptz)
  AS period, activity,
  AVG(heart_rate)::int AS avg_rate,
  MAX (heart_rate)::int AS max_rate, MIN (heart_rate)::int AS min_rate
FROM watch.heart_rate_measurements
WHERE watch_id = 1 AND recorded_at >= '2025-04-01' AND
  recorded_at < '2025-04-15'
GROUP BY period, activity ORDER BY period, activity;
```

The query produces the following result using 2025-04-01 as the beginning (origin) for the weekly intervals:

```
         period         | activity | avg_rate | max_rate | min_rate
------------------------+----------+----------+----------+----------
 2025-04-01 00:00:00+00 | resting  |       67 |       69 |       65
 2025-04-01 00:00:00+00 | sleeping |       57 |       59 |       55
 2025-04-01 00:00:00+00 | walking  |       79 |       84 |       75
 2025-04-01 00:00:00+00 | workout  |      131 |      199 |      120
 2025-04-08 00:00:00+00 | resting  |       67 |       69 |       65
 2025-04-08 00:00:00+00 | sleeping |       57 |       59 |       55
 2025-04-08 00:00:00+00 | walking  |       79 |       84 |       75
 2025-04-08 00:00:00+00 | workout  |      130 |      198 |      120
(8 rows)
```

Finally, by default, the `time_bucket` function uses the UTC time zone to calculate bucket start and end times. This is also the default time zone used by our database server and user connections. We can verify that by executing the `SHOW time zone` command:

```
SHOW time zone;

 TimeZone
----------
 UTC
(1 row)
```

Let's suppose that our second user lives in Japan and wears a watch with `watch_id = 2`. As listing 9.9 shows, we can explicitly set the `time zone` parameter of the `time_bucket` function to Asia/Tokyo for that user, making sure the start and end times of the buckets align with the user's local time.

Listing 9.9 Setting a user-specific time zone for calculations

```
BEGIN;

SET LOCAL time zone 'Asia/Tokyo';

SELECT time_bucket('1 week', recorded_at, 'Asia/Tokyo',
  '2025-04-01'::timestamptz) AS period, activity,
  AVG(heart_rate)::int AS avg_rate,
  MAX (heart_rate)::int AS max_rate, MIN (heart_rate)::int AS min_rate
FROM watch.heart_rate_measurements
WHERE watch_id = 2 AND recorded_at >= '2025-04-01'AND
  recorded_at < '2025-04-15'
GROUP BY period, activity ORDER BY period, activity;

COMMIT;
```

The query also uses the `SET LOCAL` command to change the `time zone` setting from UTC (the default for our Postgres instance) to Asia/Tokyo. Because `SET LOCAL` is executed within an explicit transaction, the Asia/Tokyo time zone will apply only during this transaction, and its effect will disappear after the `COMMIT` statement is executed.

Postgres uses the Asia/Tokyo time zone for all operations within the boundaries of the transaction—including comparing the `recorded_at` column to the provided date range (`recorded_at >= '2025-04-01' AND recorded_at < '2025-04-15'`) and calculating the bucket boundaries in the `time_bucket` function. The result of `time_bucket`—stored in the `period` column—is also aligned to the Asia/Tokyo time zone. This transformation to Asia/Tokyo happens implicitly because both `recorded_at` and `period` are of the `timestamptz` data type, which means Postgres uses the time zone associated with the current database connection—or, as in our case, the current transaction.

Once we execute the query, the database returns the following weekly summaries:

```
         period         | activity | avg_rate | max_rate | min_rate
------------------------+----------+----------+----------+----------
 2025-04-01 00:00:00+09 | resting  |       77 |       79 |       75
 2025-04-01 00:00:00+09 | sleeping |       67 |       69 |       65
 2025-04-01 00:00:00+09 | walking  |       90 |       94 |       85
 2025-04-01 00:00:00+09 | workout  |      142 |      199 |      130
```

```
 2025-04-08 00:00:00+09 | resting  |       77 |       79 |       75
 2025-04-08 00:00:00+09 | sleeping |       67 |       69 |       65
 2025-04-08 00:00:00+09 | walking  |       90 |       94 |       85
 2025-04-08 00:00:00+09 | workout  |      145 |      198 |      130
(8 rows)
```

And if we take a closer look at the values in the `period` column, we can see that each one ends with `+09`, indicating that the returned timestamps are in the UTC+9 time zone, which corresponds to Asia/Tokyo.

> **NOTE** The example from listing 9.9 demonstrates one way an application can support users in different time zones by fully relying on Postgres's built-in capabilities. Another common approach is to convert the user's local time to UTC in the application layer before sending it to Postgres. In that case, the database performs all calculations in UTC.

9.5.2 Using the time_bucket_gapfill function

Even though our smartwatch application measures user heartbeats at regular intervals, it might not be able to send all measurements to the backend and database. Sometimes the user might take the watch off, and at other times the device could be disconnected from the network. As a result, we might miss some measurements for specific time periods in Postgres.

If the `time_bucket` function encounters a gap in the data, it simply skips the corresponding time bucket. For instance, the following query calculates the average heart rate for one-minute intervals within a specific time window defined in the `WHERE` clause.

Listing 9.10 Observing gaps in time buckets

```
SELECT watch_id, time_bucket('1 minute', recorded_at) AS minute,
  AVG(heart_rate)::int AS avg_rate
FROM watch.heart_rate_measurements
WHERE watch_id=1 AND recorded_at BETWEEN '2025-03-02 07:25'
  AND '2025-03-02 07:36'
GROUP BY watch_id, minute ORDER BY minute;
```

The query produces the following result, which shows that average heart rate information is missing for the buckets at 07:30, 07:31, and 07:35:

```
 watch_id |         minute         | avg_rate
----------+------------------------+----------
        1 | 2025-03-02 07:25:00+00 |      127
        1 | 2025-03-02 07:26:00+00 |      132
        1 | 2025-03-02 07:27:00+00 |      132
        1 | 2025-03-02 07:28:00+00 |      121
        1 | 2025-03-02 07:29:00+00 |      121
        1 | 2025-03-02 07:32:00+00 |      122
        1 | 2025-03-02 07:33:00+00 |      136
```

```
        1 | 2025-03-02 07:34:00+00 |      132
        1 | 2025-03-02 07:36:00+00 |      125
(9 rows)
```

These gaps exist because the device didn't send any measurements during those minutes. And in scenarios where users expect to see charts with continuous heart rate measurements, missing data points like these may not provide the best experience.

The TimescaleDB extension provides the `time_bucket_gapfill` function, which creates a continuous set of time buckets including the gaps. The following query uses `time_bucket_gapfill` instead of `time_bucket` to include the missing buckets in the result.

Listing 9.11 Adding time buckets for gaps

```
SELECT watch_id, time_bucket_gapfill('1 minute', recorded_at) AS minute,
  AVG(heart_rate)::int AS avg_rate
FROM watch.heart_rate_measurements
WHERE watch_id=1 AND recorded_at BETWEEN '2025-03-02 07:25'
  AND '2025-03-02 07:36'
GROUP BY watch_id, minute ORDER BY minute;
```

The output of the query now includes buckets for 07:30, 07:31, and 07:35:

```
 watch_id |         minute         | avg_rate
----------+------------------------+----------
        1 | 2025-03-02 07:25:00+00 |      127
        1 | 2025-03-02 07:26:00+00 |      132
        1 | 2025-03-02 07:27:00+00 |      132
        1 | 2025-03-02 07:28:00+00 |      121
        1 | 2025-03-02 07:29:00+00 |      121
        1 | 2025-03-02 07:30:00+00 |
        1 | 2025-03-02 07:31:00+00 |
        1 | 2025-03-02 07:32:00+00 |      122
        1 | 2025-03-02 07:33:00+00 |      136
        1 | 2025-03-02 07:34:00+00 |      132
        1 | 2025-03-02 07:35:00+00 |
        1 | 2025-03-02 07:36:00+00 |      125
(12 rows)
```

However, the heart rate values for those minutes are still empty because `time_bucket_gapfill` only adds the missing time buckets and doesn't fill them with any specific value.

TimescaleDB comes with additional functions that can be used alongside `time_bucket_gapfill` to fill the gaps with data. One of the functions is `LOCF`, which stands for *last observation carried forward*. This function takes the last known value and uses it as a replacement for missing measurement data. LOCF is best suited for scenarios where values tend to remain constant or change only slightly between measurements. The following query shows how to apply the `LOCF` function to the result of the `AVG(heart_rate)` aggregate that is calculated for each bucket.

Listing 9.12 Filling gaps with the `LOCF` function

```
SELECT watch_id, time_bucket_gapfill('1 minute', recorded_at) AS minute,
  LOCF(AVG(heart_rate)::int) AS avg_rate
FROM watch.heart_rate_measurements
WHERE watch_id=1 AND recorded_at BETWEEN '2025-03-02 07:25'
  AND '2025-03-02 07:36'
GROUP BY watch_id, minute ORDER BY minute;
```

The query returns the following result, where every time bucket is now assigned a value:

```
 watch_id |         minute         | avg_rate
----------+------------------------+----------
        1 | 2025-03-02 07:25:00+00 |      127
        1 | 2025-03-02 07:26:00+00 |      132
        1 | 2025-03-02 07:27:00+00 |      132
        1 | 2025-03-02 07:28:00+00 |      121
        1 | 2025-03-02 07:29:00+00 |      121
        1 | 2025-03-02 07:30:00+00 |      121
        1 | 2025-03-02 07:31:00+00 |      121
        1 | 2025-03-02 07:32:00+00 |      122
        1 | 2025-03-02 07:33:00+00 |      136
        1 | 2025-03-02 07:34:00+00 |      132
        1 | 2025-03-02 07:35:00+00 |      132
        1 | 2025-03-02 07:36:00+00 |      125
(12 rows)
```

In particular, the `avg_rate` for minutes 07:30 and 07:31 is set to `121`, which is the value carried forward from 07:29. Similarly, the `avg_rate` for 07:35 is set to `132`, which is carried over from 07:34.

In addition to `LOCF`, our smartwatch application can also use the `interpolate` function, which fills gaps by performing linear interpolation between the known previous and next values. This method works best when data tends to change smoothly over time. Here's how to use `interpolate` for gap-filling purposes.

Listing 9.13 Filling gaps with the `interpolate` function

```
SELECT watch_id, time_bucket_gapfill('1 minute', recorded_at) AS minute,
  interpolate(AVG(heart_rate)::int) AS avg_rate
FROM watch.heart_rate_measurements
WHERE watch_id=1 AND recorded_at BETWEEN '2025-03-02 07:25'
  AND '2025-03-02 07:36'
GROUP BY watch_id, minute ORDER BY minute;
```

After applying linear interpolation, the query returns the following result:

```
 watch_id |         minute         | avg_rate
----------+------------------------+----------
        1 | 2025-03-02 07:25:00+00 |      127
        1 | 2025-03-02 07:26:00+00 |      132
```

```
        1 | 2025-03-02 07:27:00+00 |      132
        1 | 2025-03-02 07:28:00+00 |      121
        1 | 2025-03-02 07:29:00+00 |      121
        1 | 2025-03-02 07:30:00+00 |      121
        1 | 2025-03-02 07:31:00+00 |      122
        1 | 2025-03-02 07:32:00+00 |      122
        1 | 2025-03-02 07:33:00+00 |      136
        1 | 2025-03-02 07:34:00+00 |      132
        1 | 2025-03-02 07:35:00+00 |      129
        1 | 2025-03-02 07:36:00+00 |      125
(12 rows)
```

TIP We've learned how to use some of the basic functions for analyzing and querying time-series data in Postgres. TimescaleDB also offers dozens of other specialized functions that you can explore in the official documentation: https://docs.timescale.com/api/latest/hyperfunctions/.

9.6 Using continuous aggregates

In addition to various functions for time-series data analysis, the TimescaleDB extension provides a feature called *continuous aggregates,* which stores a precomputed query result and refresh it in the background based on a predefined policy. Continuous aggregates are useful when an application runs analytical queries over large historical datasets or frequently queries data that updates less often than the query rate.

Imagine that our smartwatch application continuously monitors a user's heart rate in the background and sends an alert if the heart rate stays below 50 BPM during the last five-minute window. This capability is useful for detecting symptoms of bradycardia, a condition where the heart beats more slowly than expected. If the user receives such alerts regularly, it's a good idea to consult a doctor for an accurate diagnosis and appropriate care.

The smartwatch can handle this monitoring locally by analyzing heart rate measurements collected over the last five minutes, or it can rely on the application backend, which already receives measurements from the device at regular intervals and can implement the logic for detecting bradycardia symptoms. Let's explore how this can be supported by the application backend that will rely on the continuous aggregates of the TimescaleDB extension.

9.6.1 Creating and using aggregates

A continuous aggregate is a special type of Postgres materialized view that stores its precomputed result in a hypertable. These views can be queried like regular tables, and the hypertable ensures that the result is partitioned appropriately for efficient storage and querying.

TIP Refer to chapter 2 if you'd like to learn more about materialized views in Postgres.

Let's run the following query to create a continuous aggregate that counts the number of times a low heart rate is reported during five-minute windows.

Listing 9.14 Creating a continuous aggregate

```
CREATE MATERIALIZED VIEW watch.low_heart_rate_count_per_5min
WITH (timescaledb.continuous) AS
SELECT
  watch_id,
  time_bucket('5 minutes', recorded_at) AS bucket,
  MIN(heart_rate) as min_rate,
  COUNT(*) FILTER (WHERE heart_rate < 50) AS low_rate_count,
  COUNT(*) AS total_measurements
FROM watch.heart_rate_measurements
GROUP BY watch_id, bucket;
```

The aggregate is a special type of Postgres materialized view that does the following:

- The view is created by specifying the `timescaledb.continuous` parameter.
- It's populated with data computed for five-minute time buckets over the `recorded_at` column. The calculated buckets are stored in the `bucket` column.
- For every bucket, we calculate the minimal reported rate (`min_rate`), the number of heart rates below 50 BPM (`low_rate_count`), and the total number of measurements (`total_measurements`).
- This data is aggregated for every device separately (`GROUP BY watch_id, bucket`).

The created aggregate is already populated with data and stores the following number of records:

```
SELECT count(*) FROM watch.low_heart_rate_count_per_5min;

 count
--------
 288577
(1 row)
```

The data is stored in a hypertable, and we can use the `show_chunks` function to see the number of hypertable partitions:

```
SELECT show_chunks('watch.low_heart_rate_count_per_5min');
```

The function reports that the precomputed records are spread across three chunks (partitions):

```
               show_chunks
-----------------------------------------
 _timescaledb_internal._hyper_2_14_chunk
 _timescaledb_internal._hyper_2_15_chunk
 _timescaledb_internal._hyper_2_16_chunk
(3 rows)
```

Now, let's assume it's late at night on November 30, 2025. The user is sleeping with a smartwatch that continuously monitors their heart rate and sends it to our backend

system for analysis. The device receives a push notification stating that the user had a low heart rate within the five-minute interval starting at 2:35. The notification includes the payload that was retrieved from the `watch.low_heart_rate_count_per_5min` continuous aggregate using the following query.

Listing 9.15 Querying continuous aggregates

```
SELECT bucket, low_rate_count,
  (low_rate_count >= 5) AS bradycardia_detected
FROM watch.low_heart_rate_count_per_5min
WHERE bucket = '2025-11-30 02:35' AND watch_id = 3;
```

The query sets the `bradycardia_detected` flag to `true` if a low heart rate (below 50 BPM) was reported at least five times during the five-minute interval. The device sends a heart rate reading at least once every minute. Thus, if `low_rate_count >= 5`, it's highly likely that the user had a low heart rate during each minute of the five-minute interval.

The output for the query looks as follows, confirming that our application detected symptoms of bradycardia during the five minutes starting at 2:35:

```
         bucket         | low_rate_count | bradycardia_detected
------------------------+----------------+----------------------
 2025-11-30 02:35:00+00 |              5 | t
(1 row)
```

The device acknowledges the notification to the backend and sets up a timer to check whether the symptoms of bradycardia persist during the next 10 minutes. Once the timer fires, the smartwatch connects to the backend and asks it to return the data for three consecutive five-minute intervals between 2:35 and 2:45. The backend uses the following query to pull the required data from our `watch.low_heart_rate_count_per_5min` continuous aggregate.

Listing 9.16 Getting data for a 15-minute window

```
SELECT bucket, low_rate_count,
  (low_rate_count >= 5) AS bradycardia_detected
FROM watch.low_heart_rate_count_per_5min
WHERE bucket BETWEEN '2025-11-30 02:35' AND
  '2025-11-30 02:45' AND watch_id = 3
ORDER BY bucket;
```

The backend reports that the low heart rate persisted throughout the queried 15-minute window:

```
         bucket         | low_rate_count | bradycardia_detected
------------------------+----------------+----------------------
 2025-11-30 02:35:00+00 |              5 | t
 2025-11-30 02:40:00+00 |              5 | t
```

```
 2025-11-30 02:45:00+00 |               5 | t
(3 rows)
```

The device gently vibrates for a few seconds and displays an alert to make sure the user pays attention to the detected symptoms of bradycardia after waking up.

9.6.2 Refreshing aggregates

Next, assume it's late at night on December 1, 2025, and the device of the same user reports the following measurements to the application backend.

Listing 9.17 Inserting new heartbeat measurements

```
INSERT INTO watch.heart_rate_measurements
(watch_id, recorded_at, heart_rate, activity) VALUES
(3, '2025-12-01 00:45:00+00', 48, 'sleeping'),
(3, '2025-12-01 00:46:00+00', 46, 'sleeping'),
(3, '2025-12-01 00:47:00+00', 44, 'sleeping'),
(3, '2025-12-01 00:47:30+00', 47, 'sleeping'),
(3, '2025-12-01 00:48:00+00', 48, 'sleeping'),
(3, '2025-12-01 00:48:30+00', 45, 'sleeping'),
(3, '2025-12-01 00:49:00+00', 43, 'sleeping');
```

The measurements are inserted into the `watch.heart_rate_measurements` table and include the heart rate readings for the five-minute window starting at 00:45. However, even though the data is already inserted into the table, it doesn't automatically trigger an update of the continuous aggregate. Let's query the aggregate to return the data for the `2025-12-01 00:45` bucket:

```
SELECT bucket, low_rate_count,
  (low_rate_count >= 5) AS bradycardia_detected
FROM watch.low_heart_rate_count_per_5min
WHERE bucket = '2025-12-01 00:45' AND watch_id = 3;
```

The query returns an empty result:

```
 bucket | low_rate_count | bradycardia_detected
--------+----------------+----------------------
(0 rows)
```

The TimescaleDB extension allows us to define a policy that automatically refreshes a continuous aggregate. Let's use the following query to define a sample policy.

Listing 9.18 Defining a refresh policy for continuous aggregates

```
SELECT add_continuous_aggregate_policy
  ('watch.low_heart_rate_count_per_5min',
  start_offset => INTERVAL '15 minutes',
  end_offset => INTERVAL '1 minute',
  schedule_interval => INTERVAL '1 minute');
```

The `add_continuous_aggregate_policy` function creates the policy with the following parameters:

- `start_offset` tells TimescaleDB how far back from the *current time* it needs to refresh the continuous aggregate. In our case, the policy will refresh data starting 15 minutes before the current time.
- `end_offset` defines how far behind the *current time* the refresh needs to stop. In our case, the extension will stop refreshing one minute before the current time to avoid potential conflicts with ongoing inserts.
- `schedule_interval` controls how often the refresh policy runs. With this setting, the continuous aggregate will be refreshed every minute.

Because the policy works relative to the current time, and our dataset uses predefined timestamps, it's unlikely that the aggregate will be updated with the data inserted by the query from listing 9.17. Because of that, the following query still might return an empty result:

```
SELECT bucket, low_rate_count,
  (low_rate_count >= 5) AS bradycardia_detected
FROM watch.low_heart_rate_count_per_5min
WHERE bucket = '2025-12-01 00:45' AND watch_id = 3;

 bucket | low_rate_count | bradycardia_detected
--------+----------------+---------------------
(0 rows)
```

To make sure the aggregate can be refreshed for the dataset from the book, regardless of the current time, we can manually call the `refresh_continuous_aggregate` function.

Listing 9.19 Refreshing aggregates manually

```
CALL refresh_continuous_aggregate('watch.low_heart_rate_count_per_5min',
    '2025-12-01 00:45:00+00', '2025-12-01 00:50:00+00');
```

The function refreshes the aggregate with data recorded between 00:45 and 00:50 on 2025-12-01, which matches the measurements inserted by the query from listing 9.17.

After refreshing the aggregate, we can run the following query against the continuous aggregate again to confirm that it was refreshed:

```
SELECT bucket, low_rate_count,
  (low_rate_count >= 5) AS bradycardia_detected
FROM watch.low_heart_rate_count_per_5min
WHERE bucket = '2025-12-01 00:45' AND watch_id = 3;

         bucket         | low_rate_count | bradycardia_detected
------------------------+----------------+---------------------
 2025-12-01 00:45:00+00 |              7 | t
(1 row)
```

> **Using a data retention policy to drop data from aggregates**
>
> The TimescaleDB extension also allows us to set a data retention policy at the continuous aggregate level. Here's an example that shows how to create a seven-day retention policy on the aggregate data:
>
> ```
> SELECT add_retention_policy(
>    'watch.low_heart_rate_count_per_5min', INTERVAL '7 days');
> ```
>
> Once the policy is defined, the extension schedules a job that runs daily and drops partitions (chunks) of the continuous aggregate containing only data older than seven days. We can configure the job to run more or less frequently if necessary.

9.7 Indexing time-series data

After learning how to store and work with time-series data in Postgres, let's see how to optimize queries using the database's built-in indexing capabilities. As of now, we don't have any indexes on our table with heart rate measurements. We can easily confirm that by executing the following query.

Listing 9.20 Checking existing indexes on a table

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE schemaname = 'watch' AND tablename = 'heart_rate_measurements';
```

The query returns the following result:

```
 indexname | indexdef
-----------+----------
(0 rows)
```

This means Postgres performs a full table scan every time we query or analyze the stored time-series data. Considering that our application ingests large volumes of measurements at regular intervals, search performance can degrade over time as the data volume grows.

One of the features of our smartwatch application is to show daily statistics for a specific type of activity. For example, the following query calculates the average heart rate during walking, grouped into one-hour buckets.

Listing 9.21 Calculating the average heart rate while walking

```
SELECT
  time_bucket('1 hour', recorded_at) AS period,
  AVG(heart_rate)::int AS avg_rate
FROM watch.heart_rate_measurements
WHERE activity = 'walking' AND watch_id = 1
```

```
  AND recorded_at >= '2025-03-15 00:00'
  AND recorded_at < '2025-03-16 00:00'
GROUP BY period ORDER BY period;
```

The query produces the following result, showing each hour on 2025-03-15 when the user walked, even if only briefly:

```
         period         | avg_rate
------------------------+----------
 2025-03-15 06:00:00+00 |       80
 2025-03-15 08:00:00+00 |       79
 2025-03-15 09:00:00+00 |       79
 2025-03-15 10:00:00+00 |       80
 2025-03-15 11:00:00+00 |       79
 2025-03-15 12:00:00+00 |       81
 2025-03-15 13:00:00+00 |       80
 2025-03-15 14:00:00+00 |       79
 2025-03-15 15:00:00+00 |       80
 2025-03-15 16:00:00+00 |       80
 2025-03-15 17:00:00+00 |       80
 2025-03-15 19:00:00+00 |       79
 2025-03-15 20:00:00+00 |       80
 2025-03-15 21:00:00+00 |       80
(14 rows)
```

Now, let's add the `EXPLAIN (analyze, costs off)` statement to the query to see the execution plan selected by Postgres:

```
EXPLAIN (analyze, costs off)
SELECT
  time_bucket('1 hour', recorded_at) AS period,
  AVG(heart_rate)::int AS avg_rate
FROM watch.heart_rate_measurements
WHERE activity = 'walking' AND watch_id = 1
  AND recorded_at >= '2025-03-15 00:00'
  AND recorded_at < '2025-03-16 00:00'
GROUP BY period ORDER BY period;
```

The execution plan looks as follows, confirming that the database performs a full table scan (`Seq Scan`) due to the lack of indexes on the `watch.heart_rate_measurements` table:

```
                              QUERY PLAN
------------------------------------------------------------------------
 Finalize GroupAggregate (actual time=15.846..18.238 rows=14 loops=1)
   Group Key: (time_bucket('01:00:00'::interval,
➥   _hyper_1_4_chunk.recorded_at))
   ->  Gather Merge (actual time=15.797..18.210 rows=14 loops=1)
         Workers Planned: 1
         Workers Launched: 1
         ->  Partial GroupAggregate
➥             (actual time=6.498..6.536 rows=7 loops=2)
```

```
               Group Key: (time_bucket(
➥   '01:00:00'::interval, _hyper_1_4_chunk.recorded_at))
               ->  Sort (actual time=6.489..6.499 rows=118 loops=2)
                     Sort Key: (time_bucket(
➥   '01:00:00'::interval, _hyper_1_4_chunk.recorded_at))
                     Sort Method: quicksort  Memory: 32kB
                     Worker 0:  Sort Method: quicksort  Memory: 25kB
                     ->  Result (actual time=0.223..6.331 rows=118 loops=2)
                           ->  Parallel Seq Scan on _hyper_1_4_chunk
➥   (actual time=0.222..6.309 rows=118 loops=2)
                                 Filter: ((recorded_at >=
➥   '2025-03-15 00:00:00+00'::timestamp with time zone) AND
➥   (recorded_at < '2025-03-16 00:00:00+00'::timestamp with time zone)
➥   AND (activity = 'walking'::text) AND (watch_id = 1))
                                 Rows Removed by Filter: 62726
 Planning Time: 0.667 ms
 Execution Time: 18.335 ms
(17 rows)
```

The execution plan looks complex because Postgres decided to perform a parallel sequential scan by dividing the table's data into ranges and distributing them among cooperating processes. The leader process (which received the query for execution) initiates and drives the execution, scans the `_hyper_1_4_chunk` partition, and gathers and aggregates the final result. In addition, the database starts one parallel worker process (`Workers Launched: 1`) to help scan data from the same partition. The `loops=2` in the `Parallel Seq Scan on _hyper_1_4_chunk (... loops=2)` node implies that the partition was scanned by two processes: the leader and the parallel worker. Let's learn how to make the search more efficient by using Postgres's indexing capabilities.

9.7.1 Using a B-tree index

The B-tree index in Postgres fully supports date/time data types such as `date`, `time`, and `timestamp`. This means we can create a B-tree index on the `recorded_at` column of our `watch.heart_rate_measurements` table to speed up queries that filter, sort, and aggregate data based on the time stored in that column.

A single-column index on the `recorded_at` column is a good general-purpose solution for queries analyzing time-series data. It works well in most cases where the date/time column appears in a query condition.

However, if the date/time column isn't the only one that is frequently used in the query condition, it's reasonable to create a composite index that includes multiple columns. For example, our smartwatch application typically queries time-series data for a specific device using both the `watch_id` and `recorded_at` columns in the query filter. In this case, we can create a composite index on those columns by executing the following query.

Listing 9.22 Creating a composite index

```
CREATE INDEX heart_rate_btree_idx
ON watch.heart_rate_measurements (recorded_at, watch_id);
```

With this index, we can optimize queries that analyze time-series data based on the `recorded_at` column alone or on both `recorded_at` and `watch_id`.

Postgres uses the B-tree index type by default, and the composite index we created is not an exception. We can run the following query to confirm that the index is backed by the B-tree data structure:

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE schemaname = 'watch' AND tablename = 'heart_rate_measurements';
```

As we can see from the output, Postgres added the `USING btree` clause to the `CREATE INDEX` statement while creating the index:

```
      indexname        |                    indexdef
----------------------+---------------------------------------------
 heart_rate_btree_idx | CREATE INDEX heart_rate_btree_idx ON
                      | watch.heart_rate_measurements USING btree
                      | (recorded_at, watch_id)
```

Because our table is split into several partitions, Postgres creates a local index for each partition based on the data it owns. This can be easily validated using the next query, which returns all indexes created on the partitions belonging to our heart rate measurements table.

Listing 9.23 Checking indexes created on partitions

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE schemaname = '_timescaledb_internal'
  AND tablename LIKE '_hyper_1\_%_chunk' ESCAPE '\';
```

The truncated output showing results for the first three partitions is as follows:

```
               indexname                |              indexdef
---------------------------------------+------------------------------
 _hyper_1_1_chunk_heart_rate_btree_idx  | CREATE INDEX
➥   _hyper_1_1_chunk_heart_rate_btree_idx ON
➥   _timescaledb_internal._hyper_1_1_chunk USING
➥   btree (recorded_at, watch_id)

 _hyper_1_2_chunk_heart_rate_btree_idx  | CREATE INDEX
➥    _hyper_1_2_chunk_heart_rate_btree_idx ON
➥   _timescaledb_internal._hyper_1_2_chunk USING
➥   btree (recorded_at, watch_id)

 _hyper_1_3_chunk_heart_rate_btree_idx  | CREATE INDEX
➥    _hyper_1_3_chunk_heart_rate_btree_idx ON
➥   _timescaledb_internal._hyper_1_3_chunk USING
➥   btree (recorded_at, watch_id)
```

When we execute a query that filters by the `recorded_at` column, Postgres determines which partitions need to be accessed and then uses the local indexes on those partitions to speed up the search. Let's check the execution plan for the query from listing 9.21 again, which calculates the average heart rate during walking, grouped into one-hour buckets:

```
EXPLAIN (analyze, costs off)
SELECT
  time_bucket('1 hour', recorded_at) AS period,
  AVG(heart_rate)::int AS avg_rate
FROM watch.heart_rate_measurements
WHERE activity = 'walking' AND watch_id = 1
  AND recorded_at >= '2025-03-15 00:00'
  AND recorded_at < '2025-03-16 00:00'
GROUP BY period ORDER BY period;
```

The execution plan looks much simpler compared to the earlier plan with a full table scan, and the execution time is orders of magnitude faster with the index at 0.7 ms with the `Index Scan` versus 18 ms with the `Seq Scan`:

```
                              QUERY PLAN
-----------------------------------------------------------------------
 GroupAggregate (actual time=0.246..0.605 rows=14 loops=1)
   Group Key: (time_bucket('01:00:00'::interval,
➥      _hyper_1_4_chunk.recorded_at))
   ->  Result (actual time=0.195..0.557 rows=236 loops=1)
         ->  Index Scan using _hyper_1_4_chunk_heart_rate_btree_idx
➥    on _hyper_1_4_chunk (actual time=0.194..0.531 rows=236 loops=1)
               Index Cond: ((recorded_at >=
➥    '2025-03-15 00:00:00+00'::timestamp with time zone) AND
➥    (recorded_at < '2025-03-16 00:00:00+00'::timestamp with time zone)
➥    AND (watch_id = 1))
               Filter: (activity = 'walking'::text)
               Rows Removed by Filter: 1157
 Planning Time: 0.829 ms
 Execution Time: 0.703 ms
(9 rows)
```

The execution plan also shows that the database determined that all the required data is stored in the `_hyper_1_4_chunk` partition and performed an `Index Scan` using that partition's local index (`_hyper_1_4_chunk_heart_rate_btree_idx`).

> **NOTE** The execution time for queries using the `Index Scan` or `Seq Scan` access method may vary on your system depending on the available memory, CPU, and other resources.

Finally, because we're using a composite index, we can also optimize queries that don't specify `watch_id` in the search criteria and instead analyze heart rate measurements reported across all devices. For instance, the following statement shows the execution plan for a query that calculates the average heart rate during walking across all devices:

```
EXPLAIN (analyze, costs off)
SELECT
  time_bucket('1 hour', recorded_at) AS period,
  AVG(heart_rate)::int AS avg_rate
FROM watch.heart_rate_measurements
WHERE activity = 'walking'
  AND recorded_at >= '2025-03-15 00:00'
  AND recorded_at < '2025-03-16 00:00'
GROUP BY period ORDER BY period;
```

According to the execution plan, Postgres continued using the composite index, which is backed by a B-tree:

```
                              QUERY PLAN
------------------------------------------------------------------
 GroupAggregate (actual time=0.708..2.248 rows=17 loops=1)
   Group Key: (time_bucket('01:00:00'::interval,
➥       _hyper_1_4_chunk.recorded_at))
   ->  Result (actual time=0.617..2.131 rows=757 loops=1)
         ->  Index Scan using _hyper_1_4_chunk_heart_rate_btree_idx
➥   on _hyper_1_4_chunk (actual time=0.615..2.039 rows=757 loops=1)
               Index Cond: ((recorded_at >=
➥   '2025-03-15 00:00:00+00'::timestamp with time zone) AND
➥   (recorded_at < '2025-03-16 00:00:00+00'::timestamp with time zone))
               Filter: (activity = 'walking'::text)
               Rows Removed by Filter: 3423
 Planning Time: 0.752 ms
 Execution Time: 2.317 ms
(9 rows)
```

NOTE The order of columns in a composite index matters. Our composite index on (`recorded_at`, `watch_id`) can be used in queries that include conditions on the `recorded_at` column only, or on both `recorded_at` and `watch_id`. However, if a query skips the first column (`recorded_at`) and uses only `watch_id` in a condition, Postgres versions 17 and earlier won't be able to use the index. However, starting with Postgres 18, the database introduced support for skip scan lookups on composite B-tree indexes, allowing us to skip leading columns and still use the index in more scenarios. To learn more about composite indexes, refer to chapter 4.

9.7.2 Using BRIN indexes

The block range index (BRIN) is a specialized index type that works particularly well with time-series data. It can serve as an alternative to B-tree indexes, especially when index size matters. BRINs are designed for large tables where the values in certain columns already have some *strong correlation with their physical location* in the table's internal structure. Let's break down what this correlation with physical location means in practice by using the `recorded_at` column of our heart rate measurements table as an example.

Every time a device sends a new heart rate measurement, the measurement's `recorded_at` column is set to the current time, which is greater than the time of the previous measurement. The clock always moves forward.

The new measurement row is then added to the `watch.heart_rate_measurements` table, and Postgres places it physically into a page. The page size in Postgres is 8 KB, which is much larger than the size of any individual measurement record. As a result, each page will store dozens of records with measurements.

Also, because our time-series data is always appended to the table and never updated, the rows with heart rate measurements simply fill the available pages. Each new row added to a page will have a `recorded_at` time that is strongly correlated with the times of the rows already in that page. The time can be either slightly greater or slightly smaller than that of the previous rows, but the difference shouldn't deviate significantly—even when the page contains measurements from different devices.

For example, as figure 9.1 shows, each new row added to page 1 has a greater `recorded_at` value than the rows before it. Once page 1 is full and has no more room for new records, the database continues inserting measurements into the next available page. As a result, as figure 9.2 shows, each following page stores rows with later measurement times.

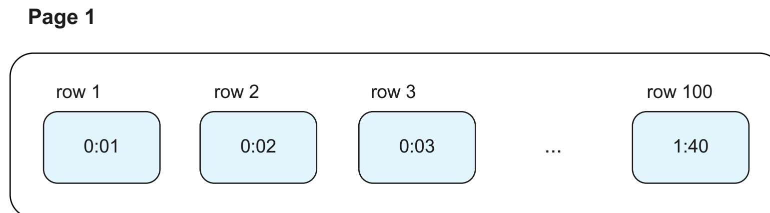


Figure 9.1 The measurement time (`recorded_at` column) is always greater in rows added last to the page.

NOTE We skip the year, month, and date portions of the timestamp in this example for simplicity. In reality, pages store the full timestamp, and BRIN uses the complete timestamp information in its structure.

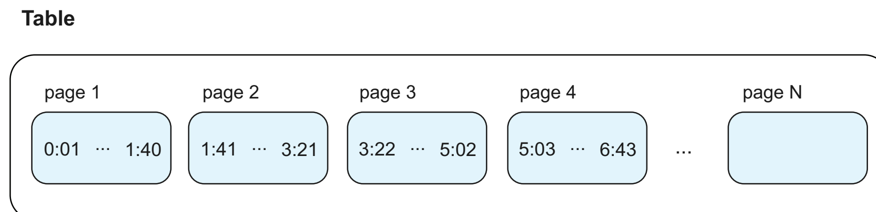


Figure 9.2 The measurement time (`recorded_at` column) of inserted rows continues to increase in the next available page.

For example, page 2 stores rows with measurement times greater than those in page 1. On page 2, the row with the minimum `recorded_at` time is 1:41, and the row with the maximum time is 3:21. All other rows have `recorded_at` values from 1:41 through 3:21. This trend continues with the following pages—page 3 stores rows with measurement times greater than those in pages 1 and 2. Considering this data correlation and its placement in Postgres storage, let's learn how the BRIN index is structured and how it works (see figure 9.3).

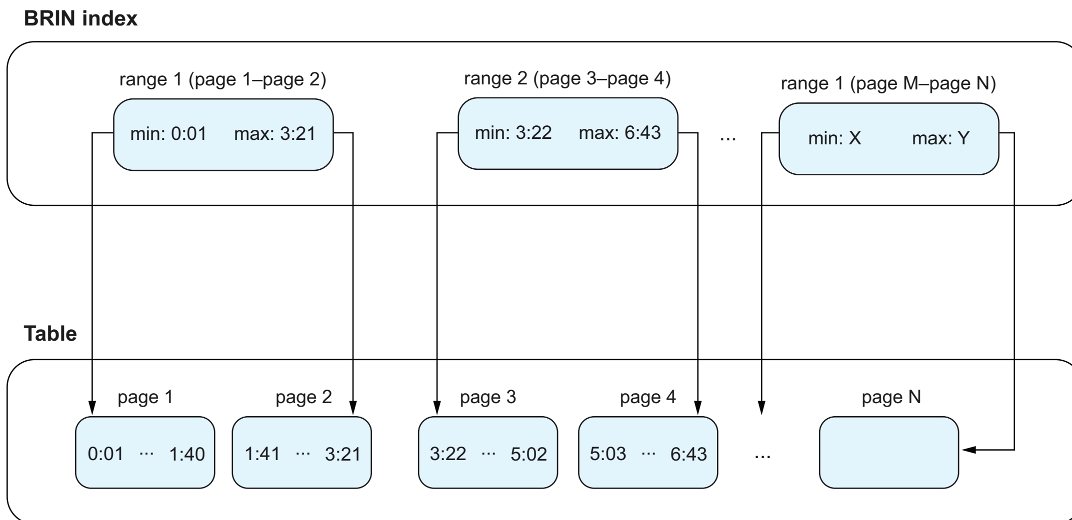


Figure 9.3 Sample BRIN index on the table with heart rate measurements

When we create a BRIN index, Postgres scans the measurements table and records the information about the minimum and maximum values of the `recorded_at` column for *ranges* of pages. In our example, each range covers two pages. As a result, range 1 stores the minimum and maximum `recorded_at` values across both page 1 and page 2. Specifically, the minimum value is 0:01, which comes from page 1, and the maximum is 3:21, which comes from page 2. The following ranges, such as range 2, continue to store the minimum and maximum values for their respective sets of pages.

Later, the BRIN index uses the information from the ranges to decide which pages to visit or skip during a search. For example, suppose our application needs to analyze measurements reported between 0:20 and 0:50 at night. Figure 9.4 shows how the BRIN index satisfies this search criterion.

In this case, the BRIN index optimizes search performance as follows:

1 It scans all the ranges and determines that only one range—range 1—satisfies the search condition. This is because range 1 has a minimum value of 0:01 and a

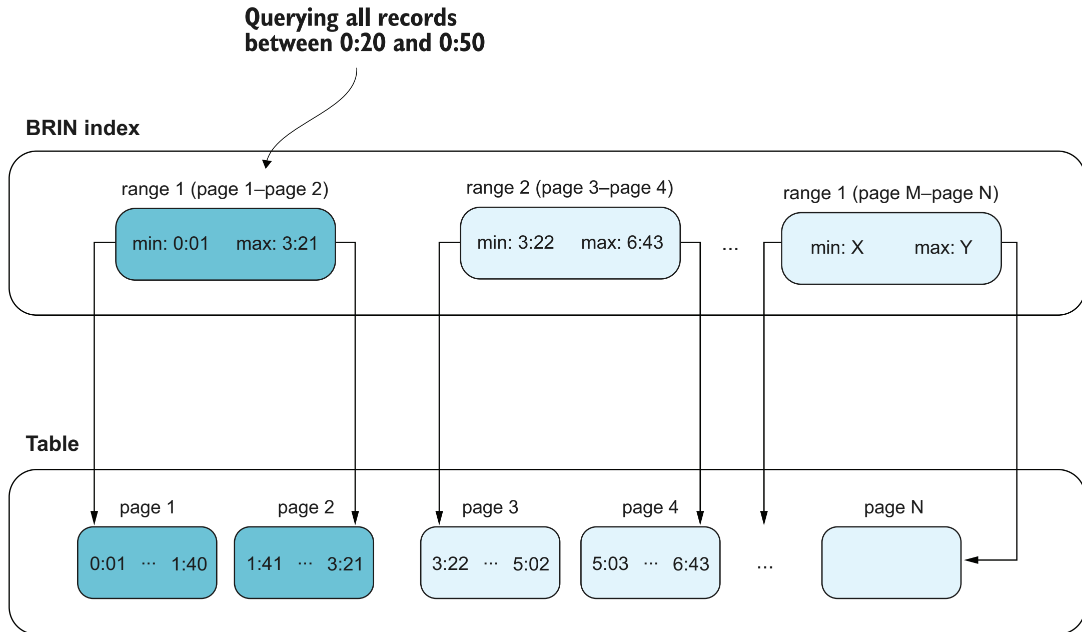


Figure 9.4 Searching for records with measurement times between 0:20 and 0:50

maximum of 3:21, and the search condition (between 0:20 and 0:50) falls within that range.

2. Because range 1 covers page 1 and page 2, Postgres scans only those two pages and skips the rest of the table.
3. The database then filters the rows from page 1 and page 2, returning only those that actually meet the condition (between 0:20 and 0:50).

Also, even though page 2 doesn't contain any matching rows (its times range from 1:41 to 3:21), Postgres still scans it because the BRIN index stores only the minimum and maximum values per range of pages and doesn't index actual timestamp values like a B-tree does. However, this small overhead allows the BRIN index to maintain a significantly smaller index size compared to a B-tree. Let's use the following query to create a BRIN index on the `recorded_at` column and see how much smaller the BRIN index is.

Listing 9.24 Creating a BRIN index

```
CREATE INDEX heart_rate_brin_idx
ON watch.heart_rate_measurements
USING brin (recorded_at);
```

Just as with the B-tree index, because our table is split into several partitions, Postgres creates a local BRIN index for each partition based on the data it owns. If we

execute the following query, we can see the sizes of all indexes created on the partitions (chunks) of the `heart_rate_measurements` table.

Listing 9.25 Comparing the sizes of B-tree and BRIN indexes

```
SELECT
  i.indexrelid::regclass AS index_name,
  pg_size_pretty(pg_relation_size(i.indexrelid)) AS index_size
FROM timescaledb_information.chunks c
JOIN pg_index i
  ON i.indrelid = format('%I.%I', c.chunk_schema, c.chunk_name)::regclass
WHERE c.hypertable_name = 'heart_rate_measurements'
ORDER BY index_name;
```

The truncated output looks as follows, showing that B-tree-based indexes occupy thousands of kilobytes (KB) of space, whereas BRIN indexes remain extremely compact, taking only 24 KB. This is a hundred-fold reduction in storage usage, and the difference between BRIN and B-tree index sizes will continue to grow as more measurements are inserted into the table:

```
                         index_name                            | index_size
---------------------------------------------------------------+------------
 _timescaledb_internal._hyper_1_1_chunk_heart_rate_btree_idx  | 1432 kB
 _timescaledb_internal._hyper_1_2_chunk_heart_rate_btree_idx  | 3896 kB
 _timescaledb_internal._hyper_1_3_chunk_heart_rate_btree_idx  | 3896 kB
 _timescaledb_internal._hyper_1_4_chunk_heart_rate_btree_idx  | 3896 kB
...truncated output (more B-tree indexes listed here)
 _timescaledb_internal._hyper_1_1_chunk_heart_rate_brin_idx   | 24 kB
 _timescaledb_internal._hyper_1_2_chunk_heart_rate_brin_idx   | 24 kB
 _timescaledb_internal._hyper_1_3_chunk_heart_rate_brin_idx   | 24 kB
 _timescaledb_internal._hyper_1_4_chunk_heart_rate_brin_idx   | 24 kB
...truncated output (more BRIN indexes listed here)
(26 rows)
```

Again, our B-tree indexes occupy more space because they store actual values of the `recorded_at` and `watch_id` columns in the index structure, whereas the BRIN index keeps only the metadata about the minimum and maximum values of the `recorded_at` column within a range of pages. This is exactly what makes BRIN indexes so much more compact than B-tree indexes.

However, because the B-tree index stores actual values of the indexed columns, it also includes a pointer (address) to the specific table rows those values belong to. This allows the B-tree index to access rows directly, without scanning through other records on the table pages as the BRIN index does. As a result, the B-tree index can still offer better performance than BRIN, especially when only a small subset of the data needs to be analyzed.

For instance, let's execute the following query to see the execution plan Postgres chooses to calculate the average sleeping heart rate across all users for a single day (`recorded_at >= '2025-03-15 00:00' AND recorded_at < '2025-03-16 00:00'`).

Listing 9.26 Execution plan for the average sleeping heart rate calculation

```
EXPLAIN (analyze, costs off)
SELECT
  time_bucket('1 hour', recorded_at) AS period,
  AVG(heart_rate)::int AS avg_rate
FROM watch.heart_rate_measurements
WHERE activity = 'sleeping'
  AND recorded_at >= '2025-03-15 00:00'
  AND recorded_at < '2025-03-16 00:00'
GROUP BY period ORDER BY period;
```

As long as the average is calculated for a single day and our dataset is still relatively small—with around 1.4 million records total and approximately 125,000 records per partition—the database prefers to use the B-tree index, which stores pointers to specific rows and can access them directly (`Index Scan using _hyper_1_4_chunk_heart_rate_btree_idx`):

```
                              QUERY PLAN
---------------------------------------------------------------------
 GroupAggregate (actual time=0.208..2.213 rows=10 loops=1)
   Group Key: (time_bucket('01:00:00'::interval,
➥     _hyper_1_4_chunk.recorded_at))
   ->  Result (actual time=0.047..2.038 rows=1510 loops=1)
         ->  Index Scan using _hyper_1_4_chunk_heart_rate_btree_idx
➥   on _hyper_1_4_chunk (actual time=0.046..1.883 rows=1510 loops=1)
               Index Cond: ((recorded_at >=
➥   '2025-03-15 00:00:00+00'::timestamp with time zone) AND
➥   (recorded_at < '2025-03-16 00:00:00+00'::timestamp with time zone))
               Filter: (activity = 'sleeping'::text)
               Rows Removed by Filter: 2670
 Planning Time: 0.831 ms
 Execution Time: 2.316 ms
(9 rows)
```

The execution time for the query is slightly more than 2 ms. Now, let's compare it to the execution time when using the BRIN index. To do that, we first need to drop the B-tree index:

```
DROP INDEX watch.heart_rate_btree_idx;
```

And if we execute the query from listing 9.26 again, we can see that Postgres now chooses the BRIN index to perform the calculation:

```
                              QUERY PLAN
----------------------------------------------------------------------
 Sort (actual time=8.406..8.409 rows=10 loops=1)
   Sort Key: (time_bucket('01:00:00'::interval,
➥     _hyper_1_4_chunk.recorded_at))
   Sort Method: quicksort  Memory: 25kB
```

```
  ->  HashAggregate (actual time=8.368..8.389 rows=10 loops=1)
        Group Key: time_bucket('01:00:00'::interval,
➥       _hyper_1_4_chunk.recorded_at)
        Batches: 1  Memory Usage: 40kB
        ->  Result (actual time=0.482..8.077 rows=1510 loops=1)
              ->  Bitmap Heap Scan on _hyper_1_4_chunk
➥                  (actual time=0.480..7.926 rows=1510 loops=1)
                    Recheck Cond: ((recorded_at >=
➥  '2025-03-15 00:00:00+00'::timestamp with time zone) AND
➥  (recorded_at < '2025-03-16 00:00:00+00'::timestamp with time zone))
                    Rows Removed by Index Recheck: 65452
                    Filter: (activity = 'sleeping'::text)
                    Rows Removed by Filter: 2670
                    Heap Blocks: lossy=512
                    ->  Bitmap Index Scan on
➥  _hyper_1_4_chunk_heart_rate_brin_idx
➥  (actual time=0.031..0.031 rows=5120 loops=1)
                          Index Cond: ((recorded_at >=
➥  '2025-03-15 00:00:00+00'::timestamp with time zone) AND
 v  (recorded_at < '2025-03-16 00:00:00+00'::timestamp with time zone))
 Planning Time: 0.510 ms
 Execution Time: 8.566 ms
(17 rows)
```

The execution time with the BRIN index for the given query is around 8 ms, which is longer than with the B-tree index (2 ms). This happens because Postgres needs to perform the following steps when using the BRIN index:

- `Bitmap Index Scan on _hyper_1_4_chunk_heart_rate_brin_idx`—The database uses the BRIN index on the `_hyper_1_4_chunk` partition to find page ranges whose minimum and maximum values satisfy the condition on the `recorded_at` column from the query. As a result, the BRIN index prepares a bitmap of table pages that need to be visited.
- `Bitmap Heap Scan on _hyper_1_4_chunk`—Postgres accesses the `_hyper_1_4_chunk` partition that stores the pages from the bitmap and scans through them. Over 65,000 rows stored in those pages didn't satisfy the search condition (`Rows Removed by Index Recheck: 65452`). Those rows didn't match the condition from the `WHERE` clause on either the `recorded_at` or `activity` column.
- `HashAggregate`—The bitmap heap scan returned 1,510 rows that satisfied the `WHERE` clause (see `rows=1510` in the `Bitmap Heap Scan` step). That result is then aggregated and sorted by the database.

This example doesn't imply that the B-tree index is always more performant than the BRIN index. A BRIN index can match the performance of a B-tree index if the data volume or query conditions change. The main reason we're comparing the size and performance characteristics of these two index types is to show that Postgres provides multiple options to choose from, allowing us to make better choices based on a particular time-series workload.

Using a columnar store to optimize queries on large time intervals

If our application starts analyzing time-series data over large or historical time intervals, such as months or years, the database may stop using both B-tree and BRIN indexes, as it becomes more efficient to perform a full table scan over the partitions storing the required data. In such cases, we can benefit from the columnar storage capabilities of the TimescaleDB extension by storing data from specific partitions in a columnar format optimized for analytics.

For example, we might continue storing the most recent three months of measurements in the row store and maintain indexes only on that subset of data. Records older than three months can be moved to the columnar store and left unindexed.

However, this goes beyond the scope of this book. You're encouraged to explore TimescaleDB's columnar store, called Hypercore, by visiting the official documentation: https://docs.timescale.com/use-timescale/latest/hypercore/.

Summary

- Postgres can be easily used for time-series workloads that need to process data ingested both continuously and at irregular intervals.
- Postgres's table partitioning feature allows us to split a large table with time-series data into partitions, each holding a subset of the dataset. This makes queries more efficient by targeting only the partitions that satisfy the search criteria.
- The TimescaleDB extension turns Postgres into a time-series database by automatically partitioning tables, providing specialized functions for querying time-series data, supporting a columnar storage engine optimized for analytical queries, and much more.
- The specialized functions are designed to efficiently process and analyze large volumes of time-series data while maintaining high performance.
- The `time_bucket` function simplifies the aggregation of time-series data by grouping timestamps into buckets, which are useful for rollups and aggregations.
- The `time_bucket_gapfill` function creates a continuous set of time buckets, including gaps that can be filled using the `LOCF` or `interpolation` function.
- TimescaleDB supports continuous aggregates, which store a precomputed query result and refresh it in the background based on a predefined policy.
- The extension allows us to define a retention policy for both hypertables and continuous aggregates that runs automatically and discards data older than a specified interval.
- Queries over time-series data can be optimized using both B-tree and BRIN index types, letting us find the right balance between index size and performance.

10 Postgres for geospatial data

This chapter covers

- Exploring Postgres capabilities for geospatial workloads
- Using the PostGIS extension to store and query location-based data
- Visualizing geographic information with QGIS
- Optimizing the search with spatial indexes

Spatial data refers to information that represents the position and shape of objects in geometric space. Examples include points, lines, and polygons defined in two- or three-dimensional Euclidean space. *Geospatial data* is a subset of spatial data that specifically refers to geographic information (geodata) associated with locations on the Earth's surface. Think of the physical location of a car, or the size and position of a thunderstorm, expressed in latitude and longitude coordinates.

Let's explore how to use Postgres for geospatial data and workloads as we work with an OpenStreetMaps dataset for Florida, the Sunshine State of the United States

of America. We'll learn how to store, query, and process the geodata in Postgres while exploring natural wonders, bustling cities, and world-famous amusement parks across Florida.

10.1 How Postgres works with geodata

Geospatial data stores the location and additional information about an object on the planet's surface. The stored data can be static, dynamic, or even temporal in nature, depending on whether the position or other attributes of the object change.

Most of us take advantage of geospatial data when navigating the streets of the places we live in or exploring new places we visit. Imagine that you're traveling to Miami, Florida. It's Sunday morning. You wake up eager to plan your day. You open Google Maps or a similar application and look at the city map, which is a product of several layers of geodata combined:

- The location and shape of areas, buildings, parks, and other objects are defined by *closed polygons*. In geometry, a closed polygon is a two-dimensional (2D) figure formed by connecting straight lines. For instance, some of the buildings near you might be represented as rectangles: polygons made up of four lines with opposite sides equal in length and parallel to each other. You might also see parks and lakes nearby, which are examples of polygons composed of many interconnected lines of varying lengths and directions, accurately defining the shape of the object. Finally, as you start zooming out, you'll see the boundaries of Miami, then the boundaries of the state of Florida, along with small parts of the Gulf of Mexico and Atlantic Ocean that border the state. The boundaries of cities and states, as well as large water bodies like the Gulf of Mexico and the Atlantic Ocean, are all examples of geospatial data represented as polygons. All the geographic objects discussed are examples of *static geodata*; their location or shape remains unchanged (like the boundaries of a building) or changes only rarely (like the boundaries of a neighborhood).
- Points of interest, amenities, and various landmarks that you see on the map make up another layer of geodata, typically represented as *points* with two coordinates. For example, cafes and restaurants are shown as points on the map, along with associated descriptions such as name, street address, customer reviews, and rating. Notably, these spots (points) are located in buildings (polygons), and several places might be in the same building. This type of geodata can be considered either *static* or *temporal*, depending on the time span you're looking at. For instance, some businesses may operate in a neighborhood for generations, whereas others come and go.
- Roads and trails you'll take from the hotel to a specific point of interest are represented as a series of *line segments*, each defined by two points and a length. The location-related information about roads and trails is typically static or temporal in nature, because these objects don't undergo major reconstruction on a

regular basis. However, live traffic data is an example of dynamic geodata—it's tied to specific roads and changes as traffic conditions improve or deteriorate.

What you see on Google Maps or other mapping platforms is just one example of how geospatial data is used in practice. The same geographic objects—the city of Miami with its neighborhoods, lakes, buildings, roads, and more—can be associated with other types of geodata, used by different groups of people for various purposes.

For example, each neighborhood in Miami has sociodemographic data associated with it that can be accessed and analyzed by local governments, scientists, businesses, and individuals. This type of data refers to the social and demographic characteristics of a population or group living in a specific geographic area. These factors can include age, gender, ethnicity, education, and income levels. As a result, each neighborhood in Miami can be represented as a polygon that stores the area's sociodemographic information.

Apart from that, Miami is located in the southern part of Florida, which has a tropical monsoon climate with hot summers, short winters, and a hurricane season. There is geodata that stores information about temperature, sea levels, wind strength, and other natural factors over specific time frames. Moreover, this data can be collected not only for the entire city of Miami but also at more granular levels, such as neighborhoods, communities, or even individual buildings. This makes it possible to analyze and forecast the effect of natural events or climate changes on the city as a whole or on specific areas.

These examples of geographic information are often combined by layering one on top of another. We see this in Google Maps, where roads and buildings appear over city and neighborhood boundaries, or where cafes and landmarks are shown as part of buildings, parks, or other areas. At the same time, a special class of software called geographic information systems (GISs) allows us to load, analyze, edit, and visualize geographic data for a variety of purposes.

Suppose Whole Foods, an American grocery chain specializing in organic food, is planning to open another location in Miami. The company can use a GIS and available geodata to identify the best possible location. In particular, the GIS can be used to load and analyze the following geographic data:

- The base layer of geodata defines the boundaries of Miami, including its districts, roads, and buildings. All other layers of data will be placed on top of this foundation.
- The second layer can be sourced from geodata that includes existing Whole Foods locations in Miami as well as the locations of the company's competitors. This information helps the company identify areas where it wants to grow or is willing to compete.
- Sociodemographic data serves as another layer, allowing the company to pinpoint areas with its primary customer base, as well as neighborhoods undergoing population shifts that could either hinder or boost business performance.

- The latest information about nearby roads' load and congestion helps Whole Foods assess how easily customers can access the future store, as well as how efficiently the company's trucks can deliver products from the nearest warehouses.
- Overlaying historical flooding and surf data on top of the shortlisted areas for the future store allows the company to better forecast and plan for risks associated with hurricanes.

With that, we now have a solid understanding of the nature of geospatial data and how it can be used in practice. Next, let's see how a GIS, Google Maps, or another application can use Postgres to store and process geographic information.

10.1.1 Built-in Postgres capabilities

As discussed, objects like buildings, lakes, city boundaries, or points of interest can be represented using geometric shapes. Postgres provides seven built-in geometric data types for working with 2D spatial data. These types are listed in table 10.1.

Table 10.1 Postgres geometric data types

Type name	Description	Input format
`point`	A point on a 2D plane	`(x, y)`
`line`	An infinite straight line in 2D space	`{A, B, C}` where `Ax + By + C = 0`
`lseg`	A finite line segment between two points	`[(x1, y1), (x2, y2)]`
`box`	A rectangle defined by two opposite corners	`((x1, y1), (x2, y2))`
`path`	An open or closed path defined as a sequence of connected points	open path: `[(x1, y1), (x2, y2), ...]` closed path: `((x1, y1), (x2, y2), ...)`
`polygon`	A closed polygon with three or more points	`((x1, y1), (x2, y2), ...)`
`circle`	A circle with a center and a radius	`<(x, y), r>`

Using these types, we can store information about geographic objects of varying complexity. For instance, the boundaries of Miami can be stored as a `polygon`, the city's roads can be represented as line segments using the `lseg` type, and the coordinates of points of interest can be stored using the `point` type.

In addition to the listed data types, Postgres supports special operators and functions that can be used to perform various operations on geometric objects. For example, the `<->` operator can be used to calculate the distance between two objects in 2D space. Suppose the location of your hotel is defined as `point(10, 15)` and the nearest coffee shop is at `point(20, 17)`. The distance between them can be calculated as follows:

```
(10,15) <-> (20,17)
```

If we'd like to check whether the coffee shop is located in a particular building, we can use the `<@` operator, which checks if the first object is contained within the second. Assuming the building is defined as `polygon((0,0),(0,30),(50,30),(50,0))`, we can confirm that the point representing the coffee shop's location is indeed contained within the building as follows:

```
(10,15) <@ ((0,0),(0,30),(50,30),(50,0))
```

However, even though Postgres provides a rich set of data types, operators, and functions for working with geometric objects, these capabilities are not enough for real-world geospatial applications. Such applications need to work with coordinates from one of the standard spatial reference systems, account for the Earth's curvature in calculations, integrate seamlessly with GISs, and more. This goes beyond Postgres's built-in support for geometric data types but is easily enabled with PostGIS, the extension that we're going to look into next.

10.1.2 Extended capabilities with PostGIS

PostGIS is an extension that turns Postgres into a geospatial database. It implements the standards of the Open Geospatial Consortium (OGC) by supporting fundamental `geometry` and `geography` data types, adds a wide range of functions for geospatial data querying and analysis, expedites the search using Postgres's indexing capabilities, and integrates with third-party tools for geodata visualization and processing.

Let's briefly go over the `geometry` and `geography` data types of PostGIS and then dive into other capabilities of the extension in the following sections. The `geometry` data type uses an Euclidean plane to represent geospatial objects and perform calculations such as distance, area, and intersection. It requires specifying a geometric subtype such as `Point`, `LineString`, or `Polygon`, and allows us to define one of the supported coordinate systems for spatial calculations and projections.

For example, if we want to store the coordinates of points of interest and landmarks using the `geometry` type with the Web Mercator projection, the full data type definition for a column named `location` would look as follows:

```
CREATE TABLE coordinates_plane (
    location geometry(Point, 3857)
);
```

Here, 3857 is the SRID (spatial reference system identifier) of the Web Mercator projection, which is used by Google Maps, OpenStreetMap, and many other applications to render and work with geographic data on a flat plane. This system uses meters as the unit of measurement when calculating the distance between two points or performing other computations.

The `Point` type expects *x* and *y* coordinates in the selected coordinate system. This is how we can add the coordinates for the center of downtown Miami in the Web Mercator projection:

```
INSERT INTO coordinates_plane (location)
VALUES (
  ST_MakePoint(-8930649.94197805, 2972156.8479390536)
);
```

Calculations on `geometry` types use Cartesian mathematics, which makes them easier to implement and faster to execute. However, they can be less accurate due to the curvature of the Earth, especially when a computation takes place over large distances. In this book, we'll primarily focus on using this data type.

The `geography` data type of the PostGIS extension, on the other hand, takes the Earth's curvature into account by using a spherical model for calculations. Its default coordinate system is WGS 84, which stands for World Geodetic System 1984. This system has an SRID of 4326 and represents coordinates as degrees of longitude and latitude—the same format used by the Global Positioning System (GPS). All calculations such as distance and area are performed using meters as the unit of measurement.

The following example shows how to create the `location` column storing a `Point` of the `geography` type in the WGS 84 coordinate system:

```
CREATE TABLE coordinates_sphere (
    location geography(Point, 4326)
);
```

Note that 4326 is optional and can be omitted, as it's the default SRID used by the `geography` type. The `Point` type expects longitude as the first argument and latitude as the second. With that in mind, here's how we can insert the coordinates for the center of downtown Miami:

```
INSERT INTO coordinates_sphere (location)
VALUES (
  ST_MakePoint(-80.2253934, 25.782414)
);
```

Calculations on the `geography` type take the spheroidal shape of the Earth into account, making them more accurate than those performed with the `geometry` type. However, because the underlying math for the `geography` type is more complex, some computations can be slower, especially on large datasets.

Now that we understand how Postgres can store and work with geospatial data, let's start a database container with the PostGIS extension, load a sample dataset for our experiments, and explore the extension in action.

10.2 Starting Postgres with PostGIS

Just like the other extensions we've explored in previous chapters, PostGIS can be deployed in a Docker container. So let's go ahead and provision a Postgres instance with the extension using one of the available PostGIS Docker images.

Stop the Postgres container used in previous chapters

In previous chapters, we used Postgres containers that don't include the PostGIS extension. If you have any of those Postgres containers running, stop them to free port 5432. First, find any running Postgres container that is listening on port 5432:

```
docker ps --filter "name=postgres" --filter "publish=5432"
```

The command might output a result similar to the following, indicating that a container is currently running:

```
IMAGE            STATUS         PORTS                      NAMES
postgres:latest  Up 50 minutes  0.0.0.0:5432->5432/tcp     <container-name>
```

Stop the container by providing its name in the `<container-name>` placeholder:

```
docker container stop <container-name>
```

At the time of writing, there was no official multiarch Docker image for PostGIS that supported both amd64 (x86_64) and arm64 architectures. An amd64 image was the only available option. So, start the container using one of the commands from listing 10.1, 10.2, or 10.3, depending on your operating system and instruction set architecture.

If you're on a Unix-based operating system such as Linux or macOS with a CPU supporting the amd64 instruction set, use the following command to start the container with PostGIS.

Listing 10.1 Starting Postgres with PostGIS on Unix with amd64

```
docker volume create postgres-postgis-volume

docker run --name postgres-postgis \
    -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password \
    -p 5432:5432 \
    -v postgres-postgis-volume:/var/lib/postgresql/data \
    -d postgis/postgis:17-3.5
```

If you're on a Mac with Apple Silicon (arm64 instruction set), use the command from listing 10.2, which starts the container in amd64 emulation mode (see the `--platform linux/amd64` parameter in the Docker command). This option works on macOS that supports Rosetta 2—a dynamic binary translator that allows apps compiled for the amd64 architecture to run on Apple Silicon.

Listing 10.2 Starting Postgres with PostGIS on a Mac with Apple Silicon

```
docker volume create postgres-postgis-volume

docker run --platform linux/amd64 --name postgres-postgis \
```

```
    -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password \
    -p 5432:5432 \
    -v postgres-postgis-volume:/var/lib/postgresql/data \
    -d postgis/postgis:17-3.5
```

Finally, if you're a Windows user, use the following command in PowerShell. If you use Command Prompt (CMD), replace each backtick (`) with a caret (^) at the end of each line:

Listing 10.3 Starting Postgres with PostGIS on Windows

```
docker volume create postgres-postgis-volume

docker run --name postgres-postgis `
    -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password `
    -p 5432:5432 `
    -v postgres-postgis-volume:/var/lib/postgresql/data `
    -d postgis/postgis:17-3.5
```

The commands start the `postgres-postgis` container using the `postgis/postgis:17-3.5` image that includes the PostGIS extension. The container listens for incoming connections on port 5432 and stores Postgres data in the Docker-managed volume named `postgres-postgis-volume`.

> **NOTE** The way we deploy Postgres in Docker works well for development and for exploring the database's capabilities. However, if you plan to use this deployment option in production, be sure to review security and other deployment best practices.

Once the container is started, connect to it using the psql client:

```
docker exec -it postgres-postgis psql -U postgres
```

Confirm that the PostGIS extension exists in the Postgres container by executing the following command.

Listing 10.4 Checking the available PostGIS version

```
SELECT * FROM pg_available_extensions
WHERE name = 'postgis';
```

The output should look similar to the following:

```
  name   | default_version | installed_version |        comment
---------+-----------------+-------------------+------------------------
 postgis | 3.5.2           | 3.5.2             | PostGIS geometry and
         |                 |                   | geography spatial
         |                 |                   | types and functions
```

Because the `installed_version` column is set to a specific version, it means that PostGIS is already enabled for the `postgres` database we're connected to. You can always check the current database using the `\conninfo` command, which returns output like this:

```
\conninfo

You are connected to database "postgres" as user "postgres" via socket in
➥ "/var/run/postgresql" at port "5432".
```

With that, we're ready to move on to the next step by preloading a dataset for Florida from OpenStreetMap.

10.3 Loading the OpenStreetMap dataset

Imagine that you're heading to Florida in a week with your family for a long-awaited vacation. You plan to start your trip in Tampa, a hidden gem on the Gulf of Mexico, and then take a road trip across the state to Miami, a bustling city on the Atlantic coast. You want to make the most of the trip and plan your stops in advance. And you'd like to do it the developer way by analyzing a dataset for Florida from OpenStreetMap.

> **What is OpenStreetMap?**
>
> OpenStreetMap (OSM) is an open source project maintained by a global community of contributors who create and update a free map of the world. You can explore and even edit the map by visiting www.openstreetmap.org.
>
> In addition to the interactive world map, you can download the underlying data for the entire world, specific countries or regions, or even individual objects such as buildings or roads. One such resource is the OpenStreetMap Data Extracts website, which makes it easy to select and download data for specific continents, countries, or regions: https://download.geofabrik.de.

Let's use the `osm2pgsql` open source tool to import OSM data for Florida into the running Postgres instance with the PostGIS extension. The tool can be easily deployed as a Docker container and will import the dataset from a provided URL. All you need to do is run one of the commands from listing 10.5 or 10.6, depending on whether you're on Windows or a Unix-based operating system such as Linux or macOS.

First, use the following command to create a schema named `florida` in your Postgres container. The `osm2pgsql` tool will load the dataset into this schema:

```
CREATE SCHEMA florida;
```

Next, if you're on Linux or macOS, use the following commands to download and import the data into Postgres.

Listing 10.5 Preloading the OSM dataset on Unix

```
docker volume create osm2pgsql-volume

docker run --name osm2pgsql --network="host" \
 -e PGPASSWORD=password \
 -v osm2pgsql-volume:/data \
 iboates/osm2pgsql:2.1.1 \
 -H 127.0.0.1 -P 5432 -d postgres -U postgres --schema florida \
 http://d3e4uq6jj8ld3m.cloudfront.net/florida-250501.osm.pbf
```

If you're on Windows, then use the command from the following listing instead.

Listing 10.6 Preloading the OSM dataset on Windows

```
docker volume create osm2pgsql-volume

docker run --name osm2pgsql --network="host" `
 -e PGPASSWORD=password `
 -v osm2pgsql-volume:/data `
 iboates/osm2pgsql:2.1.1 `
 -H 127.0.0.1 -P 5432 -d postgres -U postgres --schema florida `
 http://d3e4uq6jj8ld3m.cloudfront.net/florida-250501.osm.pbf
```

Both commands spin up an `osm2pgsql` container that downloads the OSM dataset for Florida as of May 1, 2025, and then imports it into the database. Note that, depending on your machine's resources, the entire process may take around 10 minutes to complete.

> **TIP** The dataset was originally downloaded from https://download.geofabrik.de, a server that regularly extracts data from the OSM project. You can find another version of the dataset on the following page if you'd like to work with the latest version of the Florida dataset or if the URL used in the book no longer works for you: https://download.geofabrik.de/north-america/us/florida.html.

Once the dataset is preloaded, use the following query to view the tables created by the `osm2pgsql` tool.

Listing 10.7 Listing tables created for the dataset

```
SELECT tablename
FROM pg_catalog.pg_tables
WHERE schemaname = 'florida';
```

The output will look as follows:

```
      tablename
----------------------
 osm2pgsql_properties
 planet_osm_roads
 planet_osm_point
 planet_osm_line
```

```
 planet_osm_polygon
(5 rows)
```

The `osm2pgsql_properties` table stores the properties related to `osm2pgsql` that were used during data import. The other four tables store actual geospatial data for points of interest, roads, areas, and other places in Florida.

All `planet_osm_…` tables contain dozens of columns that define the characteristics of an OSM object at a granular level. For instance, if we execute the `\d florida.planet_osm_point` command, we can see the columns of that table:

```
                      Table "florida.planet_osm_point"
       Column        |         Type          | Collation | Nullable | Default
--------------------+----------------------+-----------+----------+---------
 osm_id              | bigint                |           |          |
 access              | text                  |           |          |
 addr:housename      | text                  |           |          |
 addr:housenumber    | text                  |           |          |
 addr:interpolation  | text                  |           |          |
 admin_level         | text                  |           |          |
 aerialway           | text                  |           |          |
..other columns
```

Not all columns are used by every record stored in the table. A column is set to a value only if its data is relevant or has some relation to the respective OSM object.

> **TIP** Explore the OpenStreetMap Wiki to learn more about the purpose of the columns and their possible values: https://wiki.openstreetmap.org/wiki/Map_features.

If we inspect the columns of other `planet_osm_…` tables using the `\d` command, we'll see that most columns are the same across them. One such column present in every table is `way`, which stores the coordinates of the OSM object. However, each table uses a different data type to define those coordinates. Let's explore the data types in more detail.

10.3.1 Exploring tables with points

The `planet_osm_point` table stores all OSM nodes that can be represented as single points on the map. Examples of such nodes include restaurants, schools, bus stops, mountain peaks, and traffic signals.

If we execute the `\d florida.planet_osm_point` command and scroll to the end of the output by pressing Enter, we'll see the data type used by the `way` column, which defines the coordinates of OSM points:

```
                      Table "florida.planet_osm_point"
..other columns

 way                 | geometry(Point,3857) |           |          |
Indexes:
    "planet_osm_point_way_idx" gist (way) WITH (fillfactor='100')
```

The data type is `geometry(Point, 3857)`, meaning that every OSM node is defined as a `Point` of the `geometry` type with the coordinates stored in the Web Mercator projection (SRID 3857).

Next, let's see what some sample points from the table actually look like. Execute the `\x on` command to switch to extended display mode, and then run the following query, which returns coordinates for three arbitrary Whole Foods stores in Florida.

Listing 10.8 Getting coordinates for three stores

```
SELECT name, shop, way as WKB, ST_AsText(way) as WKT
FROM florida.planet_osm_point
WHERE name = 'Whole Foods Market' LIMIT 3;
```

The query output looks as follows:

```
-[ RECORD 1 ]-------------------------------------------
name | Whole Foods Market
shop | supermarket
wkb  | 0101000020110F0000ABD7121A209161C11F4787E7DDCD4841
wkt  | POINT(-9210112.814800104 3251131.8088158513)
-[ RECORD 2 ]-------------------------------------------
name | Whole Foods Market
shop | supermarket
wkb  | 0101000020110F00008EBF1DA1A20761C114742A951F7D4741
wkt  | POINT(-8928533.03488138 3078719.165358076)
-[ RECORD 3 ]-------------------------------------------
name | Whole Foods Market
shop | supermarket
wkb  | 0101000020110F0000B274A5B35E8261C1C8BF2E2E1D334841
wkt  | POINT(-9179893.613947246 3171898.3608016707)
```

As mentioned earlier, PostGIS follows the OGC standards, which also define two formats for representing geometry values: well-known text (WKT) and well-known binary (WKB). The output from the WKB and WKT columns shows how the coordinates of a `Point` are represented in each format.

> **TIP** Use the `\x on` meta-command in psql to display data in a format similar to the output for listing 10.8. This command enables expanded display mode, which shows each row vertically. It's helpful for results with many columns or large column values, making the data easier to read. To revert to the default tabular display mode, use the `\x off` command, which displays query results in a tabular format with rows and columns aligned horizontally.

10.3.2 Exploring tables with ways

The next `planet_osm_line` table stores OSM ways (line segments) that do not form closed loops. Examples include highways, roads, paths, rivers, and railways. The `planet_osm_roads` table stores a subset of `planet_osm_line`, optimized for rendering at low zoom levels. It usually includes segments of long roads or small local pathways.

Both tables store the location of the ways as a `LineString` of the `geometry` type. A `LineString` defines a path between the start and end of an OSM way as an ordered series of two or more points. In our case, the location is stored in the `way` column with the `geometry(LineString, 3857)` data type, which you can confirm by inspecting the tables with the `\d` command.

Let's see how a sample `LineString` is defined in the Web Mercator's projection by executing the following query, which returns the shortest named ways.

Listing 10.9 Getting the shortest named ways

```
SELECT name, ST_AsText(way)
FROM florida.planet_osm_line
WHERE name IS NOT NULL
ORDER BY ST_Length(way) ASC LIMIT 3;
```

The query uses the `ST_Length` function to calculate the 2D Cartesian length of each line segment, orders the result, and returns the three shortest ones:

```
-[ RECORD 1 ]------------------------------------------------------------
name      | Rice Creek Conservation Area
st_astext | LINESTRING(-9103060.734422492 3455483.5395689947,
 ➥   -9103060.845741984 3455483.731658956,
 ➥   -9103060.734422492 3455483.731658956,
 ➥   -9103060.734422492 3455483.5395689947)
-[ RECORD 2 ]------------------------------------------------------------
name      | Federal Highway
st_astext | LINESTRING(-8913795.280516023 3051428.9145732434,
 ➥   -8913795.369571615 3051428.330344116)
-[ RECORD 3 ]------------------------------------------------------------
name      | Embassy Drive
st_astext | LINESTRING(-8937543.523896862 3003115.226359022,
 ➥   -8937543.112014748 3003115.6723581622)
```

The query also uses the `ST_AsText` function to convert the coordinates of each `LineString` from binary (WKB) into a human-readable text format (WKT).

10.3.3 Exploring table with polygons

The last `planet_osm_polygon` table stores OSM data that can be represented as closed ways or polygons. Examples of such regions include cities, parks, lakes, and buildings.

If we execute the `\d florida.planet_osm_polygon` command and scroll to the end of the output by pressing Enter, we can see that the actual data type of the `way` column, which stores the location information, is `geometry(Geometry, 3857)`:

```
                      Table "florida.planet_osm_polygon"
       Column        |           Type           | Collation |
--------------------+-------------------------+-----------+
 osm_id             | bigint                  |           |
```

```
.other columns

 way                 | geometry(Geometry,3857) |           |
Indexes:
    "planet_osm_polygon_way_idx" gist (way) WITH (fillfactor='100')
```

Geometry is a generic subtype, which means the column can store a more specific subtype, such as Point, LineString, Polygon, MultiPolygon, or others.

In the case of OSM data from the planet_osm_polygon table, the coordinates and boundaries are usually defined using the Polygon and MultiPolygon types. The former represents a 2D closed planar region, and the latter is a collection of nonoverlapping and non-adjacent polygons. For example, let's see how sample OSM polygons can be defined by executing the following query, which returns the smallest named regions.

Listing 10.10 Getting the shortest named regions

```
SELECT name, ST_AsText(way)
FROM florida.planet_osm_polygon
WHERE name IS NOT NULL
ORDER BY ST_Area(way) ASC LIMIT 3;
```

The query uses the ST_Area function to calculate the area of each geometry, with units determined by the SRID of the column's data type. In our case, the SRID of the way column is 3857 (Web Mercator), so the area is calculated in square meters. After computing the area, the query orders the results and returns the three smallest ones:

```
-[ RECORD 1 ]---------------------------------------------------------
name      | Apalachicola River Wildlife and Environmental Area
st_astext | POLYGON((-9453576.600696124 3472118.429214709,
➥   -9453576.511640532 3472118.480504914,-9453576.600696124
➥   3472118.6087304275,-9453576.600696124 3472118.429214709))
-[ RECORD 2 ]---------------------------------------------------------
name      | Bay City Point
st_astext | POLYGON((-9534762.094574794 3527804.4425976463,-9534750.06093784
➥   3527804.4812334497,-9534738.505974695 3527804.519869253,
➥   -9534762.094574794 3527804.4425976463))
-[ RECORD 3 ]----------------------------------------------------------
name      | Rice Creek Conservation Area
st_astext | POLYGON((-9103060.734422492 3455483.5395689947,
➥   -9103060.734422492 3455483.731658956,-9103060.845741984
➥   3455483.731658956,-9103060.734422492 3455483.5395689947))
```

As we can see from the output, the Polygon type was used to define the coordinates and boundaries of the returned regions.

10.4 Visualizing geospatial data

PostGIS integrates with various systems and tools that allow querying and working with geospatial data through a graphical user interface. One such tool is QGIS, an open

source GIS that lets us load, analyze, manipulate, and visualize geographic data for a variety of purposes. Let's learn how to visualize our Florida dataset with QGIS by connecting the tool to the database and displaying data from the `planet_osm_polygon` and `planet_osm_point` tables.

QGIS is available for all major operating systems, including Windows, Linux, and macOS. Visit the tool's download page and choose the installation option that works best for you: https://qgis.org/download/.

Once QGIS is installed, start the tool, locate and right-click PostgreSQL in the list of supported data sources, and create a new PostGIS connection, as shown in figure 10.1.

In particular, the connection settings need to be as follows:

- *Name*—`postgres_docker` (or any other custom name)
- *Host*—`localhost`
- *Port*—`5432`
- *Database*—`postgres`
- *User name*—`postgres`
- *Password*—`password`

WARNING As figure 10.1 shows, we use the basic authentication method that stores the username and password as plain text in a project file. This is acceptable while exploring Postgres's geospatial capabilities, but once you start using QGIS professionally, you should use a more secure authentication method offered by QGIS: https://docs.qgis.org/latest/en/docs/user_manual/auth_system/auth_overview.html.

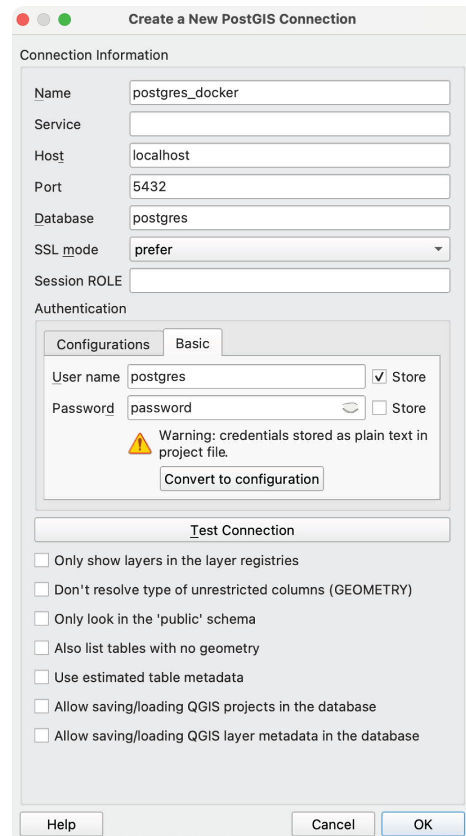


Figure 10.1 Configuring a PostGIS connection in QGIS

After connecting QGIS to your Postgres container, open the `florida` schema, select the `planet_osm_polygon` table, and select Add Layer to Project. Figure 10.2 shows how the data from the layer is rendered.

NOTE Depending on your system resources, QGIS might not display the tables under the `florida` schema or render the layers with their data immediately. If you see a spinning wheel, it means the tool is still processing your request.

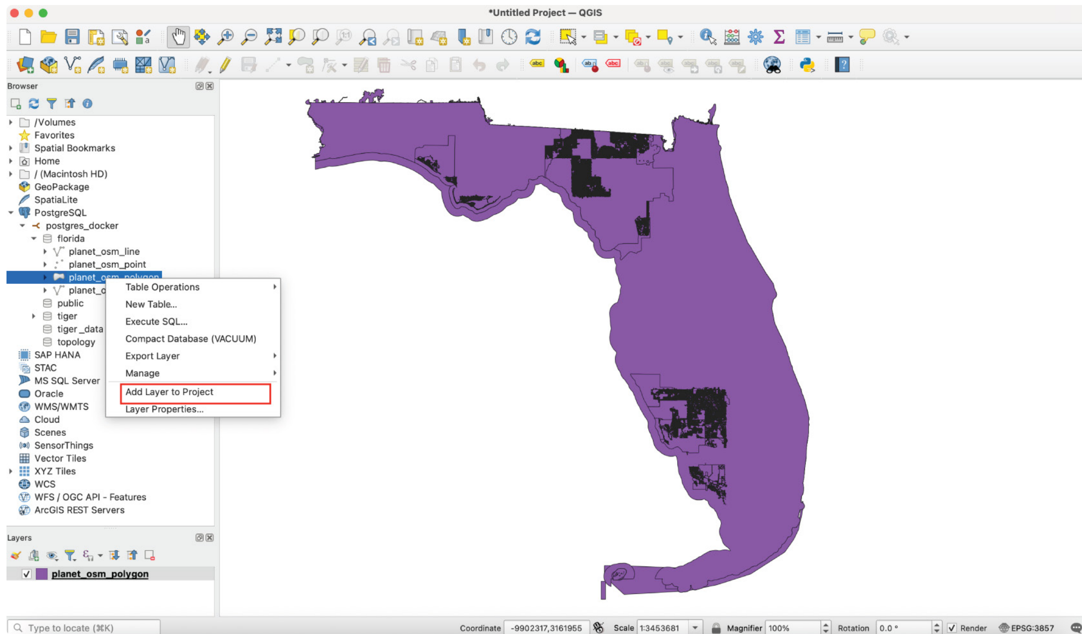


Figure 10.2 Visualizing polygons from the `planet_osm_polygon` table

The `planet_osm_polygon` table contains more than 6.8 million polygons. The final map in figure 10.2 doesn't display all of them, as larger areas can cover smaller ones. However, you'll see the map progressively rendered as QGIS loads the geometries.

GIS, like QGIS, allows you to work with multiple layers of data at once. Add the points from the `planet_osm_point` table by selecting it and then selecting Add Layer to Project to show the data as a new layer on top of the existing layer with the polygons. Figure 10.3 shows how the final map appears.

With that, we can see how our geospatial data from OSM actually looks, and we're ready to query it using the capabilities of the PostGIS extension. As for QGIS, an in-depth overview of the tool's capabilities is beyond the scope of this book. If you're interested in learning more, visit the official website at https://qgis.org/.

10.5 Querying geospatial data

The PostGIS extension provides dozens of functions and operators that can be used to query and manipulate geospatial data. In this section, we explore some of the essential functions that let us easily work with various geometries such as points, lines, and regions.

Let's get back to planning your imaginary family trip to Florida by visiting the areas highlighted in figure 10.4. Suppose the journey starts in Tampa (area 1 on the map), a cozy coastal city situated on the Gulf of Mexico. Then you'll visit the region with world-famous amusement parks such as Walt Disney World (area 2) and finish your journey in Miami (area 3).

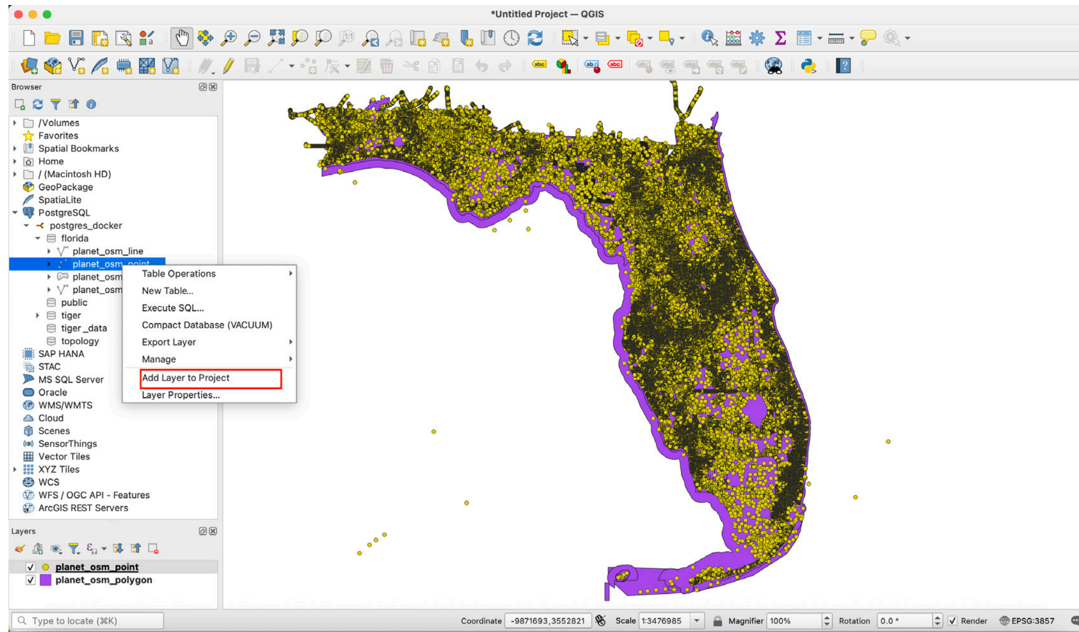


Figure 10.3 Visualizing points from the `planet_osm_point` table

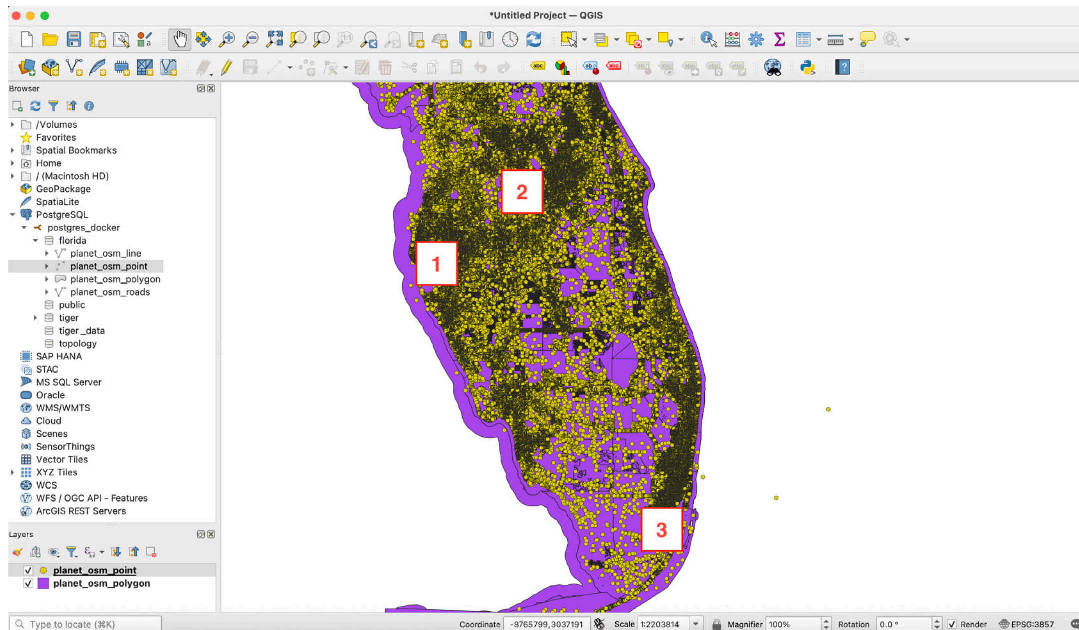


Figure 10.4 Areas in Florida to query on

10.5.1 Working with points

We'll start by exploring the first set of PostGIS functions while working with geometries of the `Point` type stored in the `florida.planet_osm_point` table. In particular, we'll learn to use the following capabilities:

- `ST_Transform(geometry geom, text to_srid)`—Transforms the coordinates of the provided `geom` geometry object to the `to_srid` spatial reference system. A new geometry object is created and returned as a result of the transformation.
- `ST_X(geometry a_point)` and `ST_Y(geometry a_point)`—Return the `X` and `Y` coordinates of the provided `a_point` geometry.
- `ST_DWithin(geometry g1, geometry g2, double precision distance_of_srid)`—Checks whether the `g1` and `g2` geometry objects are within the given distance from each other. The `distance_of_srid` is specified in units defined by the spatial reference system of the geometries.
- `ST_Distance(geometry g1, geometry g2)`—Calculates the minimum 2D Cartesian (planar) distance between two geometries using the units of their spatial reference system.
- `ST_MakePoint(float x, float y)`—Creates a `Point` from the provided `X` and `Y` coordinates.
- `ST_SetSRID(geometry geom, integer srid)`—Sets the SRID on the `geom` geometry to the given `srid` value. This function doesn't transform the geometry's coordinates but instead assigns metadata defining the spatial reference system the geometry is assumed to be in. This is especially useful when creating points with the `ST_MakePoint` function.

TRANSFORMING AND RETRIEVING POINT COORDINATES

Because the trip starts in Tampa, let's first find a point that represents the center of the city in our dataset. Such a point is usually aligned with the city's administrative center rather than being the geometric centroid of a polygon representing the city boundaries. The following query shows how to find this point by searching for Tampa among the places categorized as cities.

Listing 10.11 Searching for the center of Tampa

```
SELECT name, ST_AsText(way) AS coordinates
FROM florida.planet_osm_point
WHERE name = 'Tampa' and place = 'city';
```

The query finds the city center and returns the coordinates in a human-readable text format:

```
 name  |                  coordinates
-------+-------------------------------------------
 Tampa | POINT(-9179231.997685665 3242389.052874039)
(1 row)
```

As we already know, the returned coordinates are defined in the Web Mercator projection. But what if we'd like to convert these coordinates to degrees of longitude and latitude—the same format used by GPS?

This can be easily done by transforming the coordinates to SRID 4326, which represents locations on the Earth's surface using longitude and latitude. The next query shows how to perform this transformation using the `ST_Transform` function.

Listing 10.12 Transforming coordinates to latitude and longitude

```
SELECT name, ST_AsText(ST_Transform(way, 4326)) AS coordinates
FROM florida.planet_osm_point
WHERE name = 'Tampa' and place = 'city';
```

The result of the transformation looks as follows, where the longitude comes first and the latitude comes next in the returned `Point`:

```
 name  |              coordinates
-------+--------------------------------------
 Tampa | POINT(-82.458444 27.947759499952074)
(1 row)
```

We can use these transformed coordinates to view the location in a popular map service such as OSM or Google Maps. For example, the following query generates a URL for OSM that we can open in a browser.

Listing 10.13 Generating a URL for OSM

```
WITH tampa_city_point AS (
  SELECT ST_Transform(way, 4326) AS coordinate
  FROM florida.planet_osm_point
  WHERE name = 'Tampa' AND place = 'city'
)
SELECT 'https://www.openstreetmap.org/?mlat=' ||
  ST_Y(coordinate) || '&mlon=' || ST_X(coordinate) AS osm_url
FROM tampa_city_point;
```

The query executes as follows:

- The common table expression (CTE) named `tampa_city_point` locates the point and transforms it into degrees of longitude and latitude.
- The main query uses the transformed coordinate to generate a URL that, once opened, will place a marker on the map representing the center of Tampa. We use the `ST_X` function to extract the longitude and `ST_Y` to extract the latitude from the coordinate.

This is what the generated URL looks like:

```
                              osm_url
----------------------------------------------------------------------
https://www.openstreetmap.org/?mlat=27.947759499952074&mlon=-82.458444
(1 row)
```

If we open the URL in a browser, OSM will display a map similar to the one in figure 10.5. The marker on the map corresponds to the city center and uses the coordinates passed via the `mlat` and `mlon` parameters in the URL.

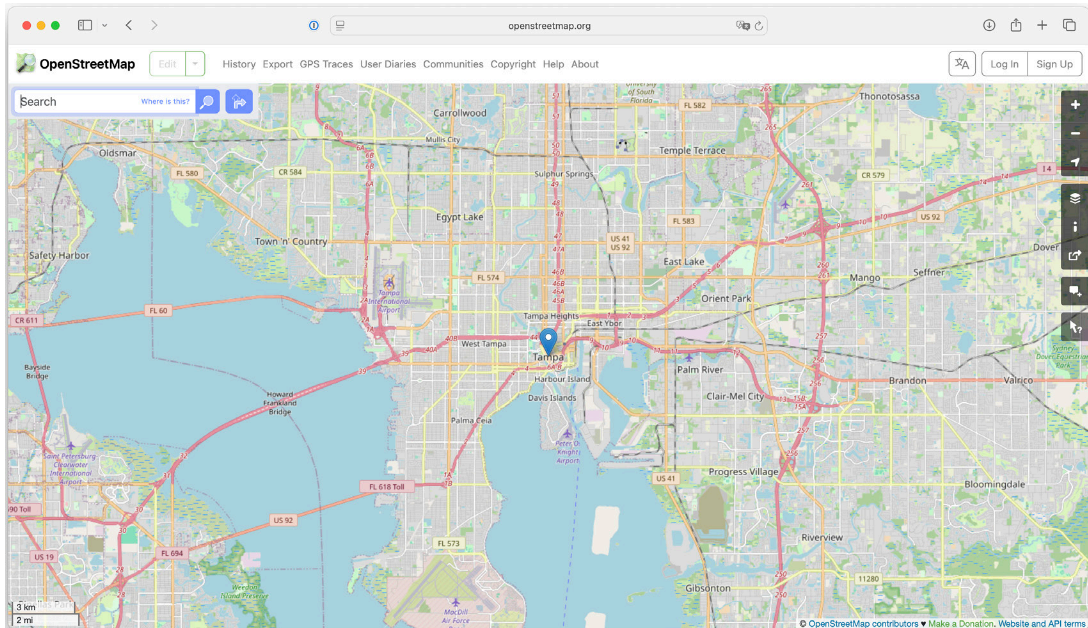


Figure 10.5 Map view with the marker showing the center of Tampa. (Photo courtesy of OpenStreetMap, https://www.openstreetmap.org/copyright.)

Working with the distance between points

Next, suppose that you'd like to stay in the Water Street District, a vibrant waterfront neighborhood in downtown Tampa offering a mix of public spaces, top-notch restaurants, and walkable access to cultural venues and parks. You've booked a unit at the Roost Apartment Hotel, located in the heart of the district, and now want to find restaurants within walking distance of the hotel. The following query shows how to find the closest restaurants using the `ST_DWithin` and `ST_Distance` functions of PostGIS.

Listing 10.14 Finding the closest restaurants

```
WITH roost_hotel AS (
  SELECT way FROM florida.planet_osm_point
```

```
  WHERE name = 'Roost Apartment Hotel' AND tourism='hotel'
)
SELECT p.name, p.amenity,
  round(ST_Distance(h.way,p.way)) as distance_meters,
  round(ST_Distance(h.way,p.way) * 3.28) as distance_feet
FROM roost_hotel as h
JOIN florida.planet_osm_point AS p
  ON ST_DWithin(h.way, p.way, 500)
WHERE p.amenity = 'restaurant'
ORDER BY distance_meters LIMIT 5;
```

The query execution flow is as follows:

1. The database first evaluates the `roost_hotel` CTE, which locates the coordinate of the Roost Apartment Hotel.
2. The CTE is joined with the `florida.planet_osm_point` table using the `ST_DWithin(h.way, p.way, 500)` condition, which returns `true` if the point `p.way` is within 500 meters of the hotel's location (`h.way`). The `WHERE` clause further filters the results to include only restaurants.
3. For each restaurant that matches the criteria, the database calculates the distance from the hotel using `ST_Distance`, returning it in both meters and feet. The `round` function rounds the calculated distance to the nearest integer value.
4. The query orders the results by distance, placing the closest restaurants first, and returns the top five.

Both `ST_DWithin` and `ST_Distance` operate in meters as long as we use the Web Mercator projection (SRID 3857) for our dataset.

Once the query is executed, you'll get the following selection of restaurants located in the Water Street District:

```
          name           |  amenity   | distance_meters | distance_feet
-------------------------+------------+-----------------+--------------
 Wagamama                | restaurant |              68 |           224
 Boulon Brasserie        | restaurant |             132 |           435
 The Pearl               | restaurant |             173 |           567
 3 Corners               | restaurant |             252 |           825
 Fabrica Woodfired Pizza | restaurant |             293 |           962
(5 rows)
```

NOTE We use the `ST_DWithin` function to find restaurants within 500 meters and not `ST_Distance` because the former can take advantage of spatial indexes, whereas the latter cannot. We'll discuss indexing capabilities in more detail in the following section.

CREATING POINTS FROM COORDINATES

Finally, assume that we already know the coordinates of the Roost Apartment Hotel and don't need to execute the `SELECT` statement in the `roost_hotel` CTE to retrieve them.

In this case, we can directly provide the coordinates to the query using `ST_MakePoint` and `ST_SetSRID` functions.

Let's first run this query to get the hotel's coordinates:

```
SELECT ST_AsText(way) FROM florida.planet_osm_point
WHERE name = 'Roost Apartment Hotel' AND tourism='hotel';
This coordinate looks as follows:
                  st_astext
----------------------------------------------
 POINT(-9178356.224987695 3242002.0503882724)
(1 row)
```

Next, let's use these coordinates directly in the following query, which—similar to the query in listing 10.14—finds the restaurants nearest to the hotel.

Listing 10.15 Finding the nearest restaurants to provided coordinates

```
WITH roost_hotel AS (
  SELECT ST_SetSRID(
    ST_MakePoint(-9178356.224987695, 3242002.0503882724), 3857) as point
)
SELECT p.name, p.amenity,
  round(ST_Distance(h.point,p.way)) as distance_meters,
  round(ST_Distance(h.point,p.way) * 3.28) as distance_feet
FROM roost_hotel as h
JOIN florida.planet_osm_point AS p
  ON ST_DWithin(h.point, p.way, 500)
WHERE p.amenity = 'restaurant'
ORDER BY distance_meters LIMIT 5;
```

The query uses the `ST_MakePoint` function to create a `Point` object from the hotel's coordinates and then applies the `ST_SetSRID` function to specify that the point uses the 3857 spatial reference system. The rest of the query is similar to the one in listing 10.14 and returns the same result:

```
          name           |  amenity   | distance_meters | distance_feet
-------------------------+------------+-----------------+--------------
 Wagamama                | restaurant |              68 |           224
 Boulon Brasserie        | restaurant |             132 |           435
 The Pearl               | restaurant |             173 |           567
 3 Corners               | restaurant |             252 |           825
 Fabrica Woodfired Pizza | restaurant |             293 |           962
(5 rows)
```

TIP PostGIS also allows us to create geometries by providing their coordinates and SRID in WKT format. For example, the `Point` representing the Roost Apartment Hotel from listing 10.15 can also be created using the `ST_GeomFromEWKT('SRID=3857;POINT(-9178356.224987695 3242002.0503882724)')` function call.

After selecting the hotel, suppose you're staying a few days in Tampa—exploring the vibrant neighborhoods of Water Street, Hyde Park, and Ybor City, riding the coasters at Busch Gardens, and cooling off at Adventure Island. You're also planning a quick drive to St. Petersburg, Tampa's twin city across the bay, connected by several bridges. There, you'll unwind on the pristine Gulf beaches, stroll the scenic piers, and visit the striking Salvador Dalí Museum. What comes after Tampa? Let's see next while learning how to work with polygons in PostGIS.

10.5.2 Working with polygons

After staying in Tampa, you're planning to immerse yourself in a world of magic by spending several days at Walt Disney World, Universal Studios, and the Legoland theme park, which are located less than a 1.5-hour drive from your hotel in Tampa. Assume that Walt Disney World is going to be the first theme park you visit during the trip, and now you'd like to learn more about it using data from our OSM dataset. In particular, we'll focus on the `planet_osm_polygon` table, which stores polygons representing the boundaries of objects such as cities, theme parks, buildings, and more.

We'll also practice using four additional PostGIS functions in this section:

- `ST_NPoints(geometry g1)`—Returns the number of points in the `g1` geometry.
- `ST_Area(geometry g1)`—Calculates the area of a polygonal geometry. The area is computed in the units defined by the geometry's SRID.
- `ST_Within(geometry A, geometry B)`—Checks whether geometry `A` is completely within geometry `B`. That is, all points of `A` must lie inside `B`.
- `ST_Equals(geometry A, geometry B)`—Checks whether the geometries `A` and `B` are topologically equal, meaning they have the same dimension and their point sets occupy the same space.

Let's start by searching for the polygon that defines the boundaries of the Walt Disney World theme park in the `florida.planet_osm_polygon` table. The following query shows how to find this polygon.

Listing 10.16 Finding a polygon and the number of points it's made of

```
SELECT name, tourism, ST_NPoints(way)
FROM florida.planet_osm_polygon
WHERE name = 'Walt Disney World';
```

The query uses the `ST_NPoints` function to calculate the number of points the polygon is made of and outputs the following result:

```
       name        |  tourism   | st_npoints
-------------------+------------+------------
 Walt Disney World | theme_park |        473
(1 row)
```

With more than 470 points defining the boundary of the Walt Disney World park, we can assume it spans a fairly large area. This can be verified using the `ST_Area` function.

Listing 10.17 Calculating the area of Walt Disney World

```
SELECT name, tourism,
  ST_Area(way) / 1000000.0 AS area_sq_km,
  ST_Area(way) / 4046.86 AS area_acres
FROM florida.planet_osm_polygon
WHERE name = 'Walt Disney World';
```

The `ST_Area` function calculates the area using the units defined by the spatial reference system. In our case, the data is stored using SRID 3857 (Web Mercator), so the area is computed in square meters. The result of the `ST_Area` calculation is then converted to square kilometers (`area_sq_km`) and acres (`area_acres`), making it easier to understand the park's size depending on the units you're most comfortable with:

```
       name        |  tourism    |     area_sq_km     |     area_acres
-------------------+-------------+--------------------+-------------------
 Walt Disney World | theme_park | 96.08026819284302 | 23741.930334343917
(1 row)
```

As we can see, with more than 96 square kilometers (23,741 acres), Walt Disney World is indeed a large theme park with a lot to offer. In fact, it's not just a single park, but a whole Disney universe made up of several distinct parks, each appealing to visitors of different ages and interests. And we can find the names of those distinct parks easily by executing the following query.

Listing 10.18 Finding distinct parks within Walt Disney World

```
WITH disney_world AS (
  SELECT way AS boundaries
  FROM florida.planet_osm_polygon
  WHERE name = 'Walt Disney World'
)
SELECT p.name, p.tourism
FROM florida.planet_osm_polygon AS p
JOIN disney_world AS d ON ST_Within(p.way, d.boundaries)
WHERE p.name IS NOT NULL
  AND p.tourism = 'theme_park'
  AND NOT ST_Equals(p.way, d.boundaries)
ORDER BY p.name;
```

The query execution flow looks as follows:

- The `disney_world` CTE finds the coordinates of the Walt Disney World park and assigns them to the `boundaries` column of the CTE result.
- This CTE is joined with the `florida.planet_osm_polygon` table using `ST_Within(p.way, d.boundaries)` as the join condition, where `p` is the alias for

florida.planet_osm_polygon and d is the alias for disney_world. The ST_Within function checks whether a polygon from the p table (p.way) is located within the boundaries of Walt Disney World (d.boundaries). All polygons that satisfy this condition are considered for the final result.

- The result is further filtered by the WHERE clause to ensure that the found polygons have a name (p.name IS NOT NULL) and are classified as theme parks (p.tourism = 'theme_park').
- Because Walt Disney World is fully contained within itself, it also satisfies the ST_Within(p.way, d.boundaries) condition. To exclude it from the final result, the NOT ST_Equals(p.way, d.boundaries) check is used to filter out the polygon with the exact same geometry as the one from the disney_world CTE.

Once we execute the query, we'll see that Walt Disney World is, in fact, made up of six theme parks, each with a distinct focus on particular themes, experiences, or audiences:

```
            name            |  tourism
----------------------------+-----------
 Disney's Animal Kingdom    | theme_park
 Disney's Blizzard Beach    | theme_park
 Disney's Hollywood Studios | theme_park
 Disney's Typhoon Lagoon    | theme_park
 EPCOT                      | theme_park
 Magic Kingdom              | theme_park
(6 rows)
```

Because each distinct park is a polygon in itself, we can continue using the ST_Within function to explore the attractions or other amenities each park has to offer. For example, the next query finds all attractions located within Disney's Hollywood Studios.

Listing 10.19 Finding attractions within Disney's Hollywood Studios

```
WITH disney_world AS (
  SELECT way AS boundaries
  FROM florida.planet_osm_polygon
  WHERE name = 'Disney''s Hollywood Studios'
)
SELECT p.name, p.tourism
FROM florida.planet_osm_polygon AS p
JOIN disney_world AS d ON ST_Within(p.way, d.boundaries)
WHERE p.name is NOT NULL AND p.tourism = 'attraction';
```

The query finds all named polygons (p.name IS NOT NULL) that are located within Disney's Hollywood Studios (ST_Within(p.way, d.boundaries)) and returns only those classified as attractions (p.tourism = 'attraction'). This query doesn't require the NOT ST_Equals(p.way, d.boundaries) condition because Disney's Hollywood Studios

is classified as a theme park, not an attraction, so its polygon will be excluded by the `p.tourism = 'attraction'` filter.

The result of the query looks as follows, suggesting that this particular theme park appeals more to audiences interested in experiences based on popular movies, including *Star Wars, Indiana Jones,* and *Toy Story*:

```
                    name                    |  tourism
--------------------------------------------+-----------
 Millennium Falcon: Smugglers Run           | attraction
 Star Wars: Rise of the Resistance          | attraction
 Star Tours: The Adventures Continue        | attraction
 Jim Henson's Muppet*Vision 3-D             | attraction
 Indiana Jones Epic Stunt Spectacular!      | attraction
 Mickey Shorts Theater                      | attraction
 Mickey and Minnie's Runaway Railway        | attraction
 Disney Junior Dance Party!                 | attraction
 Star Wars: Launch Bay                      | attraction
 Voyage of the Little Mermaid               | attraction
 Walt Disney Presents                       | attraction
 Toy Story Mania!                           | attraction
 Alien Swirling Saucers                     | attraction
 Slinky Dog Dash                            | attraction
 Rock 'n' Roller Coaster Starring Aerosmith | attraction
 The Twilight Zone Tower of Terror          | attraction
 Theater of the Stars                       | attraction
(17 rows)
```

So far, we've seen how the `ST_Within` function is used to check whether one polygon is located within another polygon. However, this function can also be applied to other geometry types, such as `Point` and `LineString`. For example, the following query demonstrates how to find all amenities within Disney's Hollywood Studios by using data from the `florida.planet_osm_point` table, where each amenity is represented as a `Point`.

Listing 10.20 Finding all amenities within Disney's Hollywood Studios

```
WITH disney_world AS (
  SELECT way AS boundaries
  FROM florida.planet_osm_polygon
  WHERE name = 'Disney''s Hollywood Studios'
)
SELECT p.name, p.amenity
FROM florida.planet_osm_point AS p
JOIN disney_world AS d ON ST_Within(p.way, d.boundaries)
WHERE p.name IS NOT NULL AND p.amenity IS NOT NULL;
```

The query uses the `ST_Within(p.way, d.boundaries)` condition to find all points (`p.way`) located within the polygon of the theme park (`d.boundaries`). The final result includes only those points that are classified as amenities (`p.amenity IS NOT NULL`) and have a name (`p.name IS NOT NULL`):

```
                name                |    amenity
------------------------------------+---------------
 Oga's Cantina                      | bar
 Docking Bay 7 Food and Cargo       | fast_food
 BaseLine Tap House                 | pub
 Chase                              | atm
 Jedi Training: Trials of the Temple | theatre
 Pizza Rizzo                        | fast_food
 Mama Melrose's Ristorante Italiano | restaurant
 First Aid                          | clinic
...truncated output
 Fantasmic!                         | theatre
(25 rows)
```

With that, suppose you've continued planning your time in the theme parks before heading south to the final stop of the journey—Miami. Let's see what awaits you in Miami while exploring the remaining PostGIS capabilities discussed in the chapter.

10.5.3 Working with line segments

Once you get to Miami, you're planning to stay at one of the hotels in Miami Beach, which is within walking distance of the pristine beaches along the Atlantic Ocean. The area is also close to Downtown Miami, Brickell, and the Design District, where your family can enjoy a taste of city life by visiting museums, shopping at boutique stores, and dining at top-rated restaurants.

Imagine that as you're planning the final days of your journey in Miami, you get curious about how to work with roads and other types of linear objects from the OSM dataset located within the city or crossing its boundaries. In this section, we'll learn how to work with line segments from the `florida.planet_osm_line` table, which stores highways, roads, rivers, and other objects represented as `LineString`. We'll also explore the following functions provided by the PostGIS extension:

- `ST_Length(geometry g1)`—Calculates the 2D Cartesian length of the geometry, such as a `LineString`
- `ST_Intersects(geometry A, geometry B)`—Returns `true` if the two geometries share at least one point in common

First, let's see what types of roads are stored in the `florida.planet_osm_line` table by executing the following query.

Listing 10.21 Finding the top 10 types of roads

```
SELECT highway, count(highway) AS total_count
FROM florida.planet_osm_line
WHERE highway IS NOT NULL
GROUP BY highway ORDER BY total_count DESC LIMIT 10;
```

This query retrieves all line segments classified as roads (`highway IS NOT NULL`) and uses `GROUP BY` along with the `count(highway)` aggregate function to calculate the total

number of roads for each type. The final result includes only the top 10 road types, sorted by their total count in descending order:

```
   highway     | total_count
--------------+-------------
 service       |     1121448
 residential   |      569409
 footway       |      513955
 track         |       62237
 tertiary      |       61515
 primary       |       57634
 secondary     |       42877
 path          |       25646
 unclassified  |       23243
 trunk         |       15869
(10 rows)
```

Next, let's execute the following query to see how the `ST_Intersects` function can be used to find the top 10 primary and secondary roads that either lie entirely within or cross the boundaries of Miami.

Listing 10.22 Finding the top 10 roads within or crossing Miami

```
WITH miami AS (
  SELECT way AS boundaries
  FROM florida.planet_osm_polygon
  WHERE name = 'Miami' AND place = 'city'
)
SELECT l.name, l.highway,
  ST_NPoints(l.way) AS points_number,
  round(ST_Length(l.way)) AS len_meters,
  round(ST_Length(l.way) * 3.28084) AS len_feet
FROM florida.planet_osm_line l
JOIN miami m ON ST_Intersects(l.way, m.boundaries)
WHERE l.highway IN ('primary', 'secondary')
ORDER BY len_meters DESC LIMIT 10;
```

The query performs the following steps:

- The `miami` CTE locates the polygon that defines the boundaries of Miami in the `florida.planet_osm_polygon` table.
- The result of the CTE is joined with the `florida.planet_osm_line` table using the `ST_Intersects(l.way, m.boundaries)` condition, which finds all ways (`l.way`) that either cross or are fully contained within the city boundaries (`m.boundaries`).
- The `WHERE` clause filters the result to include only roads classified as primary or secondary (`l.highway IN ('primary', 'secondary')`).
- For each matching road, the query calculates its total length in meters and feet using the `ST_Length` function and counts the number of points that make up the road segment with the `ST_NPoints` function.

- The final result is ordered by road length in descending order, and the query returns the top 10 longest roads.

The query produces the following result:

```
         name          |  highway   | points_number | len_meters | len_feet
-----------------------+-----------+---------------+------------+---------
 Southwest 22nd Avenue | secondary |            31 |       1422 |     4666
 Northwest 62nd Street | secondary |            13 |       1326 |     4350
 Rickenbacker Causeway | primary   |             2 |       1221 |     4006
 Rickenbacker Causeway | primary   |             2 |       1221 |     4006
 Northwest 32nd Avenue | secondary |            30 |       1142 |     3748
 South Miami Avenue    | secondary |            27 |       1046 |     3433
 Southwest 37th Avenue | secondary |            22 |       1003 |     3289
 Brickell Avenue       | primary   |            23 |       1002 |     3287
 Northwest 62nd Street | secondary |            10 |        991 |     3250
 Southwest 37th Avenue | secondary |            35 |        981 |     3219
(10 rows)
```

As the `points_number` column shows, roads are represented as line segments defined by multiple points. Additionally, some road names appear more than once in the output because those roads are split into several segments, each stored separately in the `florida.planet_osm_line` table.

> **TIP** PostGIS can be complemented by the `pgRouting` extension when you need to find routes between two locations, calculate driving distances, or perform other tasks typical of a navigation application. Although this goes beyond the scope of the book, you can learn more by visiting the `pgRouting` website at https://pgrouting.org.

With that, we've learned some of the essential data types and functions of the PostGIS extension for working with geospatial data. Next, let's see how to take advantage of Postgres's indexing capabilities to optimize the search over geodata.

10.6 Indexing geospatial data

As we've seen, geospatial data can be represented by various geometries such as points, polygons, and line segments. The location and boundaries of these geometric objects are defined using 2D or multidimensional coordinates within a specific spatial reference system. Because geometries operate in coordinate systems with two or more dimensions and can take on many shapes and forms, we need specialized spatial indexes to query geodata efficiently.

Geospatial indexes don't store the actual coordinates or shapes of the geometric objects. Instead, they index the bounding boxes that those geometries belong to. In a 2D space, a bounding box is a rectangle that defines the spatial extent and position of an area. By indexing bounding boxes, geospatial indexes can optimize searches by quickly identifying geometries that lie within or intersect a given area—while skipping over those that don't.

The PostGIS extension supports the following three Postgres index types for indexing and querying geospatial data:

- *GiST (generalized search tree)*—This index type divides the spatial space into bounding boxes that cover all the geometries from the dataset. Larger bounding boxes are recursively split into smaller ones, allowing the database to traverse the index structure efficiently by reaching the smallest bounding boxes that store references to the actual geometries being indexed, such as points or line segments. GiST is considered a default choice for geospatial data, as it supports a wide range of spatial operators and functions and performs well across datasets of various sizes and complexity.
- *SP-GiST (space-partitioned generalized search tree)*—This index type divides the spatial space into four non-overlapping areas called *quadrants* around a selected point known as the *centroid*. Each quadrant is a bounding box. The top-level quadrants are further split into smaller ones, each built around its own centroid. The search starts at the top level and proceeds down to the smaller quadrants that store references to the actual geometries being indexed. SP-GiST is best suited for geospatial data with little or no overlap and that is evenly distributed across the space. For instance, it performs well for point data, which doesn't overlap by nature, especially when the points are spread uniformly. However, SP-GiST supports fewer spatial operators and functions compared to GiST.
- *BRIN (block range index)*—This index type stores bounding boxes that cover all geometries from a specific range of table pages. Each range spans multiple pages containing rows with actual geospatial data, and the goal is to define a bounding box that covers all geometries within that range. During the search, the index identifies all ranges whose bounding boxes match the search criteria and then scans the corresponding pages to find matching rows. BRIN's biggest advantage is its size: because it indexes ranges, it's much more compact than GiST or SP-GiST for large datasets. However, for BRIN to perform well, the data in the table needs to be stored in some order. Chapter 9 explores how BRIN works for time-series data, as the same concepts apply to geospatial data as well.

Next, let's explore in detail how PostGIS uses GiST, the general-purpose index for geospatial data.

TIP Refer to the official PostGIS documentation if you'd like to learn how SP-GiST and BRIN indexes can be used with geospatial data: https://postgis.net/docs/using_postgis_dbmanagement.html#build-indexes.

10.6.1 Understanding GiST structure

GiST is a foundational index type in Postgres that enables support for specialized data types and query patterns. In a nutshell, GiST integrates into Postgres by

implementing low-level indexing capabilities and exposing an interface (a set of functions) that needs to be implemented to support a custom data type or access method.

In chapter 6, we already saw how GiST helps to optimize full-text search by indexing lexemes of the `tsvector` type. Beyond that, GiST is used to index network addresses of the `inet` type, ranges of values such as `int4range` and `daterange`, trigrams from the `pg_trgm` extension, and many other types that require specialized handling and data access methods.

PostGIS uses GiST to provide indexing support for the `geometry` and `geography` data types, implementing the access methods for geospatial functions and operators such as containment and intersection. The extension implements an R-tree on top of GiST. The *R* stands for *rectangular*, referring to how the index divides space into bounding rectangles (boxes) that encompass geospatial data like points and line segments. Query performance is improved by narrowing the search to only those data elements that are contained in, intersect with, or fall outside specific bounding rectangles.

Let's take a closer look at a sample GiST structure when the index is used for points from our `florida.planet_osm_point table`. First, as shown in figure 10.6, the points can

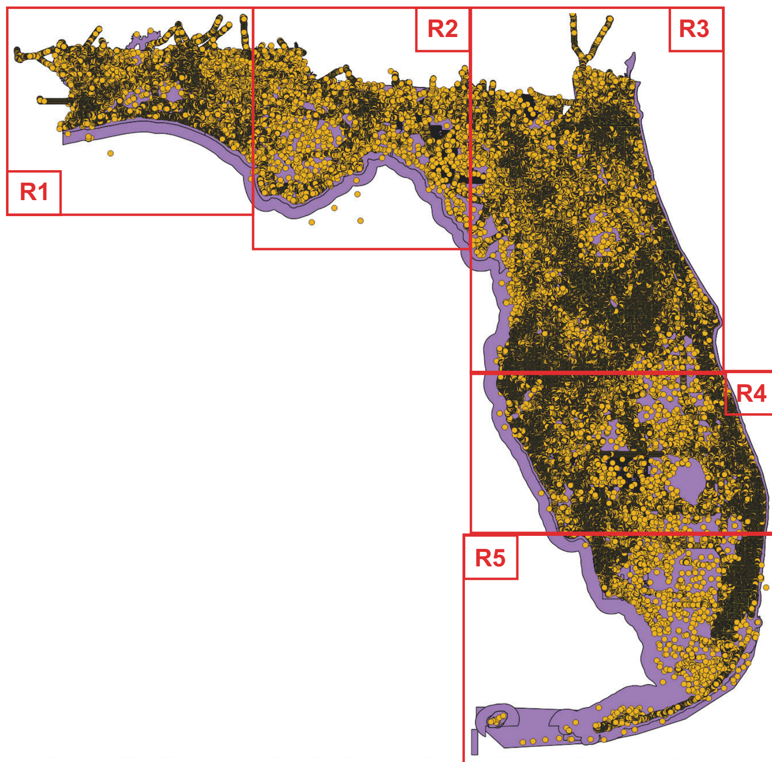


Figure 10.6 The top-level bounding rectangles of the GiST index covering points from the dataset for Florida

be divided into several top-level bounding boxes that cover the entire state of Florida. In our example, there are five top-level rectangles labeled R1 through R5. These rectangles can intersect if needed.

Next, the top-level rectangles are further divided into smaller ones, as shown in figure 10.7. At the next level of the index, there are 19 additional bounding rectangles labeled R6 through R25.

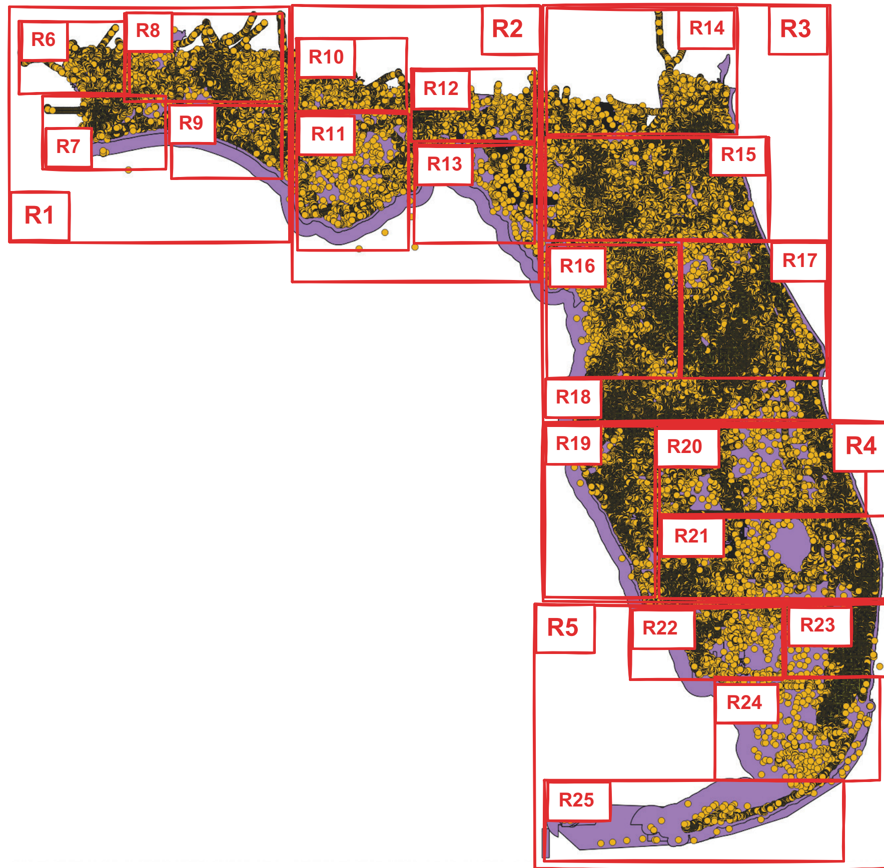


Figure 10.7 The next-level bounding rectangles of the GiST index covering points from the dataset for Florida

These rectangles can be further subdivided if needed to achieve better performance by distributing the points more evenly across bounding boxes. However, let's assume that in our case, the index construction stopped after GiST defined the second-level boxes shown in figure 10.7. As a result, the final index structure looks like figure 10.8.

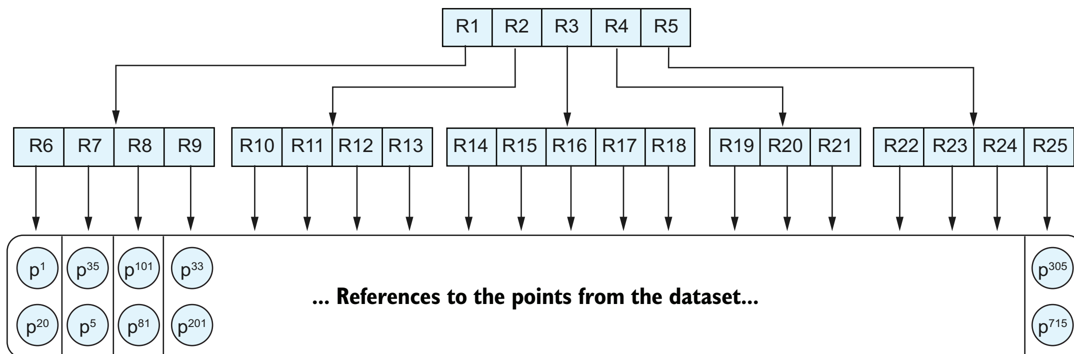


Figure 10.8 Sample GiST index structure with two levels of bounding rectangles

In particular, the index is organized as follows:

- The root-level index entries store the boundaries of the top-level rectangles labeled R1 through R5.
- The second level of the index stores smaller rectangles located within one of the larger rectangles from the upper level. For example, rectangles R6 through R9 are located within the boundaries of rectangle R1 and are therefore referenced by it.
- Rectangles from the second level are considered leaf index entries because they don't reference any smaller rectangles. As leaf entries, they point to the actual table data being indexed—in our case, points. For instance, rectangle R6 references table rows p1 and p20, which store points that fall within the boundaries of R6.

Finally, let's see how GiST uses the created index structure to optimize search. Imagine that you'd like to find some cultural places near Downtown Miami. In this case, Postgres can use the GiST index to narrow the search to a specific bounding rectangle and, as a result, a subset of points. Figure 10.9 visualizes how the search takes place using our sample index structure.

The search begins with the root-level index entries, which store coordinates for five large bounding rectangles.

1. While checking those root index entries, Postgres determines that Downtown Miami is located within rectangle R5, which covers the southern part of Florida.
2. The search continues within that area by visiting index entries for rectangles R22 through R25, which are referenced by rectangle R5.
3. The database identifies that Downtown Miami falls within the boundaries of rectangle R24.
4. Postgres scans all the points from rectangle R24 and returns only those matching the search criteria, such as cultural places in close proximity to Downtown Miami. In our example, those points happen to be p423 and p345.

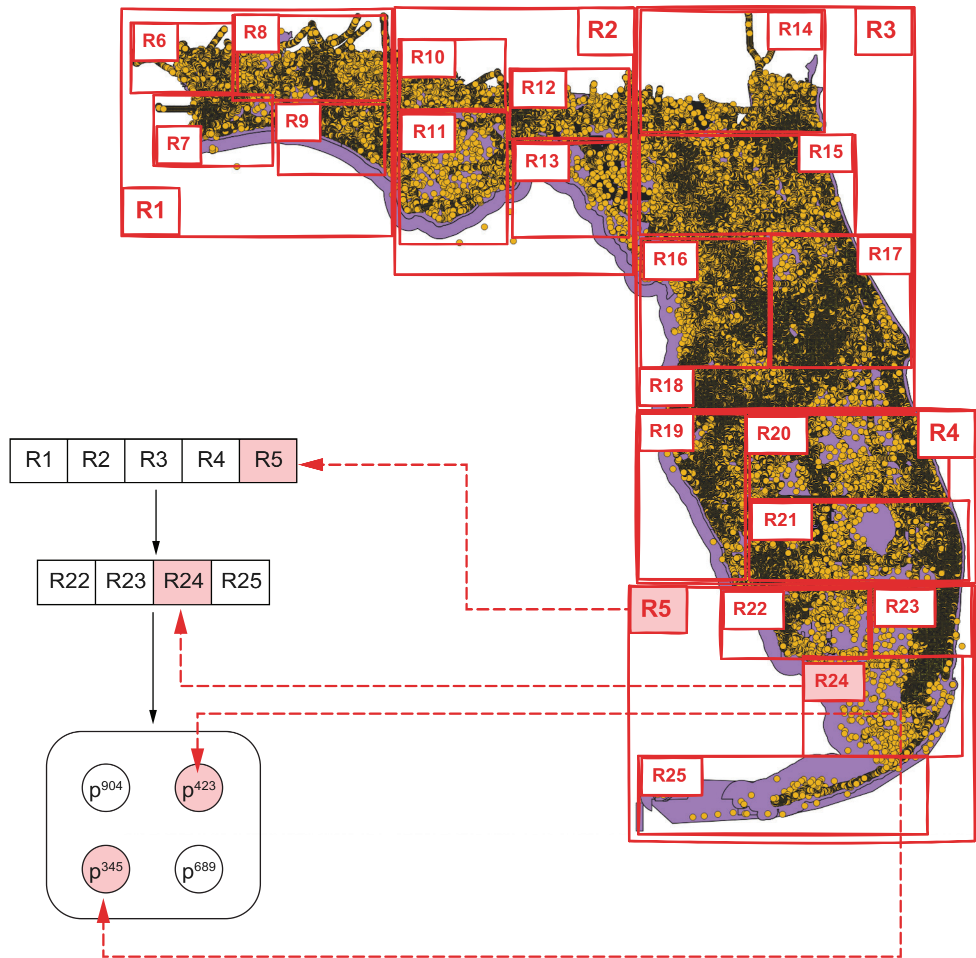


Figure 10.9 Using the GiST index to find cultural places near Downtown Miami

10.6.2 Using a GiST index

Now that we understand how GiST optimizes searches for geospatial data, let's see it in action. Imagine that during your stay in Miami, your family wants to do some shopping in the Brickell neighborhood, and you'd like to put together a list of shops and boutiques within walking distance of the Brickell City Centre station. To find such a list, execute the following query.

Listing 10.23 Finding the 10 closest shops near Brickell City Centre

```
WITH brickell AS (
  SELECT way FROM florida.planet_osm_point
```

```
  WHERE name = 'Brickell City Centre' and railway = 'station'
)
SELECT p.name, p.shop,
  round(ST_Distance(b.way,p.way)) as distance_meters,
  round(ST_Distance(b.way,p.way) * 3.28) as distance_feet
FROM brickell as b
JOIN florida.planet_osm_point AS p
  ON ST_DWithin(b.way, p.way, 500)
WHERE p.shop IS NOT NULL
ORDER BY distance_meters LIMIT 10;
```

The query uses the `brickell` CTE to find the coordinates (`way`) for Brickell City Centre and then takes advantage of the `ST_DWithin` function to return only those places within 500 meters of the station. The final result is further reduced to the 10 nearest places classified as shops (`p.shop IS NOT NULL`):

```
        name         |   shop    | distance_meters | distance_feet
---------------------+-----------+-----------------+---------------
 Intimissimi         | clothes   |              63 |           207
 Sur La Table        | houseware |              67 |           220
 Bath & Body Works   | cosmetics |              69 |           227
 Swarovski           | jewelry   |              69 |           228
 Acqua di Parma      | perfumery |              76 |           250
 Sunglass Hut        | optician  |              78 |           256
 Santa Maria Novella | perfumery |              79 |           260
 Vilebrequin         | clothes   |              79 |           259
 Agent Provocateur   | clothes   |              81 |           266
 APM Monaco          | jewelry   |              83 |           271
(10 rows)
```

Before we take a look at the execution plan for this query, let's run the following statement to view the existing indexes on the tables in our dataset.

Listing 10.24 Checking a list of existing indexes

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE schemaname = 'florida' AND tablename LIKE 'planet_osm_%';
```

The query reports that every table already has a GiST index on its `way` column:

```
        indexname          |                  indexdef
---------------------------+------------------------------------------
 planet_osm_line_way_idx   | CREATE INDEX planet_osm_line_way_idx ON
                           | florida.planet_osm_line USING gist (way)
                           | WITH (fillfactor='100')
 planet_osm_point_way_idx  | CREATE INDEX planet_osm_point_way_idx ON
                           | florida.planet_osm_point USING gist (way)
                           | WITH (fillfactor='100')
 planet_osm_polygon_way_idx | CREATE INDEX planet_osm_polygon_way_idx ON
                           | florida.planet_osm_polygon USING gist (way)
```

```
                             | WITH (fillfactor='100')
 planet_osm_roads_way_idx    | CREATE INDEX planet_osm_roads_way_idx ON
                             | florida.planet_osm_roads USING gist (way)
                             | WITH (fillfactor='100')
```

These indexes were created automatically by the `osm2pgsql` tool during the preloading of the OSM dataset for Florida. The `indexdef` column in the output shows the actual statement used to create each GiST index:

- The `USING gist (way)` clause defines a GiST index on the `way` column of the respective table.
- The `WITH (fillfactor='100')` parameter configures `fillfactor`, which defines the percentage that determines how full the index method will try to pack index pages. The parameter is set to its default value of 100, which will result in a denser tree with fewer levels.

TIP When you create a GiST index from scratch and don't need to adjust `fillfactor`, you can omit the `WITH (fillfactor='100')` clause from the `CREATE INDEX` statement.

With that, we already have GiST indexes in place for the OSM dataset, and they are actively used by Postgres. However, before the query from listing 10.23 can use the GiST index on the `way` column, it first needs to find the coordinates for the Brickell City Centre station using the `WHERE name = 'Brickell City Centre' AND railway = 'station'` clause.

Because no index is on either the `name` or `railway` columns (see the output for listing 10.24), this will lead to a full table scan of the `florida.planet_osm_point` table. To avoid that, let's create a B-tree index on the `name` column.

Listing 10.25 Creating a B-tree index on the `name` column

```
CREATE INDEX florida_point_name_idx
ON florida.planet_osm_point(name)
WHERE name IS NOT NULL;
```

The `florida.planet_osm_point` table has more than 2 million unnamed points with the `name` column set to `NULL`. For efficiency, we create a partial index using `WHERE name IS NOT NULL` condition that adds only named points to the index structure.

Now, let's run the following query to see the execution plan for the query returning the 10 closest shops near Brickell City Centre station.

Listing 10.26 Checking the execution plan using the GiST index

```
EXPLAIN (analyze, costs off)
WITH brickell AS (
  SELECT way FROM florida.planet_osm_point
  WHERE name = 'Brickell City Centre' and railway = 'station'
```

```
)
SELECT p.name, p.shop,
  round(ST_Distance(b.way,p.way)) as distance_meters,
  round(ST_Distance(b.way,p.way) * 3.28) as distance_feet
FROM brickell as b
JOIN florida.planet_osm_point AS p
  ON ST_DWithin(b.way, p.way, 500)
WHERE p.shop IS NOT NULL
ORDER BY distance_meters LIMIT 10;
```

Postgres selects the following execution plan:

```
                                QUERY PLAN
--------------------------------------------------------------------------
 Limit (actual time=0.991..1.005 rows=10 loops=1)
   ->  Result (actual time=0.987..0.998 rows=10 loops=1)
         ->  Sort (actual time=0.981..0.984 rows=10 loops=1)
               Sort Key: (round(st_distance(planet_osm_point.way, p.way)))
               Sort Method: top-N heapsort  Memory: 27kB
               ->  Nested Loop (actual time=0.456..0.948 rows=36 loops=1)
                     ->  Index Scan using florida_point_name_idx
➥   on planet_osm_point (actual time=0.074..0.078 rows=1 loops=1)
                           Index Cond: (name = 'Brickell City Centre'::text)
                           Filter: (railway = 'station'::text)
                           Rows Removed by Filter: 2
                     ->  Bitmap Heap Scan on planet_osm_point p
➥   (actual time=0.370..0.831 rows=36 loops=1)
                           Filter: ((shop IS NOT NULL) AND
➥   st_dwithin(planet_osm_point.way, way, '500'::double precision))
                           Rows Removed by Filter: 1169
                           Heap Blocks: exact=21
                           ->  Bitmap Index Scan on planet_osm_point_way_idx
➥    (actual time=0.308..0.308 rows=1205 loops=1)
                                 Index Cond: (way &&
➥   st_expand(planet_osm_point.way, '500'::double precision))
 Planning Time: 2.925 ms
 Execution Time: 1.125 ms
(18 rows)
```

These are the main execution steps of the plan:

1. The database performs a `Nested Loop` join, where each row from the outer loop is compared to each row from the inner loop.
2. The outer loop is evaluated first. It executes the query from the `brickell` CTE. The database uses an `Index Scan` on the B-tree index (`florida_point_name_idx`) to find all records that match the `name = 'Brickell City Centre'` condition. It finds three matching records. Two are filtered out (`Rows Removed by Filter: 2`) because they don't satisfy the `railway = 'station'` condition. The remaining row becomes the result of the outer loop (see `rows=1` produced by the `Index Scan using florida_point_name_idx on planet_osm_point` phase).

3. The database then proceeds with the inner loop using the single row from the outer loop, which contains the coordinates for the Brickell City Centre station. These coordinates are used in the `Bitmap Index Scan` phase, which traverses the GiST index (`planet_osm_point_way_idx`) to find all points whose `way` coordinates are within 500 meters of the station. In particular, this check is performed using the `&&` bounding box operator, which tests whether a point intersects/overlaps with the 500-meter bounding box around the station, as produced by the `st_expand(planet_osm_point.way, '500')` function. In the end, the `Bitmap Index Scan` produces a bitmap specifying table rows that match this index condition.
4. Postgres carries on with the `Bitmap Heap Scan` phase by visiting all the pages with rows listed in the bitmap. It visits 21 pages in total (`Heap Blocks: exact=21`) and filters out 1,169 rows that are either not classified as shops (`shop IS NOT NULL`) or not within 500 meters of the station's coordinates (`st_dwithin(planet_osm_point.way, way, '500')`). The `st_dwithin` function is more accurate than `st_expand` because it compares the actual distance between two points, whereas `st_expand` only checks whether a point overlaps the 500-meter bounding box around the station's coordinates.
5. Once the `Nested Loop` join completes, the database sorts the result and returns the 10 places nearest to the Brickell City Centre station.

The `ST_DWithin` function used in the `join` condition of the query is one of the index-aware functions that can use the GiST index by performing an initial fast filtering of the data using the combination of the bounding box operator `&&` and the `ST_Expand` function. Other examples include `ST_Contains`, `ST_Intersects`, `ST_Overlaps`, and so on, which also rely on bounding box operators such as `&&` to speed up search with GiST.

TIP Refer to the official PostGIS documentation to learn more about additional bounding box operators and index-aware functions that can expedite search via the GiST index: https://postgis.net/docs/using_postgis_query.html#using-query-indexes.

The `ST_Distance` function, though, is not index-aware because it always calculates the actual distance between geometries and doesn't use the bounding box operators that can speed up the search through the GiST index. To confirm that, let's execute the following query, which uses the `ST_Distance(b.way, p.way) <= 500` condition instead of `ST_DWithin(b.way, p.way, 500)` to find all the shops near the Brickell City Centre station.

Listing 10.27 Checking the execution plan with `ST_Distance`

```
EXPLAIN (analyze, costs off)
WITH brickell AS (
  SELECT way FROM florida.planet_osm_point
  WHERE name = 'Brickell City Centre' and railway = 'station'
)
SELECT p.name, p.shop,
  round(ST_Distance(b.way,p.way)) as distance_meters,
```

```
  round(ST_Distance(b.way,p.way) * 3.28) as distance_feet
FROM brickell as b
JOIN florida.planet_osm_point AS p
  ON ST_Distance(b.way, p.way) <= 500
WHERE p.shop IS NOT NULL
ORDER BY distance_meters LIMIT 10;
```

Now the execution plan looks as follows:

```
                                QUERY PLAN
--------------------------------------------------------------------------
 Limit (actual time=474.273..481.189 rows=10 loops=1)
   ->  Result (actual time=426.333..433.247 rows=10 loops=1)
         ->  Sort (actual time=426.214..433.125 rows=10 loops=1)
               Sort Key: (round(st_distance(planet_osm_point.way, p.way)))
               Sort Method: top-N heapsort  Memory: 27kB
               ->  Nested Loop
➥   (actual time=388.195..433.100 rows=36 loops=1)
                     Join Filter: (st_distance(planet_osm_point.way, p.way)
➥   <= '500'::double precision)
                     Rows Removed by Join Filter: 18640
                     ->  Index Scan using florida_point_name_idx
➥   on planet_osm_point (actual time=0.701..0.706 rows=1 loops=1)
                           Index Cond: (name = 'Brickell City Centre'::text)
                           Filter: (railway = 'station'::text)
                           Rows Removed by Filter: 2
                     ->  Gather
➥   (actual time=1.748..426.856 rows=18676 loops=1)
                           Workers Planned: 2
                           Workers Launched: 2
                           ->  Parallel Seq Scan on planet_osm_point p
➥   (actual time=103.089..300.202 rows=6225 loops=3)
                                 Filter: (shop IS NOT NULL)
                                 Rows Removed by Filter: 807749
 Planning Time: 0.556 ms
 Execution Time: 488.119 ms
(20 rows)
```

As we can see, Postgres no longer uses the existing GiST index (`planet_osm_point_way_idx`). Instead, it performs a parallel full table scan using several workers (`Parallel Seq Scan on planet_osm_point`).

With that, we've covered enough PostGIS capabilities to start building geospatial applications on Postgres and explored some places in the Sunshine State of the United States you might want to visit in the future.

Summary

- PostGIS is an extension that turns Postgres into a geospatial database.
- The extension introduces special data types and adds a wide range of functions for geospatial data querying and processing.

- The `geometry` data type uses a Euclidean plane to represent geospatial objects and relies on Cartesian mathematics for calculations, which makes them easier to implement and faster to execute.
- The `geography` data type takes the Earth's curvature into account by using a spherical model for calculations, making them more accurate than those performed with the `geometry` type but slower due to more complex underlying math.
- Dozens of geospatial functions let you query, create, transform, and process geospatial data in Postgres.
- PostGIS integrates with various systems and tools that allow querying and working with geospatial data through a GUI.
- Queries over geospatial data can be optimized using GiST, SP-GiST, and BRIN indexes. GiST is considered the default choice for geospatial data.
- Index-aware functions such as `ST_DWithin`, `ST_Contains`, and `ST_Intersects` can use the GiST index.

11 Postgres as a message queue

This chapter covers

- When, why, and how to use Postgres as a message queue
- Implementing a custom message queue using Postgres built-in capabilities
- Notifying subscribers of new messages in the queue
- Using the pgmq extension as a lightweight, ready-to-use messaging solution

A *message queue* is a component that enables services and other system components to communicate asynchronously. Instead of calling each other directly, one service (the producer) sends a message to the queue, and another service (the consumer) reads and processes the message later. The queue holds messages until they are processed and deleted. Common use cases for message queues include decoupling applications by using the queues as an asynchronous communication layer, broadcasting and exchanging events, and distributing jobs among worker processes.

Let's explore when and how to use Postgres as a message queue as we build a visitor registration and tracking system for a Department of Motor Vehicles (DMV). We'll learn how to create, manage, and work with queues in Postgres while processing new registrations and notifying visitors in line.

11.1 When to use Postgres as a message queue

At a minimum, every message queue needs to provide two basic capabilities. First, it must be able to receive messages from producers and retain them while they are needed by consumers. Some queues store messages only in memory, providing fast access to the data, whereas others persist them on disk to ensure that the data is not lost even if the queue restarts. Second, every queue must provide an API that can be used for publishing, reading, and managing messages efficiently. Most specialized messaging solutions introduce their own purpose-built APIs and libraries.

As a general-purpose database, Postgres supports these essential capabilities of message queues as follows:

- A queue can be implemented on top of a regular Postgres table, with each message stored as an individual table record. As with any other table data, Postgres persists messages on disk to ensure transactional consistency and durability, and caches them in memory depending on available resources.
- The structure of a message can be defined using the rich set of Postgres data types. For example, fields like `message_id` and `created_time` can use strict types such as `int8` and `timestamptz`, ensuring that producers don't compromise data integrity by always sending those values in a valid format. The actual message payload, however, can be stored in a column of more flexible `json` or `jsonb` data types, allowing producers to send custom payloads without adhering to a fixed schema.
- When it comes to APIs, at the lowest level, all messages are stored, queried, and manipulated using SQL. At a higher level, message queue APIs can be implemented as database functions (stored procedures) or provided as client-side libraries for specific programming languages. In both cases, the underlying implementation still relies on Postgres tables and uses SQL to produce and consume the messages.
- The database also offers capabilities that ensure that the performance characteristics of a Postgres-based queue don't degrade as the message volume grows or the number of producers and consumers increases.

However, even though Postgres has the necessary building blocks, it's not a purpose-built message queue. That's why it's important to understand when it's reasonable to use it as one and when a specialized messaging solution might be a better fit. Use the following three criteria to decide when and if you should use Postgres as a queue:

1. *You need Postgres transactional capabilities to execute a business operation and record a message that the operation produces atomically.*

Imagine visitors coming to a DMV and checking in at a registration terminal for a particular service, such as a driver's license application or vehicle registration. Once visitors are checked in, they're added to a waiting queue and will be called when a government representative is ready to serve them.

In this case, a successful check-in at the terminal produces a message (event) that a new visitor has been registered and needs to be notified when it's their turn. If you want the check-in operation and the message added to the visitors queue to be executed atomically (as a single transaction), then use Postgres. In this case, the database guarantees that either both operations succeed or neither does.

With specialized messaging solutions, your application would need to provide such guarantees itself by first performing the check-in (the business operation) in Postgres and then sending a message to a separate message queue. If the queue fails to record the event for any reason, your application has to deal with this quickly—because the visitor has already checked in and expects to be called.

2 *You need a queue whose message volume can be easily handled by Postgres.*

By default, Postgres functions as a single-server instance handling both read and write requests. If the database instance can easily process the volume of messages produced and consumed by your application alongside other business operations, then use Postgres as a message queue.

If there are too many message consumers saturating the database with read requests, Postgres allows you to add replica nodes to offload read traffic from the primary instance. If there are too many producers, the database may become saturated with write requests. To scale writes, you'll need to scale up the primary Postgres instance by upgrading to a server with more resources or migrating to a sharded or distributed version of Postgres, such as CitusData or YugabyteDB.

In any case, benchmark and load test your Postgres setup to determine the level of optimization required and what the final configuration might look like for your messaging use case. If the effort is too high or the configuration becomes overly complex, consider using a specialized message queue instead.

In the DMV example, the message volume is relatively low, and we can use Postgres as a queue without spending much time on optimization. However, if you're building a large-scale application such as a global eCommerce platform or social network and adopting Postgres for the message volume becomes difficult, reconsider whether it's the right fit for the message queue use case.

3 *You already use Postgres and don't want to introduce another solution into your architecture.*

If your application already uses Postgres and now needs to support a message queue use case, consider using Postgres first before bringing in a specialized solution. If Postgres proves to be a good fit, your overall architecture stays the same, and your team won't need to learn, optimize, or maintain an additional messaging system.

At the same time, if you don't—and aren't planning to—use Postgres for other scenarios discussed in this book (such as a relational, vector, or time-series database),

and you're thinking of using it solely as a message queue, weigh that carefully. For example, if the service for the DMV uses another database for visitor check-ins and other business operations, bringing in Postgres just for the message queue might not be the most optimal choice. Instead, see if the existing database can support the messaging use case, and if not, consider a specialized messaging solution first.

Now that we understand how Postgres supports essential capabilities expected from message queues and when to use it for such workloads, let's build a custom queue on top of Postgres to see what the database has to offer. After building the queue from scratch and learning its implementation specifics, we'll explore how to use `pgmq`—an extension that offers a ready-to-use queue.

11.2 Building a custom message queue

In this section, we'll implement the following three database functions to see how Postgres can be used as a message queue:

- `mq.enqueue(new_message JSON)`—Adds a new message of type `JSON` to the queue.
- `mq.dequeue(messages_cnt INT)`—Retrieves the next `messages_cnt` messages from the top of the queue. The retrieved messages are invisible to other consumers but remain in the queue until the current consumer calls the `mq.mark_completed` function.
- `mq.mark_completed(message_ids BIGINT[], to_delete BOOLEAN DEFAULT FALSE)`—Marks the messages with the specified IDs as completed (successfully processed). If `to_delete` is set to `TRUE`, the messages are deleted from the queue instead of being marked as completed.

NOTE If you'd like to gain practical experience while reading this chapter, connect to any Postgres container started in the previous chapters. Use the `docker exec -it <container-name> psql -U postgres` command, replacing the `<container-name>` placeholder with the actual container name.

The queue will be implemented on top of a regular Postgres table and used by those three functions that add, retrieve, and update the status of messages. Let's use the statements from the following listing to create a dedicated schema for the queue, a custom type for message status, and the actual queue table.

Listing 11.1 Creating a queue as a regular Postgres table

```
CREATE SCHEMA mq;

CREATE TYPE mq.status AS ENUM ('new', 'processing', 'completed');

CREATE TABLE mq.queue (
    id BIGSERIAL PRIMARY KEY,
    message JSON NOT NULL,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    status mq.status NOT NULL DEFAULT 'new'
);
```

The structure of the queue table is minimalistic yet detailed enough to uniquely identify each message and retrieve them in the order they were added:

- `id`—Message IDs are stored as 64-bit integers, autoincremented by Postgres whenever a new message is added. This gives us up to 2^{63} unique message IDs. If you don't expect that many messages over the lifetime of the queue, you can use the `SERIAL` type instead, which generates 32-bit integers. Both the `BIGSERIAL` and `SERIAL` data types rely on Postgres sequences for ID generation. If you have a scenario where many producers are publishing messages concurrently, each will have to wait for the sequence to generate the next ID. If this becomes a bottleneck, consider switching to the `UUID` type and generating IDs with a function like `gen_random_uuid()`.
- `message`—The actual message payload, stored as a JSON object. In this case, we use the `JSON` type instead of `JSONB`, because we only need to store the original message from the producer and return it to a consumer later. The `JSONB` type would preprocess messages before storing them, which might slow down ingestion and alter the original structure—for example, by reordering object keys. As discussed in chapter 5, preprocessing is beneficial when traversing, querying, or manipulating `JSONB` data in Postgres. But in a message queue, where messages simply need to be passed from producers to consumers without transformation, the `JSON` data type is a better fit.
- `created_at`—The timestamp when the message was added to the queue. We'll use this column to return messages to consumers in FIFO (first in, first out) order.
- `status`—The current status of the message. The `new` status means the message has been published but not yet retrieved from the queue. The `processing` status is set when a message has been retrieved by a consumer and is currently being processed. While in this state, the message must not be visible to other consumers. The `completed` status indicates that the message has been successfully processed and can later be archived or deleted. The `processing` status is also useful in failure scenarios when a consumer retrieves a message, begins processing, and then fails. In such cases, your application can include logic that periodically checks for messages that remained in the `processing` state for too long and resets their status to `new` based on a time threshold defined by your application.

Next, let's create the `mq.enqueue` function that producers can use to add new messages to the queue. The next listing shows the function's implementation written in the PL/pgSQL procedural language.

Listing 11.2 Creating a function for adding messages to the queue

```
CREATE OR REPLACE FUNCTION mq.enqueue(new_message JSON)
RETURNS VOID AS $$
BEGIN
  INSERT INTO mq.queue (message)
  VALUES (new_message);
```

```
END;
$$ LANGUAGE plpgsql;
```

The function accepts a new message in `JSON` format and inserts it into the `mq.queue` table. All other fields in the new record are automatically initialized by Postgres: the `id` column is set to the next value from the sequence, `created_at` is set to the current timestamp, and `status` is set to `new`.

The implementation of the `mq.dequeue` function, which is used by consumers to retrieve messages from the queue, is shown next.

Listing 11.3 Creating a function for retrieving messages from the queue

```
CREATE OR REPLACE FUNCTION mq.dequeue(messages_cnt INT)
RETURNS TABLE (msg_id BIGINT, message JSON, enqueued_at TIMESTAMPTZ) AS $$
BEGIN
  RETURN QUERY
  WITH new_messages AS (
    SELECT id FROM mq.queue
    WHERE status = 'new' ORDER BY created_at
    FOR UPDATE SKIP LOCKED
    LIMIT messages_cnt
  )
  UPDATE mq.queue q
  SET status = 'processing'
  FROM new_messages WHERE q.id = new_messages.id
  RETURNING q.id, q.message,
    date_trunc('seconds',q.created_at) AS created_at;
END;
$$ LANGUAGE plpgsql;
```

The function accepts the `messages_cnt` argument that specifies the number of messages a consumer wants to retrieve and process. When the function is called, the following happens:

- The `new_messages` CTE (common table expression) is evaluated first. It finds all messages with the `status` of `new` and orders them by `created_at` time in ascending order. With that, the messages added earlier to the queue will be at the top of the result, and the `LIMIT` clause is used to return up to `messages_cnt` messages.
- The `UPDATE` statement changes the `status` column of the messages returned by the CTE to `processing`.
- The `RETURNING` clause returns message rows in which `status` is changed to `processing`, and the `RETURN QUERY` statement adds those rows to the function's result set. Each row includes the `id`, `message`, and `created_at` values of the message. The returned `created_at` value is truncated to seconds using the `date_trunc` function.

TIP Refer to chapters 2 and 3 if you need a refresher on PL/pgSQL functions or CTEs in Postgres.

The `FOR UPDATE SKIP LOCKED` clause in the CTE is used to optimize scenarios where the queue is accessed by multiple consumers concurrently. Specifically, the `FOR UPDATE` clause ensures that only one consumer can work with the messages returned by the `new_messages` CTE. It does this by putting row-level locks on every message included in the CTE result. These locks are released only when the current transaction executing the `mq.dequeue` function is either committed or rolled back.

The current transaction can be an explicit one that runs `mq.dequeue` alongside other business logic in a `BEGIN...COMMIT` block, as shown in the following snippet:

```
BEGIN;
--execute some business logic
mq.dequeue(1);
--execute more business logic;
COMMIT;
```

In this case, the locks are released after the `COMMIT` statement executes successfully or if the explicit transaction is rolled back due to a failure.

If the `mq.dequeue` function is not executed as part of an explicit transaction block, then Postgres starts an implicit transaction when the function is invoked and commits it when the function finishes execution. In this case, the locks on the message rows are released once the function completes.

The `SKIP LOCKED` clause tells Postgres to skip message rows that are already locked by other transactions. When multiple consumers run `mq.dequeue` concurrently, each one accesses only those messages that aren't currently being processed by others. This allows consumers to process new messages in parallel without blocking each other, improving overall throughput.

Finally, let's add the `mq.mark_completed` function, which consumers use to mark successfully processed messages as completed or remove them from the queue.

Listing 11.4 Creating a function for changing message status

```
CREATE OR REPLACE FUNCTION mq.mark_completed(
  message_ids BIGINT[], to_delete BOOLEAN DEFAULT FALSE)
RETURNS VOID AS $$
BEGIN
  IF to_delete THEN
    DELETE FROM mq.queue
    WHERE id = ANY(message_ids);
  ELSE
    UPDATE mq.queue
    SET status = 'completed'
    WHERE id = ANY(message_ids);
  END IF;
END;
$$ LANGUAGE plpgsql;
```

When the function is invoked, consumers pass an array with the IDs of successfully processed messages. If the `to_delete` flag is set to `TRUE`, the function deletes those messages

from the queue. If `to_delete` is set to `FALSE`, the messages remain in the queue, but their status is updated to `completed`.

With that, we've implemented a custom message queue using Postgres' built-in capabilities and are ready to see how it works in action.

11.3 Using a custom queue

Let's revisit the DMV scenario from the beginning of the chapter and see how our custom queue can be used to manage a waiting list for visitors seeking specific services. Before a DMV location opens, its waiting queue of visitors is empty. We can check that by querying the `mq.queue` table directly:

```
SELECT * FROM mq.queue;
```

The output confirms that there are no visitors yet:

```
 id | message | created_at | status
----+---------+-----------+--------
(0 rows)
```

Next, suppose the location opens at 8:00 a.m., and Emily Carter, the first visitor, arrives to register her new car. She stops by the registration terminal and checks in for the service. Once she is successfully checked in, the terminal puts Emily in line by calling the `mq.enqueue` function:

```
SELECT mq.enqueue('{
  "service": "car_registration",
  "visitor": "Emily Carter"
}'::json);
```

In this case, the message will be added to the queue only if all operations succeed and the transaction commits with no problems.

Adding a message to the queue as part of a larger business transaction

The `mq.enqueue` function can be used as part of an external transaction that executes several business operations atomically:

```
BEGIN;
--execute some business logic
SELECT mq.enqueue('{
  "service": "car_registration",
  "visitor": "Emily Carter"
}'::json);
--execute more business logic;
COMMIT;
```

Several minutes later, two more visitors, Liam Rodriguez and Ava Thompson, arrive at the DMV and register for other services. Liam came for a driving license test, and Ava needs to change the plate number for her car. The terminal adds Liam to the waiting queue first

```
SELECT mq.enqueue('{
  "service": "driving_license_exam",
  "visitor": "Liam Rodriguez"
}'::json);
```

and then registers Ava:

```
SELECT mq.enqueue('{
  "service": "car_plate_renewal",
  "visitor": "Ava Thompson"
}'::json);
```

All three visitors have now been added to the Postgres queue table and are sitting in the waiting room, ready to be served:

```
SELECT id, message,
  date_trunc('seconds',created_at) AS created_at,
  status
FROM mq.queue
ORDER BY created_at;

 id |                message                 |       created_at       | status
----+----------------------------------------+------------------------+-------
  1 | {                                     +| 2025-06-19 08:03:16+00 | new
    |   "service": "car_registration",      +|                        |
    |   "visitor": "Emily Carter"           +|                        |
    | }                                      |                        |
  2 | {                                     +| 2025-06-19 08:05:28+00 | new
    |   "service": "driving_license_exam",+|                        |
    |   "visitor": "Liam Rodriguez"         +|                        |
    | }                                      |                        |
  3 | {                                     +| 2025-06-19 08:07:43+00 | new
    |   "service": "car_plate_renewal",     +|                        |
    |   "visitor": "Ava Thompson"           +|                        |
    | }                                      |                        |
(3 rows)
```

Now, one of the officers starts their workday, logs in to the system, and calls the first visitor. The system pulls the first visitor from the queue by calling the `mq.dequeue` function:

```
SELECT * FROM mq.dequeue(1);
```

Because the officer can serve only one visitor at a time, the system passes `1` as the function argument, asking Postgres to return the first new message (visitor) from the top of the queue. That visitor is Emily Carter, who registered first:

```
 msg_id |              message              |      enqueued_at
--------+-----------------------------------+-----------------------
      1 | {                                +| 2025-06-19 08:03:16+00
        |   "service": "car_registration",+|
        |   "visitor": "Emily Carter"     +|
        | }                                 |
(1 row)
```

The display in the waiting area announces that Emily is ready to be served, and she walks up to the window where the officer is working.

Shortly after that, a second officer starts their workday and calls in the next visitor, who happens to be Liam Rodriguez:

```
SELECT * FROM mq.dequeue(1);
```

```
 msg_id |                message                |      enqueued_at
--------+---------------------------------------+-----------------------
      2 | {                                    +| 2025-06-19 08:05:28+00
        |   "service": "driving_license_exam",+|
        |   "visitor": "Liam Rodriguez"       +|
        | }                                     |
(1 row)
```

If we check the Postgres `mq.queue` table now, we'll see that the `status` of Emily's and Liam's records is set to `processing`, and it will remain in that state until the officers finish serving them:

```
SELECT id, message,
  date_trunc('seconds',created_at) AS created_at,
  status
FROM mq.queue
ORDER BY created_at;
```

The output is as follows:

```
 id |                message                |      created_at      |   status
----+---------------------------------------+---------------------+-----------
  1 | {                                    +| 2025-06-19 08:03:16 | processing
    |  "service": "car_registration",     +|                     |
    |  "visitor": "Emily Carter"          +|                     |
    | }                                     |                     |
  2 | {                                    +| 2025-06-19 08:05:28 | processing
    |  "service": "driving_license_exam", +|                     |
    |  "visitor": "Liam Rodriguez"        +|                     |
    | }                                     |                     |
  3 | {                                    +| 2025-06-19 08:07:43 | new
    |  "service": "car_plate_renewal",    +|                     |
    |  "visitor": "Ava Thompson"          +|                     |
    | }                                     |                     |
(3 rows)
```

NOTE In the previous and some following outputs, the value of the `created_at` column was truncated by removing the time zone information (+00) at the end. This was done to ensure the output is formatted properly in the book, but you will still see the time zone in the output on your end.

At some point, the first officer finishes registering Emily's new car, and the system updates her record in the queue by marking it as `completed`:

```
SELECT mq.mark_completed(ARRAY[1]);
```

As we can see, the DMV system passes `ARRAY[1]` as the argument to the function, where `1` corresponds to the ID of Emily's record in the queue.

The same officer is ready for the next visitor. The system makes another call to the `mq.dequeue` function, which retrieves the next visitor from the top of the queue:

```
SELECT * FROM mq.dequeue(1);
```

And that visitor is Ava Thompson, who registered last at the terminal.

```
msg_id  |               message               |      enqueued_at
--------+-------------------------------------+-----------------------
      3 | {                                  +| 2025-06-19 08:07:43+00
        |   "service": "car_plate_renewal",+|
        |   "visitor": "Ava Thompson"      +|
        | }                                   |
(1 row)
```

If we check the Postgres queue table now, we'll see that the `status` of Emily's record is set to `completed`, whereas the other two records have the `status` of `processing`, because both Liam and Ava are still being served by the officers:

```
SELECT id, message,
  date_trunc('seconds',created_at) AS created_at,
  status
FROM mq.queue
ORDER BY created_at;
```

```
 id |               message                |     created_at      |  status
----+--------------------------------------+---------------------+---------
  1 | {                                   +| 2025-06-19 08:03:16 | completed
    |  "service": "car_registration",    +|                     |
    |  "visitor": "Emily Carter"         +|                     |
    | }                                    |                     |
  2 | {                                   +| 2025-06-19 08:05:28 | processing
    |  "service": "driving_license_exam",+|                     |
    |  "visitor": "Liam Rodriguez"       +|                     |
    | }                                    |                     |
  3 | {                                   +| 2025-06-19 08:07:43 | processing
    |  "service": "car_plate_renewal",   +|                     |
```

```
    |  "visitor": "Ava Thompson"        +|                      |
    | }                                  |                      |
(3 rows)
```

The completed records can be archived and removed from the queue table at a later time. If needed, the records can also be deleted immediately by setting the second argument (`to_delete`) of the `mq.mark_completed` function to `TRUE`. Let's do this by deleting Liam's record from the queue once the officer is finished serving him:

```
SELECT mq.mark_completed(ARRAY[2], TRUE);
```

Let's query the Postgres queue table one last time to confirm that Liam's record is, in fact, deleted from the queue:

```
SELECT id, message,
  date_trunc('seconds',created_at) AS created_at,
  status
FROM mq.queue
ORDER BY created_at;

 id |              message               |     created_at      |   status
----+------------------------------------+---------------------+------------
  1 | {                                 +| 2025-06-19 08:03:16 | completed
    |   "service": "car_registration", +|                     |
    |   "visitor": "Emily Carter"      +|                     |
    | }                                  |                     |
  3 | {                                 +| 2025-06-19 08:07:43 | processing
    |   "service": "car_plate_renewal",+|                     |
    |   "visitor": "Ava Thompson"      +|                     |
    | }                                  |                     |
(2 rows)
```

By walking through this simple yet practical example, we've seen how applications can take advantage of our queue by calling the `mq.enqueue`, `mq.dequeue`, and `mq.mark_completed` functions.

11.4 Using LISTEN and NOTIFY

Imagine that after the early morning peak hours, it's less busy at the DMV location. There are no visitors in the waiting area, and most of the officers are occupied with secondary tasks. However, when a new visitor comes in and registers for a service, the DMV system automatically notifies the officers that someone is in line. Let's see how such a notification component can be implemented using the `LISTEN` and `NOTIFY` capabilities of Postgres.

The definition of the `LISTEN` command looks as follows:

```
LISTEN channel_name
```

It registers the current database session as a listener on the given channel.

The NOTIFY command can then be used to send a notification to all listeners subscribed to that channel, and the listeners can execute application-specific logic in response to the notification:

```
NOTIFY channel_name, [payload]
```

The payload argument is optional and can be used to pass extra information to the listeners in text form.

Suppose that when a DMV officer logs in to their machines, the system opens a connection with Postgres and registers for notifications on the queue_new_message channel. Execute the following command in your open psql connection to emulate this scenario:

```
LISTEN queue_new_message;
```

Next, as shown in the following listing, let's update the implementation of the mq.enqueue function to send a notification on the queue_new_message channel every time a new visitor is added to the queue.

Listing 11.5 Updating the mq.enqueue function to send notifications

```
CREATE OR REPLACE FUNCTION mq.enqueue(new_message JSON)
RETURNS VOID AS $$
BEGIN
  INSERT INTO mq.queue (message)
  VALUES (new_message);

  -- Notify listeners that a new message has been added
  PERFORM pg_notify('queue_new_message', 'new_message');
END;
$$ LANGUAGE plpgsql;
```

The mq.enqueue function sends notifications using the pg_notify function, which is similar to the NOTIFY statement but easier to work with when we need to implement database functions and stored procedures.

Finally, let's verify that the implemented notification approach works by adding a new visitor to the queue using the same psql connection that is already listening on the queue_new_message channel:

```
SELECT mq.enqueue('{
  "service": "car_plate_renewal",
  "visitor": "Marta Jones"
}'::json);
```

Once this function is executed, the psql connection will output a message similar to the following:

```
Asynchronous notification "queue_new_message" with payload "new_message"
 received from server process with PID 1699.
```

However, in a real-world scenario, visitors are added to the queue from a registration terminal, which uses a separate connection to the database. Let's refer to the steps in table 11.1 to emulate a setup where officers are connected to the database and listening for notifications using their own sessions, while the notifications are sent from a different session opened by the terminal.

Table 11.1 Sending notification from a second psql session

Initial psql session (officer's desk)	Second psql session (registration terminal)
Continue using the psql connection that is already listening on the `queue_new_message` channel.	Open a second psql connection from a different terminal window: `docker exec -it postgres psql -U postgres`

	Add a new visitor to the queue:
	`SELECT mq.enqueue('{`
	`"service": "driving_license_exam",`
	`"visitor": "Roland Deschain"`
	`}'::json);`
	The function will send a notification on the `queue_new_message` channel.

We won't see a notification until any SQL statement is executed via the current psql connection. For example, run the following command:
`SELECT 1;`
Right after that, the psql connection will print the delivered notification:
`Asynchronous notification "queue_new_message" with payload "new_message" received from server process with PID 1746.`	

The first psql connection didn't automatically print the notification because psql doesn't implement an event loop that would continuously listen for incoming notifications sent from other database connections. Instead, psql checks for and reports notifications only when the next SQL statement, such as `SELECT 1`, is executed through the connection.

However, this doesn't mean you necessarily need to implement polling logic to check whether the current database session has new notifications on the channel. Most major programming languages have Postgres drivers that handle this internally, delivering events to the application layer asynchronously as they arrive.

NOTE Why was the notification printed in the first test scenario when we added Marta Jones to the queue? In that case, the visitor was enqueued using the same psql connection that was already listening for notifications. So the connection effectively notified itself, which doesn't require a special event loop.

After seeing how the custom queue works in action for the DMV use case, let's go over some best practices and limitations you may need to consider for your own scenario.

11.5 Queue implementation considerations

The implemented custom queue is already optimal for the DMV scenario and likely doesn't require any additional optimizations. The ingestion rate is low and defined by the number of visitors and terminals available for check-in. The retrieval rate is also low and is bounded by the number of officers and their availability. Both ingestion and retrieval rates will vary, with peaks in the morning during rush hours and slower periods toward the end of the day.

However, the implemented queueing solution might not fully satisfy the requirements of your use case. In this section, we'll walk through several additional considerations to keep in mind when using Postgres as a message queue.

11.5.1 LISTEN and NOTIFY considerations

Let's begin with the `LISTEN/NOTIFY` feature, which we introduced in the previous section. Although it does what it's designed for by delivering notifications from Postgres to the application layer, it comes with two limitations you need to be aware of.

First, notifications are received only by those subscribers (clients) who are currently connected to Postgres and actively listening on a channel. Clients that connect later or get restarted won't receive any notifications that were fired before they began listening, because Postgres doesn't maintain a backlog of past events. This limitation doesn't prevent us from using `LISTEN/NOTIFY` for the DMV use case. Even if a visitor checks in 10 seconds before an officer finishes connecting to the system, the officer will miss the notification about the new visitor, but they'll still call the next visitor once they're fully connected and ready to work.

Second, if you're using Postgres in a configuration with a primary and one or more read replica nodes, clients must subscribe to notifications on the primary node. Currently, `LISTEN/NOTIFY` is not supported for replicas. To work around this limitation, your application can maintain a dedicated connection to the primary node specifically for `LISTEN/NOTIFY`. All other queries can continue to be served through a connection pool or load balancer that distributes the workload across both the primary and available replicas.

This is exactly what we can do for the DMV use case by establishing direct connections to the primary from the terminal and officers' workstations. The terminal will use its connection to check visitors in and issue notifications on the channel, and the officers will receive those notifications via their own connections to the primary. Other

queries from officers' workstations can go through a connection pool that offloads the primary by routing read requests to replicas.

11.5.2 Indexing considerations

The `mq.queue` table that stores messages for our custom message queue has only one index defined on the `id` column, which is the table's primary key. We can confirm this by executing the following query:

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE schemaname = 'mq' AND tablename = 'queue';
```

The index definition looks as follows:

```
 indexname  |                          indexdef
------------+----------------------------------------------------------------
 queue_pkey | CREATE UNIQUE INDEX queue_pkey ON mq.queue USING btree (id)
(1 row)
```

This index can help optimize the execution of the `mq.mark_completed` function, which accepts an array of message IDs whose status needs to be updated or which need to be removed from the queue. The index allows the database to quickly locate all rows matching the given IDs and then either update or delete them.

However, the `mq.dequeue` function, which retrieves the next message from the queue, currently performs a full table scan each time a consumer retrieves the next unprocessed message. There is no index on the `created_at` column, which means the table data must be sorted when a consumer selects the next message with the earliest `created_at` value. Nor is there an index on the `status` column to help Postgres skip over messages with a status other than `new`.

In the case of our DMV use case, we might not need any additional indexes, because the visitor queue is always empty by the next business day. Completed messages from the previous day can either be archived or deleted. If your use case involves much higher message ingestion and retrieval rates, consider adding secondary indexes to speed up the `mq.dequeue` function.

One option is a single-column index on the `created_at` column, which allows Postgres to efficiently retrieve the next message (the earliest one added to the queue) without sorting the entire table:

```
CREATE INDEX mq_created_at_index_btree
ON mq.queue (created_at);
```

Alternatively, you can create a partial index on the `created_at` and `status` columns that includes only messages with a status of `new`. This keeps the index structure more compact and allows Postgres to retrieve new messages even more efficiently by skipping those marked as `processing` or `completed`:

```
CREATE INDEX mq_partial_index_btree
ON mq.queue (created_at, status)
WHERE status = 'new';
```

Finally, you can select another indexing strategy tailored more to your specific use case, especially if you decide to modify the discussed custom message queue.

11.5.3 Partitioning considerations

In chapter 9, we explored how Postgres partitioning capabilities can be used to optimize time-series workloads. Instead of storing all time-series data in a single table, Postgres can automatically arrange the data across several child tables (partitions), making searches more efficient and simplifying the management of historical data. Partitioning can also be successfully applied to message queue scenarios, especially those dealing with high volumes of data.

Partitioning is not necessary for our DMV scenario; however, if your application has a high ingestion rate, the size of the queue table can grow quickly, potentially affecting the performance of functions that retrieve the next messages from the queue or update their status. For instance, even if processed messages are deleted, they don't disappear immediately. Deleted rows (dead tuples) remain in the internal table structure and are still visible to the database engine until they are garbage collected by Postgres's autovacuum process. To keep up with your workload, you may need to tune the vacuum settings and configure Postgres to remove dead tuples more aggressively from the physical table structure.

> **NOTE** Dead tuples are also created whenever you update an existing record—for instance, by changing the status of a message from `new` to `processing`. That's because an `UPDATE` statement deletes the previous version of the record and inserts a new one with the updated values. The deleted version remains in the table structure until it's garbage-collected by the autovacuum process. If you're curious to see how this works in action, check out the "PostgreSQL Internals in Action: MVCC" video listed in the book's repository: https://github.com/dmagda/just-use-postgres-book/blob/main/postgres_internals_videos.md.

With partitioning, instead of storing all queue messages in a single table, Postgres can distribute the messages across several partitions, making it easier to support high-volume workloads without requiring extensive tuning of vacuum-related or other database settings. One partitioning strategy is range-based partitioning on the `created_at` column of the `mq.queue` table. Here's how the table definition will look if you want to enable this type of partitioning:

```
DROP TABLE mq.queue;

CREATE TABLE mq.queue (
    id BIGSERIAL,
    message JSON NOT NULL,
    created_at TIMESTAMPTZ DEFAULT NOW(),
```

```
    status mq.status NOT NULL DEFAULT 'new',
    PRIMARY KEY (id, created_at)
) PARTITION BY RANGE (created_at);
```

In this case, Postgres will arrange messages across partitions based on their `created_at` value, and the partitions need to be created separately. The following examples show how to create child tables (partitions) for June 22, 2025; June 23, 2025; and the default:

```
CREATE TABLE mq.queue_2025_06_22 PARTITION OF mq.queue
FOR VALUES FROM ('2025-06-22 00:00:00') TO ('2025-06-23 00:00:00');

CREATE TABLE mq.queue_2025_06_23 PARTITION OF mq.queue
FOR VALUES FROM ('2025-06-23 00:00:00') TO ('2025-06-24 00:00:00');

CREATE TABLE mq.queue_default PARTITION OF mq.queue DEFAULT;
```

Postgres will automatically place all messages ingested on 2025-06-22 into the `mq.queue_2025_06_22` partition, and messages created on 2025-06-23 will go to the `mq.queue_2025_06_23` table instead. Messages with a different date will end up in the default partition named `mq.queue_default`. And as we learned in chapter 9, Postgres handles this transparently—meaning that all our custom queue functions (`mq.enqueue`, `mq.dequeue`, and `mq.mark_completed`) can continue working with the parent `mq.queue` table directly, letting the database decide which partition to insert into or query from.

If we suppose that today is June 23, 2025, all new messages will be inserted into and retrieved from the `mq.queue_2025_06_23` partition transparently. Other partitions storing outdated messages, such as `mq.queue_2025_06_22`, can be removed later to reduce resource utilization and ensure that Postgres no longer considers those partitions when selecting execution plans for queries over the `mq.queue` parent table. With this approach, we can retain only a single or a small number of partitions storing the latest messages, letting Postgres handle high-volume messaging workloads more efficiently without spending too much time tuning the database configuration.

> **TIP** Postgres offers extensions that can automate the creation of partitions for messages with specific dates and the removal of partitions containing legacy data. For example, we can use a combination of `pg_partman` and `pg_cron` for this purpose. The `pg_partman` extension automates partition creation and maintenance through its built-in background worker, and `pg_cron` provides a cron-based scheduler in the database that, if necessary, can run `pg_partman`'s maintenance tasks at specific times or on a recurring schedule.

The partitioning strategy we've discussed is not the only one you can use for message queue scenarios in Postgres. If it better suits your use case, messages can be partitioned by the `status` column, with a dedicated partition for each message status (`new`, `processing`, and `completed`). You can also partition by both the `created_time` and `status` columns. Or there might be another option that works best for your scenario.

11.5.4 Messages processing failover considerations

In our DMV scenario, a message representing a checked-in visitor waiting in line is moved from `new` to the `processing` state as soon as an officer calls the visitor. If the visitor doesn't show up at the officer's desk within a minute, the message status is changed from `processing` back to `new`, putting the visitor back at the top of the waiting queue. If the visitor does arrive at the officer's desk, the message remains in the `processing` state until the officer finishes working on the case, changing the message status to `completed`. Overall, it's guaranteed that a message won't get stuck in the `processing` state once the officer begins assisting the visitor.

However, in other scenarios, a consumer might retrieve the next message from the queue, start processing it, and then fail for some reason. For instance, consider a job scheduling system where producers enqueue tasks and workers later retrieve and execute them. If a worker fails after retrieving one or more tasks, those tasks will remain in the `processing` state.

To mitigate this, we can introduce a failover mechanism that allows a message to remain in the `processing` state for only a limited time. Once that time threshold expires, the message status is reset to `new`, allowing another worker to pick it up for processing. This can be automated using the `pg_cron` extension by scheduling a periodic job in the database to check for messages stuck in the `processing` state and reset their status to `new`.

11.6 Starting Postgres with pgmq

We've now seen how to implement and use a custom message queue in Postgres and discussed several important implementation considerations. With that knowledge in mind, we're ready to move on to the `pgmq` extension, which provides a ready-to-use message queue implementation.

The `pgmq` (Postgres Message Queue) extension positions itself as a lightweight message queue with API parity with AWS SQS (Simple Queue Service). The extension abstracts away the implementation details we had to take care of while building the custom message queue and supports advanced capabilities, including queue partitioning and message processing failover. Let's learn how to get started with the `pgmq` extension by deploying it in a Docker container and then practice using it for the DMV scenario.

Stop the already-running Postgres container

Before you deploy the container with `pgmq`, stop any already-running Postgres container to free up port 5432. First, find the active container that is listening on port 5432:

```
docker ps --filter "name=postgres" --filter "publish=5432"
```

The command might output a result similar to the following, indicating that a container is currently running:

```
IMAGE            STATUS          PORTS                     NAMES
postgres:latest  Up 50 minutes   0.0.0.0:5432->5432/tcp    <container-name>
```

Stop the container by providing its name in the `<container-name>` placeholder:

```
docker container stop <container-name>
```

If you're on a Unix operating system such as Linux or macOS, use the following command to start a Postgres container with `pgmq`.

Listing 11.6 Starting Postgres with `pgmq` on Unix

```
docker volume create postgres-pgmq-volume

docker run --name postgres-pgmq \
    -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password \
    -p 5432:5432 \
    -v postgres-pgmq-volume:/var/lib/postgresql/data \
    -d ghcr.io/pgmq/pg17-pgmq:v1.5.1
```

If you're a Windows user, run the following command in PowerShell instead. If you use CMD, replace each backtick (`` ` ``) with a caret (`^`) at the end of each line.

Listing 11.7 Starting Postgres with `pgmq` on Windows

```
docker volume create postgres-pgmq-volume

docker run --name postgres-pgmq `
    -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=password `
    -p 5432:5432 `
    -v postgres-pgmq-volume:/var/lib/postgresql/data `
    -d ghcr.io/pgmq/pg17-pgmq:v1.5.1
```

Both commands start the `postgres-pgmq` container using the `pg17-pgmq:v1.5.1` image that includes the `pgmq` extension. The container listens for incoming connections on port 5432 and stores Postgres data in the Docker-managed volume named `postgres-pgmq-volume`.

> **NOTE** The way we deploy Postgres in Docker works well for development and for exploring the database's capabilities. However, if you plan to use this deployment option in production, be sure to review security and other deployment best practices.

Once the container is started, connect to it using the psql client:

```
docker exec -it postgres-pgmq psql -U postgres
```

Confirm that the `pgmq` extension exists in the Postgres container by executing the next command.

Listing 11.8 Checking the available `pgmq` version

```
SELECT * FROM pg_available_extensions
WHERE name = 'pgmq';
```

The output should look similar to the following:

```
 name | default_version | installed_version |            comment
------+-----------------+-------------------+-----------------------------
 pgmq | 1.5.1           |                   | A lightweight message queue.
      |                 |                   | Like AWS SQS and RSMQ
      |                 |                   | but on Postgres.
```

The `installed_version` column is empty, which means `pgmq` has not been enabled for the `postgres` database that the psql tool is connected to. You can always check the current database using the `\conninfo` command of the psql tool, which returns output like this:

```
\conninfo

You are connected to database "postgres" as user "postgres" via socket in
➥ "/var/run/postgresql" at port "5432".
```

Let's enable the extension for the postgres database by executing the following command:

```
CREATE EXTENSION pgmq;
```

After that, execute the query from listing 11.8 one more time, to confirm that the `installed_version` column is now set to `1.5.1`:

```
 name | default_version | installed_version |            comment
------+-----------------+-------------------+------------------------------
 pgmq | 1.5.1           | 1.5.1             | A lightweight message queue.
      |                 |                   | Like AWS SQS and RSMQ
      |                 |                   | but on Postgres.
```

Now, let's see how to use the extension's capabilities instead of our previously created custom queue to manage the waiting list of visitors coming to the DMV office.

11.7 Using pgmq

The `pgmq` extension provides more than 40 functions that let us create and manage queues, insert and retrieve messages, archive data with a custom retention policy, and more. The functions are available under the `pgmq` schema and can be listed by executing the following query.

Listing 11.9 Listing functions supported by `pgmq`

```
SELECT p.proname AS function_name,
  pg_catalog.pg_get_function_arguments(p.oid) AS arguments
FROM pg_catalog.pg_proc p
JOIN pg_catalog.pg_namespace n ON n.oid = p.pronamespace
WHERE n.nspname = 'pgmq';
```

The first five functions returned by the query are as follows:

```
        function_name          |                 arguments
-------------------------------+------------------------------------------
 acquire_queue_lock            | queue_name text
 format_table_name             | queue_name text, prefix text
 read                          | queue_name text, vt integer, qty integer,
                               |   conditional jsonb DEFAULT '{}'::jsonb
 send                          | queue_name text, msg jsonb,
                               |     delay timestamp with time zone
 send                          | queue_name text, msg jsonb, headers jsonb,
                               | delay integer
```

Let's use several of the provided functions to create and manage a queue of DMV visitors.

11.7.1 Creating and using a visitors queue

First, let's create a dedicated queue named `visitors_queue` using the `pgmq.create` function:

```
SELECT pgmq.create('visitors_queue');
```

Each created queue is backed by its own Postgres table stored under the `pgmq` schema. The table name is a combination of the `q_` prefix and the queue name. In the case of the `visitors_queue`, the underlying table is named `q_visitors_queue`, and we can explore its structure by running the following command in psql:

```
\d pgmq.q_visitors_queue;
```

The output shows a list of columns used to represent a message record:

```
                        Table "pgmq.q_visitors_queue"
   Column    |           Type           | Collation | Nullable |   Default
-------------+--------------------------+-----------+----------+------------
 msg_id      | bigint                   |           | not null | generated
                                        |           |          | always
                                        |           |          | as identity
 read_ct     | integer                  |           | not null | 0
 enqueued_at | timestamp with time zone |           | not null | now()
 vt          | timestamp with time zone |           | not null |
 message     | jsonb                    |           |          |
 headers     | jsonb                    |           |          |
```

```
Indexes:
    "q_visitors_queue_pkey" PRIMARY KEY, btree (msg_id)
    "q_visitors_queue_vt_idx" btree (vt)
```

Each message has a unique `msg_id` and a payload stored in the `message` column of type `jsonb`. The time the message was enqueued is stored in the `enqueued_at` column, and the `vt` column stores the visibility timeout. This timeout allows a previously retrieved message to reappear in the queue if the consumer doesn't finish processing it within the `vt` window due to a failure or some other reason. The `headers` column stores an optional message header, and the `read_ct` column tracks how many times the message has been read using the `pgmq.read` function. As we can also see, the extension creates several default indexes, which are used to speed up access to enqueued messages.

Next, let's use the `pgmq.send` function to add a new checked-in visitor to the waiting list:

```
SELECT * FROM pgmq.send(
  queue_name  => 'visitors_queue',
  msg =>'{"service": "car_registration", "visitor": "Olivia Morgan"}'
);
```

The `queue_name` argument specifies the name of the queue the message should be added to, and the `msg` argument contains the actual message in JSON format.

When the officer is ready to serve the next visitor, the system can execute the `pgmq.pop` function, which retrieves and immediately deletes the next message from the queue:

```
SELECT msg_id, message, enqueued_at
FROM pgmq.pop('visitors_queue');
```

The function returns the following details about the next visitor:

```
 msg_id |              message              |          enqueued_at
--------+-----------------------------------+-------------------------------
      1 | {"service": "car_registration",   |  2025-06-26 01:18:19.149045+00
        |   "visitor": "Olivia Morgan"}     |
(1 row)
```

Consecutive calls to `pgmq.pop` will return an empty result until a new visitor is added to the queue.

11.7.2 Using message visibility timeouts

In addition to the `pgmq.pop` function, the `pgmq` extension provides the `pgmq.read` function, which retrieves a message from the queue without deleting it immediately. The `pgmq.read` function's definition looks as follows:

```
pgmq.read(
  queue_name text, vt integer, qty integer,
  conditional jsonb DEFAULT '{}'::jsonb)
```

The `queue_name` argument defines the name of the queue, and the `qty` argument specifies how many messages to retrieve at once. The optional `conditional` argument can be used to filter messages based on their JSON structure.

The `vt` argument (visibility timeout) defines how long a message remains invisible to other consumers after it has been read by an initial consumer. If the initial consumer doesn't delete or archive the message within that timeout, the message becomes visible again and can be processed by other consumers.

With the visibility timeout, we can support message processing failover in case the initial consumer fails, allowing other consumers to pick it up. The visibility timeout can also help implement other scenarios specific to your use case. For example, imagine that when an officer is available and calls the next visitor in line, the system gives the visitor two minutes to arrive at the officer's desk. If the visitor doesn't show up within that time, the system can automatically place them back in the waiting list and let the officer call another visitor.

Let's learn how to implement this use case using the `pgmq.read` function with a two-minute visibility timeout. Imagine that Noah Bennett registers for a driving license exam, and the terminal adds him to the waiting queue by making the following call to the `pgmq.send` function:

```
SELECT * FROM pgmq.send(
  queue_name  => 'visitors_queue',
  msg => '{"service": "driving_license_exam", "visitor": "Noah Bennett"}'
);
```

Sometime later, an officer is available to serve the next visitor, and the system calls the `pgmq.read` function.

Listing 11.10 Retrieving the next visitor with a two-minute visibility timeout

```
SELECT msg_id, message, enqueued_at
FROM pgmq.read(
  queue_name => 'visitors_queue',
  vt         => 120,
  qty        => 1
);
```

This call retrieves one visitor from the queue (`qty => 1`) and sets the visibility timeout to two minutes (`vt => 120` seconds). As it turns out, the next visitor is Noah, and his name is announced in the waiting area:

```
 msg_id |               message                |          enqueued_at
--------+--------------------------------------+------------------------------
      2 |{"service": "driving_license_exam", | 2025-06-27 00:30:10.798186+00
        |"visitor": "Noah Bennett"}          |
(1 row)
```

During the next two minutes, Noah's record in the queue won't be visible to other officers who might be ready to serve the next visitors. We can confirm this by running the query from listing 11.10 again within that time interval:

```
 msg_id | message | enqueued_at
--------+---------+------------
(0 rows)
```

But suppose Noah steps aside to take a phone call and doesn't appear at the officer's desk within the two-minute window. As a result, the `pgmq` extension puts him back on the waiting list by making his message visible to other officers again.

Several minutes later, Noah finishes his call and returns to the waiting area. Another officer becomes available, and the system executes the same `pgmq.read` function from listing 11.10, inviting Noah once again to go to the officer's desk:

```
 msg_id |              message              |         enqueued_at
--------+-----------------------------------+------------------------------
      2 |{"service": "driving_license_exam", | 2025-06-27 00:30:10.798186+00
        |"visitor": "Noah Bennett"}          |
(1 row)
```

This time, Noah doesn't miss his turn and appears at the officer's desk within the two-minute visibility timeout. The officer acknowledges that Noah has shown up, and the system calls the `pgmq.archive` function to remove Noah's record from the visitors queue:

```
SELECT pgmq.archive(
  queue_name => 'visitors_queue',
  msg_id     => 2
);
```

The `pgmq.archive` function uses Noah's record ID (`msg_id => 2`) to remove his message from the underlying `pgmq.q_visitors_queue` table used by the `visitors_queue` queue and places it in the `pgmq.a_visitors_queue` table instead. The latter table is used by the `pgmq` extension to store all archived messages, and we can confirm that Noah's message is already there by querying the table directly:

```
SELECT msg_id, message FROM pgmq.a_visitors_queue;
```

The query produces the following output:

```
 msg_id |                            message
--------+-------------------------------------------------------------
      2 | {"service": "driving_license_exam", "visitor": "Noah Bennett"}
(1 row)
```

Finally, if an officer calls the `pgmq.read` function from listing 11.10 at any later time, they will no longer find Noah in the waiting list, as his message has already been archived, and the function will return an empty result:

```
 msg_id | message | enqueued_at
--------+---------+-------------
(0 rows)
```

With that, we've learned how to use Postgres as a message queue by creating a custom solution and by using the `pgmq` extension.

> **NOTE** At the time of writing, the `pgmq` extension did not yet support the `LISTEN/NOTIFY` feature. However, this support was planned for a future release. For more details, refer to the extension's repository: https://github.com/pgmq/pgmq.

Summary

- Postgres can be used as a message queue to exchange messages, events, and tasks between producers and consumers.
- A custom message queue can be implemented on top of a regular Postgres table, allowing you to tailor the implementation to the specifics of your messaging scenario.
- The custom implementation can rely on `FOR UPDATE SKIP LOCKED` to support concurrent access by multiple consumers.
- The `LISTEN` and `NOTIFY` statements can be used to notify consumers when new messages are added to the queue.
- There are several considerations when using Postgres as a message queue, including partitioning, indexing, and failover handling for message processing.
- The `pgmq` extension provides a ready-to-use message queue for Postgres with API parity with AWS SQS (Simple Queue Service).

appendix A
Five optimization tips

I wrote this book to walk you through the breadth and depth of Postgres's capabilities. Each chapter is designed as a hands-on technical guide that lets you spin up a Postgres container, load a sample dataset, and experience a Postgres capability in action. My goal is for you to finish each chapter thinking, "Postgres can really do that too!"

To support this kind of learning, I intentionally avoid overloading you with too many details, best practices, or pitfalls: first, because too much too soon might discourage you from using any technology in the first place; and second, because your journey with Postgres shouldn't end with this book. Once you're done here, keep building real-world apps with Postgres and pick up more tips, tricks, and gotchas from other resources—and, more importantly, from your own experience.

That said, I can't resist sharing my top five query optimization tips. These should help you write more efficient code and design better solutions with Postgres. Let's run through them briefly.

A.1 Master using EXPLAIN

Throughout the book, we use the `EXPLAIN` command to analyze and compare execution plans generated by Postgres. Most of the time, a statement with `EXPLAIN` looks as follows:

```
EXPLAIN (analyze, costs off)
SELECT...
```

The `analyze` parameter requires Postgres to execute the generated plan, letting us see actual runtimes and other statistics. The `costs off` parameter makes the output more compact and readable by skipping information about the estimated cost Postgres assumed for each query phase, or the number of rows expected to be retrieved or processed.

For example, this is how an execution plan looks for one of the queries from chapter 5:

```
EXPLAIN (analyze, costs off)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"type": "custom"}';

                                QUERY PLAN
----------------------------------------------------------------------------
 Aggregate (actual time=1.501..1.502 rows=1 loops=1)
   ->  Seq Scan on order_items (actual time=0.033..1.446 rows=563 loops=1)
         Filter: (pizza @> '{"type": "custom"}'::jsonb)
         Rows Removed by Filter: 2375
 Planning Time: 0.287 ms
 Execution Time: 1.551 ms
(6 rows)
```

If we remove both parameters, Postgres starts showing estimated costs (because the `costs` parameter defaults to `true`), but there won't be any run times because the plan is not executed:

```
EXPLAIN
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"type": "custom"}';

                             QUERY PLAN
----------------------------------------------------------------------
 Aggregate  (cost=139.13..139.14 rows=1 width=8)
   ->  Seq Scan on order_items  (cost=0.00..137.72 rows=563 width=0)
         Filter: (pizza @> '{"type": "custom"}'::jsonb)
(3 rows)
```

Even though we use `EXPLAIN (analyze, costs off)` throughout the book, it doesn't mean this combination of parameters is considered a best practice and should be used all the time. It simply means these parameters provide enough detail for us to understand how Postgres executes a particular query from the book.

The `EXPLAIN` command supports many more parameters that can be useful when investigating slow queries or performing query optimizations. For example, if you're trying to optimize a long-running query, it's better to keep the estimated costs (remove `costs off`) and include information about memory buffers used by Postgres (add `buffers on`):

```
EXPLAIN (analyze, buffers on)
SELECT count(*)
```

```
FROM pizzeria.order_items
WHERE pizza @> '{"type": "custom"}';

                              QUERY PLAN
------------------------------------------------------------------------
 Aggregate  (cost=139.13..139.14 rows=1 width=8)
 ➥   (actual time=1.713..1.714 rows=1 loops=1)
   Buffers: shared hit=101
   ->  Seq Scan on order_items (cost=0.00..137.72 rows=563 width=0)
 ➥      (actual time=0.034..1.624 rows=563 loops=1)
         Filter: (pizza @> '{"type": "custom"}'::jsonb)
         Rows Removed by Filter: 2375
         Buffers: shared hit=101
 Planning Time: 0.378 ms
 Execution Time: 1.757 ms
(8 rows)
```

Having information about both estimated and actual costs helps you see how far off or close Postgres was in its estimates. This is important because during the planning phase, Postgres compares multiple execution plans based on their estimated costs. If the estimates for the selected plan are far from reality, it might mean that Postgres can't keep up with refreshing its statistics or that the query itself needs optimization.

The memory buffers section shows how often Postgres had to fetch data from disk while executing the query. For example, `Buffers: shared hit=101` in the plan output implies that all rows processed by the query were served from memory. Postgres didn't have to go to disk because the required data was already cached in 101 memory buffers. If the memory hit ratio is low and Postgres spends too much time on disk I/O, you might simply need to allocate more RAM to the database server before introducing other optimization techniques.

> **TIP** Starting with Postgres 18, which was in development while the book was being written, the database adds buffer information to the execution plan by default. Thus, you won't need to add the `buffers on` flag explicitly to the `EXPLAIN ANALYZE` or `EXPLAIN (analyze, ...)` statement.

You should master using the `EXPLAIN` statement and analyze execution plans. Don't do premature optimizations or guesswork until you've checked and understood what Postgres does and why it chooses to execute your query one way or another.

A.2 Know when to use indexes

Throughout the book, I pay special attention to indexes. Almost every chapter includes a dedicated section showing how to use a particular index type for query optimization. Why do I treat indexes so specially? Because once you understand how they work and when to apply them, your relationship with databases will change forever. The database will stop being a black box, and you'll be much more likely to start digging into its internals—which will only sharpen your skills and deepen your knowledge. And I want you to dig deeper after you're done with this book!

I do my best to show how indexes work and when to use them in the book's examples. But when you start using Postgres for your own applications, it becomes your responsibility to decide when to use an index. However, I can help you further by asking you to remember three words that will guide your decision about whether to use an index. Those words are `EXPLAIN`, *underindexing*, and *overindexing*. Let's break them down.

`EXPLAIN` is our already-familiar command that returns the query execution plan selected by Postgres. The rule here is simple: generate and understand the plan before deciding to create an index. Don't guess; always analyze the plan first. It might show that an index isn't needed at all!

Underindexing refers to situations where your queries are slow simply because there is a lack of indexes that would make the queries much more performant. Honestly, this is a pleasant problem to solve. If an execution plan shows that a slow query is doing a full table scan over a large dataset, don't rush to scale up to a machine with more CPU or memory. Consider adding an index first. A larger machine with more resources will result in significantly higher costs, whereas adding an index in an underindexing situation is often negligible.

For instance, if we take one more look at the execution plan for the following query, we'll see that Postgres has to scan the entire table to find all pizzas with a custom recipe:

```
EXPLAIN (analyze, buffers on)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"type": "custom"}';

                                 QUERY PLAN
----------------------------------------------------------------------------
 Aggregate  (cost=139.13..139.14 rows=1 width=8)
 ➥   (actual time=1.516..1.518 rows=1 loops=1)
   Buffers: shared hit=101
   ->  Seq Scan on order_items  (cost=0.00..137.72 rows=563 width=0)
 ➥       (actual time=0.037..1.454 rows=563 loops=1)
         Filter: (pizza @> '{"type": "custom"}'::jsonb)
         Rows Removed by Filter: 2375
         Buffers: shared hit=101
 Planning Time: 0.412 ms
 Execution Time: 1.565 ms
(8 rows)
```

The plan also suggests that Postgres has enough memory to cache the table records in memory because all the data was served from its buffers (`Buffers: shared hit=101`).

Considering that the application regularly filters pizza orders by type and that the number of orders will continue to grow, it's reasonable to create an index. But first, let's double-check whether there are already any indexes on the table:

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE schemaname = 'pizzeria' AND tablename = 'order_items';
```

```
    indexname     |                          indexdef
------------------+------------------------------------------------------
 order_items_pkey | CREATE UNIQUE INDEX order_items_pkey
                  | ON pizzeria.order_items USING btree
                  | (order_id, order_item_id)
(1 row)
```

There's a composite B-tree index on the `order_id` and `order_item_id` columns, but that index obviously can't be used for filtering based on the structure of JSON objects stored in the `pizza` column. So, this is an example of an underindexing scenario at the table level. We address it in chapter 5 by creating a GIN on the `pizza` column as follows:

```
CREATE INDEX idx_pizza_orders_gin
ON pizzeria.order_items USING GIN(pizza);
```

With the index in place, the database starts using the `Bitmap Index Scan` access method to expedite the search:

```
EXPLAIN (analyze, buffers on)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"type": "custom"}';

                                QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=125.34..125.35 rows=1 width=8)
➥    (actual time=0.926..0.927 rows=1 loops=1)
   Buffers: shared hit=107
   ->  Bitmap Heap Scan on order_items
➥        (cost=15.90..123.93 rows=563 width=0)
➥        (actual time=0.160..0.869 rows=563 loops=1)
         Recheck Cond: (pizza @> '{"type": "custom"}'::jsonb)
         Heap Blocks: exact=101
         Buffers: shared hit=107
         ->  Bitmap Index Scan on idx_pizza_orders_gin
➥              (cost=0.00..15.75 rows=563 width=0)
➥              (actual time=0.126..0.126 rows=563 loops=1)
               Index Cond: (pizza @> '{"type": "custom"}'::jsonb)
               Buffers: shared hit=6
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.445 ms
 Execution Time: 0.996 ms
(13 rows)
```

Overindexing refers to the opposite situation, when there are already too many indexes in place, which can do more harm than good. Considering how effective indexes are at improving query performance, it might be tempting to create an index on every table column. But that comes at a cost for the following reasons:

- An index is a data structure that consumes both storage and memory space. The more indexes you have, the more resources you need to allocate.
- Each index must be updated whenever you insert a new record into the associated table. It may also need to be updated when existing records are modified. If there are several indexes on a table, Postgres has to maintain all of them, which can lead to write amplification and slow down insert, update, and delete operations.
- There's also overhead during the query planning phase, as Postgres has to consider more indexes when deciding on the best execution plan.

So, have an indexing strategy in place. Refrain from adding a new index just to optimize yet another query. Instead, analyze execution plans for queries that are typically executed on a table, and see if it's possible to create a limited number of indexes that will make those queries performant and avoid the overindexing situation.

For example, we saw how the GIN index on the `pizza` column can optimize the search when the application filters orders by the `pizza @> '{"type": "custom"}'` condition. The good news is that the same index can also speed up queries that filter by other fields of the JSON objects stored in the `pizza` column. Here's how the execution plan looks for a query that calculates the total number of orders for a three-cheese pizza with a large crust:

```
EXPLAIN (analyze, buffers on)
SELECT count(*)
FROM pizzeria.order_items
WHERE pizza @> '{"size": "large"}' AND pizza @> '{"type": "three cheese"}';

                             QUERY PLAN
-------------------------------------------------------------------
 Aggregate  (cost=131.04..131.05 rows=1 width=8)
➥   (actual time=0.661..0.662 rows=1 loops=1)
   Buffers: shared hit=89
   ->  Bitmap Heap Scan on order_items
➥        (cost=22.54..130.67 rows=147 width=0)
➥        (actual time=0.328..0.642 rows=153 loops=1)
         Recheck Cond: ((pizza @> '{"size": "large"}'::jsonb)
➥        AND (pizza @> '{"type": "three cheese"}'::jsonb))
         Heap Blocks: exact=78
         Buffers: shared hit=89
         ->  Bitmap Index Scan on idx_pizza_orders_gin
➥              (cost=0.00..22.50 rows=147 width=0)
➥              (actual time=0.292..0.293 rows=153 loops=1)
               Index Cond: ((pizza @> '{"size": "large"}'::jsonb)
➥              AND (pizza @> '{"type": "three cheese"}'::jsonb))
               Buffers: shared hit=11
 Planning:
   Buffers: shared hit=1
 Planning Time: 0.336 ms
 Execution Time: 0.722 ms
(13 rows)
```

The database reuses the same index even though the condition has changed.

Overall, strive for balance when it comes to your indexing strategy. Too few or no indexes lead to underindexing, whereas too many indexes cause overindexing. The right balance lies somewhere in the middle, and it's unique to your use case.

A.3 Use connection pooling

Throughout the book, we use the psql tool to connect to a Postgres container and execute various queries. The connection is opened with a command similar to the following:

```
docker exec -it postgres psql -U postgres
```

This command first connects to the container named `postgres` (`docker exec -it postgres`) and then opens a database connection using psql, which is available in the container (`psql -U postgres`).

Because we don't specify the Postgres port number in the connection string, psql connects to port 5432 by default, which is equivalent to the following command:

```
docker exec -it postgres psql -U postgres -p 5432
```

Postgres starts a special process called the *postmaster*, which listens for incoming connections on port 5432. Once psql connects to this port, the postmaster spins up a new OS process to serve requests from that psql session. When the session is closed, Postgres terminates the process associated with the connection. Postgres doesn't keep or reuse this process for other clients that might connect later.

If your application executes queries concurrently, it needs to open a new database connection for each query, execute it, and then release (close) the connection. This also means Postgres has to spin up a new OS process for each concurrent query, execute it, and then terminate the process.

Although creating an OS process is a lightweight operation, especially on Unix, it still adds up to the total query execution time and requires both Postgres and the OS to do extra work. This overhead can be significantly reduced with connection pooling, which allows you to reuse already-opened database connections for future application queries.

Connection pooling can be implemented on both the client and server (database) side. *Client-side connection pooling* is handled via a language-specific library or framework that manages connections for the application. When the application needs to execute another query against Postgres, it doesn't connect to the database directly. Instead, it retrieves a connection from the pool, uses it to execute the query, and then releases the connection back to the pool. Once released, the pool doesn't close the connection but keeps it internally so it can be reused for the next application request. The maximum number of connections in the pool can be customized, giving you control over how many queries can be executed concurrently against the database.

Server-side (database-side) connection pooling is useful in scenarios where the database is accessed by multiple applications or microservices. Even if each application has its own client-side connection pool, collectively they can put extra pressure on the database server, which will need to maintain a number of connections (OS processes) equal to the sum of all connections across all client-side pools.

Postgres doesn't come with a built-in connection pooler; it doesn't need to because you can choose from several battle-tested options in the Postgres ecosystem. Examples include Pgpool, PgBouncer, and Odyssey. If you're using a managed Postgres service, it likely already includes a server-side pooler. So, you have plenty of options to ensure your database server isn't saturated by concurrent client requests.

> **TIP** If you're curious to see how Postgres spins up a dedicated OS process for each new client connection and how this can be optimized with a connection pool, check out the video "Database Connection Pooling: Why It Matters? Essential OS-Level Insights" listed in the book's repository: https://github.com/dmagda/just-use-postgres-book/blob/main/postgres_internals_videos.md.

Whether you use client-side, server-side, or both types of connection poolers, you'll make your queries more efficient by eliminating the need to open and close connections for individual client requests. Plus, you'll be able to control the number of queries that can be executed concurrently against the database, ensuring that CPU, memory, and other resource utilizations stay within the required boundaries.

A.4 Query only what you really need

If you see a query in your application logic that starts with `SELECT * FROM tableA WHERE...`, you should think twice about whether the `*` operator is really necessary. The operator instructs Postgres to retrieve the values of all `tableA` columns for rows satisfying the search criteria and add them to the query result.

A.4.1 General case

Although it's convenient to query all column values from the database and then decide on the application side what to use and what to discard, this might come at a cost. If a table has just a handful of columns with primitive fixed-size types such as `int`, `timestamp`, or `point`, there will be little difference between using the `*` operator and explicitly specifying a subset of the columns the application needs to get. However, if the table has dozens of columns or includes columns of variable-size types such as `text`, `vector`, `jsonb`, or `polygon`, it's reasonable to fetch only the values of those columns that the application actually needs.

Imagine that you have a table with 15 columns, but the application logic needs only 5 of them to fulfill a particular business operation. However, the application still fetches all columns using a `SELECT * FROM tableA WHERE...` statement, either because it's convenient or because it relies on an object-relational mapping (ORM) framework that

retrieves all columns by default. After reading the data from the database, the application ends up using only 5 columns and discarding the other 10. And the retrieval of those 10 discarded columns is far from free:

1. Postgres has to fetch the values of those 10 columns from the table structure and add their copies to the query result set. This needs to be done for every row satisfying the search criteria. If the query returns only 1 row, there will be 10 unnecessary column values. If the result set contains 10 rows, there will be 10 rows × 10 columns = 100 to-be-discarded column values.
2. The result set needs to be serialized and transferred over the network to the application. All the serialized data, including those 100 unnecessary column values (10 rows × 10 columns), needs to be divided into smaller network packets that travel through network routers from the database to the application server. If all the columns are of the `byte` type, then Postgres will be transferring just 100 bytes of redundant data. However, if any of those 10 columns is of a variable-size type, such as `text` or `jsonb`, the size of the redundant data can easily jump to kilobytes or megabytes, depending on the actual data stored in variable-size columns.
3. The application layer needs to receive those network packets, deserialize them back into a result set, and potentially transform the result into some object representation or structure used by your programming language or application framework. The application server ends up consuming compute and memory resources to deserialize and store those 100 unnecessary column values—all for nothing.

If this business operation is executed only once in a while, querying and discarding those 100 column values is unlikely to be noticeable. But what if the operation runs 100 times per second? Then the number of unnecessary column values grows to 100 query invocations × 10 rows × 10 columns = 10,000 values that are fetched, transferred, and discarded every second.

And that's just one business operation. If other operations adopt the same approach by querying all column values with `SELECT * FROM tableA...`, you're just burning CPU, memory, and network bandwidth across the stack for no good reason.

Thus, strive to develop the habit of always querying only what your application truly needs for a given business operation. If the operation requires just 5 columns from a table with 15 columns, explicitly fetch those columns using a statement like `SELECT col1, col2, col3, col4, col5 FROM tableA`. If you're using an ORM, keep an eye on the SQL queries it auto-generates, and tweak them as needed to ensure it retrieves only what's truly necessary.

A.4.2 The count(*) case

The `count(*)` function is an exception to the rule in Postgres, as the database applies a special optimization for this function call. When Postgres executes a query like `SELECT count(*) FROM tableA WHERE...`, it doesn't retrieve column values for the rows matching

the search criteria. Instead, it simply counts the number of rows produced by the query without fetching any column data. This is why we use `count(*)` throughout the book.

For example, take a look at the execution plan for the query that calculates the total number of pizza orders:

```
EXPLAIN (analyze)
SELECT count(*)
FROM pizzeria.order_items;

                                QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=91.69..91.70 rows=1 width=8)
 ➥    (actual time=0.721..0.723 rows=1 loops=1)
   ->  Index Only Scan using order_items_pkey on order_items
 ➥      (cost=0.28..84.35 rows=2938 width=0)
 ➥      (actual time=0.091..0.532 rows=2938 loops=1)
         Heap Fetches: 0
 Planning Time: 0.114 ms
 Execution Time: 0.771 ms
(5 rows)
```

The database uses the `Index Only Scan` access method and finds that there are 2,938 orders in the table (`rows=2938`). However, Postgres doesn't retrieve any column values because the size of each row is set to zero (`width=0`). This is how the `count(*)` optimization works in practice—the database skips reading any column values.

But if we change `count(*)` to `count(order_item_id)`, the database starts reading values of the `order_item_id` column while calculating the total number of orders:

```
EXPLAIN (analyze)
SELECT count(order_item_id)
FROM pizzeria.order_items;

                              QUERY PLAN
---------------------------------------------------------------------
 Aggregate  (cost=91.69..91.70 rows=1 width=8)
 ➥  (actual time=0.884..0.885 rows=1 loops=1)
   ->  Index Only Scan using order_items_pkey on order_items
 ➥       (cost=0.28..84.35 rows=2938 width=4)
 ➥       (actual time=0.031..0.534 rows=2938 loops=1)
         Heap Fetches: 0
 Planning Time: 0.270 ms
 Execution Time: 0.933 ms
(5 rows)
```

In this case, the size of each row is now set to 4 bytes (`width=4`) because the `order_item_id` column stores values of the `INT` type, which is 4 bytes in size.

A.5 Use the computational capabilities of the database

A database like Postgres is more than just a simple storage container where you put data in and query it later. It's a powerful computational platform at your disposal. As

we see throughout the book, Postgres can easily sort, group, aggregate, and perform other data manipulations using its built-in functions, common table expressions, window functions, and more. You can even run your own business logic in the database by implementing custom functions, stored procedures, and triggers.

So, study and take advantage of the computational capabilities Postgres offers. If you need to sort a query result, use the `ORDER BY` clause instead of sorting the fetched data on the application side. If you need to group data based on certain criteria, you likely don't require custom application logic; the `GROUP BY` clause will probably do the job. And if you need to perform a compute- or data-intensive business operation that heavily relies on data stored in Postgres, consider implementing a database function (or stored procedure) that runs entirely in the database. The more you can offload to the database, the less you have to handle on the application side, potentially making your logic cleaner and resource utilization across the stack more efficient.

appendix B When not to use Postgres

Like any other technology, Postgres has its strengths and weaknesses. There are use cases where Postgres shines, and at the same time, there are workloads where it's not the best option. Even though the book is titled *Just Use Postgres!* and many of us in the Postgres community champion its broader adoption, we all understand that it's neither a silver bullet nor a Swiss Army knife that can replace all other database technologies.

However, if you expect me to walk you through a list of Postgres weaknesses in this appendix, I may disappoint you, my friend: it's not going to happen. The main reason is that even if I say you should refrain from using Postgres for use case *X*, that advice may no longer be accurate a year or two after this book is published. Postgres and its ecosystem are evolving so rapidly that what's unsupported today might be easily handled by the database in the near future.

But if we put technical and business use cases aside, there are still three reasons I personally consider before deciding whether to use Postgres. Let me share them with you, as they may help you make a more weighted and less biased decision.

I personally do not use Postgres in the following instances:

1. *Another database simply suits a use case better than Postgres, and it would be hard or impossible to adopt Postgres instead.* There's no need to overengineer and complicate your and your team's lives by trying to make Postgres fit when another database supports the use case easily.

2. *An application already uses another database, and there's no real need to replace it with Postgres.* There's no point in using Postgres just for the sake of using Postgres. If another database does its job well and nobody has any concerns about it, don't break what already works. Even if that database is a paid product and the company spends a lump sum on it, let the company pay, especially if it's comfortable with the cost.
3. *The majority of the team is much more experienced with another database.* If both Postgres and the other database are equally well suited for the use case, it's reasonable to tap into the team's existing expertise. By doing that, you'll likely build and ship faster, make fewer mistakes, and be better prepared to handle production problems. Plus, it might even be a good opportunity for you to learn something new and broaden your experience in the database domain.

Even though Postgres is my favorite and default database, I take these reasons seriously before bringing it into a project. For example, I once joined a project where the team was successfully using a document database. It worked fine for the application, and the whole team was very comfortable with it. With my help, we could have easily replaced that database with Postgres, but I didn't even suggest doing so. Instead, I decided to study that database a bit more deeply and ended up helping the team optimize its usage dramatically and cutting infrastructure costs by a factor of five!

With that, this book is over, but I hope your journey with Postgres continues. If you've never used it on a project before, start using it. If you've already used it for years, consider trying it for another use case you learned about in this book. Keep investing your time in Postgres, and it should pay off well, considering how ubiquitous it has become and how quickly its adoption is growing.

afterword

I started using PostgreSQL in 2008 while working on a logistics enterprise application. That was the first time I used an open source database system, as previously I had been using only Oracle and SQL Server, which were very popular options at the time.

However, right from the start, I was surprised by the abundance of advanced features that Postgres 8.3 offered at the time, such as arrays, enum types, full-text search and GIN indexes, geometric types and GiST indexes, multivalue inserts, and NOWAIT, to name a few.

At the time, I was operating in the outsourcing business, and for this reason, I got the chance to work on many Java-based enterprise applications. They all had one thing in common: they all used Postgres, and for a very good reason. Not only did Postgres offer advanced features that you couldn't find in commercial database systems, but every new release gave us more reasons to consider using it for our next big project.

In 2011, we were working on a project that needed to process some JSON documents from external suppliers. While we were parsing the JSON documents to store the data in relational tables, we also needed to store the original JSON document to backtrack a given value that the external provider was questioning.

Because we needed to search through those JSON files during the audit process, we chose MongoDB to store the JSON documents (at the time, relational database systems didn't provide this functionality). Although some authors promote the idea of polyglot persistence, I can tell you from this experience that there are significant downsides to using multiple database systems. Adding a new database system during development is easy, but maintaining it in production is hard because you need to consider monitoring, backups, upgrades, and onboarding for new hires and carefully assess which features affect performance.

Luckily, since version 9.4, PostgreSQL has been offering native support for JSON, so many of the projects that previously required a NoSQL database could now "Just Use Postgres." And reducing the number of systems that you are running in production has many advantages, too. The fewer the moving parts, the easier it is to keep the system going. "Keep it simple" is not just for development. Actually, it's even more valuable for the system architecture. By using the advanced features that PostgreSQL offers, we can address a plethora of real-life requirements using just one relational database.

Apart from JSON, Postgres has many other custom types, such as `ARRAY`, `Range`, `Inet` (IP addresses), and `Monetary`, that are not commonly supported by other relational database systems. Using dedicated native types is preferred whenever your application requires storing nonstandard data types, so in 2017 I created Hypersistence Utils, an OSS project that allows you to map all the aforementioned PostgreSQL column types to JPA entity attributes.

In recent years, while running High-Performance Java Persistence and High-Performance SQL training courses, I asked my students what databases they were using so I could adjust the agenda to their needs. And, unsurprisingly, more and more projects started using PostgreSQL almost exclusively.

I got excited when Denis told me that he was writing a book about PostgreSQL because Denis is a subject matter expert who previously worked for YugabyteDB, a major PostgreSQL-compatible distributed database. As an application developer, I consider this book a must-read because database and SQL knowledge are often overlooked by backend and frontend developers, who prefer to focus on improving their programming language, application framework, and design pattern skills.

Covering topics such as modern SQL; JSON; full-text search; geospatial data; time series; and `pgvector`, which is a popular option for storing embeddings in RAG and GenAI applications, *Just Use Postgres!* is the right book to help you become familiar with all these topics or improve your existing knowledge. Not only is the book easy to read, with an impeccable writing style, but I was pleasantly surprised to learn many new things.

I hope you also enjoy reading the book and will now be able to design and build systems that can use Postgres to its full extent!

—VLAD MIHALCEA
AUTHOR, High-Performance Java Persistence

index

Symbols

<-> (followed by) operator 167, 172, 179, 181, 259
<N> operator 181
<@ operator 296
=> operator 49
@> containment operator 156, 160, 161, 162
|| operator 172
| (logical OR) operator 177, 179
& (AND) operator 179, 181
.key accessor operator 144
& (logical AND) operator 177
@? match operator 156, 160
@@ match operator 156, 160, 178–179
! (NOT) operator 179, 180
? operator 162
* operator 16, 365
#- operator 149

A

accounts table 34, 207
ANN (approximate nearest neighbor) search 234
a_point geometry 309

B

BETWEEN operator 100
BIGSERIAL data type 26, 336
Bitmap Heap Scan 106, 118, 157, 158, 161, 329
Bitmap Index Scan 106, 110, 118, 119, 152, 156, 157, 158, 161, 194, 196, 329
BRIN (block range index) 107, 284–290, 321
B-tree indexes 90, 281–284
  single-column 99–103
  using expression index with 151–154
Buffers metric 98
byte type 366

C

catalog table 25, 26, 28, 63
centroid 234, 321
Check constraint 25, 30
chunking 220, 263
client-side connection pooling 364
closed polygons 293
COMMIT operation 37, 270, 338
composite indexes 107–116
  caveats of 113–116
  considering additional single-column index 108–111
  creating 111–113
connection pooling 364–365
connectors and foreign data wrappers 210
constraints 29–32
context object 249

continuous aggregates 274–279
  creating and using 274–277
  refreshing 277–279
cosine distance 227
count(*) aggregate 102
CTEs (common table expressions) 66, 68–76, 230, 311
  modifying data with 73–76
  selecting data with 69
  using multiple in query 70–73
current schema 24

D

data, querying and manipulating 27–29
databases 19
  creating 20–26
  creating structure 19–26
  using computational capabilities of 368
database users 61
data integrity 29–36
  constraints 29–32
  foreign keys 32–36
daterange type 322
DDL (data definition language) 212
DELETE operation 28, 53, 69, 73
dirty read phenomenon 39
DML (data manipulation language) 27, 212
Docker
  starting Postgres in 6–9
  starting Postgres with TimescaleDB 258–259

E

embedding indexing 232–245
  HNSW indexes 240–245
  IVFFlat indexes 234–240
embedding models 216–218, 220
embeddings, generating 220–226
  for movies 221–223
  loading final dataset into Postgres 223–226
english_stem dictionary 167, 169
Exclusion constraint 30
expression indexes 122
extensibility of Postgres 202

F

FATAL error 62
FDW (foreign data wrapper) 210
filtering, using to_tsquery for advanced filtering 179–182
foreign keys 29, 32– 36
full-text search 164
  basics of 165–170
  configurations 168–170
  highlighting search results 188
  lexemes, indexing 191–196
  performing 177–182
  ranking search results 182–188
  tokenization and normalization 166
functional indexes 122
functions 44–54
  overview of 46–51
  triggers 51–54

G

general-purpose database 4
generative AI
  embedding indexing 232–245
  embeddings 220–226
  implementing RAG 245–251
  pgvector extension 218–220
  vector similarity search 226–232
geography data type 296, 297, 322
geometry data type 296, 297, 303, 304
geom geometry object 309
geospatial data 292–297
  built-in capabilities 295
  indexing 321–330
  PostGIS 296–330
  querying 308–320
  visualizing 306
GIN (generalized inverted index) 131, 173
  data structure 92
  indexing lexemes with 192
  using indexes 154–162
GISs (geographic information systems) 294
GiST (generalized search tree) 173, 321
  indexing lexemes with 194–196
  structure of 322–324
  using 325–330
GPS (Global Positioning System) 297

H

hallucination, defined 229
hash indexes, single-column indexes 103–106

highlighting search results 188
HNSW indexes 240–245
  creating and using 241–243
  improving index recall during search 243–245
hypertables 262–266
  automatically dropping data with data retention policy 266
hypopg extension 211

I

implicit transactions 37
indexes 89, 90
  composite indexes 107–116
  covering indexes 117–119
  EXPLAIN statement 95–99
  functional and expression indexes 122
  multiplayer game dataset 93
  overview of index types 92–93
  partial indexes 119–122
  popularity of 90–91
  single-column indexes 99–106
  when to use 360–364
indexing
  JSON data 150–162
  lexemes 191–196
  message queues 347
Index Only Scan 119
Index Scan 237, 243, 328
inet type 322
IN operator 104, 105
INSERT operation 27, 53, 69, 73
int4range type 322
int8 type 333
interpolate function 273
int type 365
IVFFlat indexes 234–240
  creating and using 235–237
  improving index recall during search 237–240

J

joins 42–44
jsonb data type 131, 132, 333, 336, 354, 365–366
jsonb type 133, 354, 365, 366
json data type 131, 333
JSON (JavaScript Object Notation)
  data indexing 150–162
  in Postgres 134–136
  modifying data 147–150
  querying JSON data 136–147
  storing JSON data 130–131
JSON path expressions 143–147
JSON type 336

K

kafka_fdw extension 210
keys, using ? operator to check for presence of 140

L

leading column, defined 107
LEFT JOIN 43
lexemes, indexing 191–196
  using GIN indexes 192
  using GiST indexes 194–196
lexemes column 167, 179, 184, 186, 189, 222
line segments 294, 318–320
LineString geometric subtype 296, 305, 317
LISTEN/NOTIFY capabilities 54
LLM (large language model) 213–214
  interacting with 247
  retrieving context for 248
lseg type 295

M

manipulating data 27–29
materialized views 56
message queues
  building custom 335–339
  custom queues 339–343
  implementation considerations 346–350
  LISTEN and NOTIFY commands 343–346
  Postgres as 333–335, 352–357
modern SQL 66–67
multicolumn indexes 107
MultiPolygon type 305
multitenant SaaS application 20
MVCC (multiversion concurrency control) 39
mysql_fdw extension 210

N

nearest neighbor search 226
Nested Loop join 328

normalization 165, 166, 184
NOT NULL constraint 25, 31, 134, 176
NUMERIC type 29

O

objects, comparing with @> operator 141
OGC (Open Geospatial Consortium) 296
O($\log_b N$) algorithmic complexity 91
OLTP (online transaction processing) 3
OMDB (Open Media Database) 174
ON operator 42
optimization tips
  connection pooling 364–365
  querying only what is needed 365–367
  using computational capabilities of database 368
oracle_fdw extension 210
ORM (object-relational mapping) 15, 366
  frameworks 67
osm2pgsql tool 300, 301, 302, 327
OSM (OpenStreetMap) 300–305
  exploring tables with points 302
  exploring tables with polygons 304
  exploring tables with ways 303
overindexing 361, 362

P

parquet_s3_fdw extension 210
partial indexes 119–122
partitioning, message queues 348–349
partitions 255, 263
pg_ai extension 209, 223, 251
pgaudit extension 211
PgCompute client-side library 210
pg_cron extension 58, 211, 257, 349, 350
pgcrypto extension 205, 207
  disabling 208
  enabling 206
  using 207
pg_database system catalog 21
pg_duckdb extension 210
pg_extension view 205, 206, 208
pgmq extension 210, 350–352
pgmq (Postgres message queue) extension, using 352–357
  message visibility timeouts 354–357
pg_partman extension 211, 257, 349
pgRouting extension 320
pg_search extension 197
pg_stat_statements extension 211
pg_trgm extension 178, 322
pgvector extension 7, 209, 218–223, 227, 233, 245, 249, 251
pgvectorscale extension 209, 233
PL/Java extension 210
plpgsql extension 205
PL/Python extension 210
PL/Rust extension 210
PLV8 extension 210
Point geometric subtype 296, 313, 317
points 302, 308–314
  creating from coordinates 313
  distance between 312
  transforming and retrieving point coordinates 310
Polygon geometric subtype 296, 304–305, 314–318, 365
PostGIS extension 210, 296, 321
  starting Postgres with 297–300
Postgres 3
  as message queue 332–339, 350–357
  extensibility of 202
  extensions 202–209
  extensions for developers 209–211
  for generative AI 213–218
  full-text search 171–177
  generating mock data 11–15
  geospatial data 292–297
  JSON 129, 134–135
  optimizing queries 358
  pgvector extension 218–220
  popularity of 4
  postgres_fdw extension 210
  reasons for using 6
  running basic queries 15–17
  starting in Docker 6–9
  starting with PostGIS 297–300
  starting with TimescaleDB 258
  time series 254–257, 262–265
  when not to use 369
Postgres beyond relational extensions 209
Postgres-compatible solutions 211
PostgreSQL
  Anonymizer extension 211

TimescaleDB 258–259
postgres role 59
postmaster, defined 364
Primary key 29
constraint 134
programming and procedural languages 210
PUBLIC role 61–62
public schema 19, 23, 25

Q

quadrants, defined 321
queries, using plainto_tsquery for simple queries 177
query and performance optimization 210
querying
data 27–29
only what is needed 365–367
querying geospatial data 308–320
line segments 318–320
points 308–314
polygons 314–318
querying JSON data 136–147
comparing objects with @> operator 141
extracting fields with -> and ->> operators 137–140
using ? operator to check for presence of key 140
using JSON path expressions 143–147
queues
custom 339–343
message queues 346–350

R

RAG (retrieval-augmented generation) 214, 245–251
interacting with LLM 247
preparing environment for prototype 247
retrieving context for LLM 248
using to answer questions 249–251
rank, defined 189
ranking search results 182–188
RDBMS (relational database management system) 18
creating database structure 19–26
data integrity 29–36
functions and triggers 44–54
joins 42–44
querying and manipulating data 27–29
roles and access control 59–64
transactions 36–42
views 55–59
recursive queries 76–82
querying hierarchical data 78–80
using arguments in recursion 80–82
redis_fdw extension 210
relational database, defined 3
roles, overview 59–64

S

SaaS (software-as-a-service) 19
schemas, creating 22–25
search_rank 184, 185
SELECT queries 15, 39, 42
semantic search 226
Seq Scan 100, 102, 104, 120, 151, 152, 153, 237
SERIAL data type 25, 34, 336
server-side connection pooling 365
simple queries 177
single-column indexes 99–106
B-tree 99–103
hash 103–106
spatial data 292
SP-GiST (space-partitioned generalized search tree) 321
sqlite_fdw extension 210
SQL (Structured Query Language) 3, 18, 65
CTEs (common table expressions) 68–76
recursive queries 76–82
window functions 82–87
SRID (spatial reference system identifier) 296
storing and indexing 165
storing JSON data 130–131

T

table partitioning feature 255
tables
creating 25–26
with points 302
with polygons 304
with ways 303
text data type 131
text search, preparing data for 171–177
text type 164, 365–366

TimescaleDB
  checking available version 259
  extension 210
  starting Postgres with 258
time series 253
  continuous aggregates 274–279
  hypertables 262–266
  startup 258
time-series data
  analyzing 266–274
  loading 260–262
  optimizing queries 279–290
  Postgres and 254–257
timestamp type 365
timestamptz data type 254, 270, 333
tokenization 165–166
tools and utilities 211
to_srid spatial reference system 309
transactions 36–42
  explicit 37–39
  implicit 36
  multiversion concurrency control 39–42
triggers 44–54
  functions 46–51
  overview of 51–54
trigrams, defined 178
tsquery data type 177, 181, 187
tsvector data type 171, 177, 179, 184, 222, 322

U

UI (user interface) 177
underindexing 361
Unique constraint 29, 104
USING btree clause 46, 101, 153
USING clause 92
USING GIN clause 160
USING gist clause 327
USING hash clause 104
UUID type 37, 336

V

vacuum, defined 39
vector database 221
vector data type 218, 222, 365
vector embedding 216
vector similarity search 226–232
  changing search phrase for better results 230–232
  cosine distance for 227
views 55–59
visibility timeouts 354–357
visualizing geospatial data 306

W

weights 185–187
WKB (well-known binary) 303
WKT (well-known text) 303

Some essential Postgres extensions for developers *(continued)*

Category	Extensions	Description
Connectors and Foreign Data Wrappers (Extensions that let Postgres connect to external data sources and query remote data transparently directly from Postgres)	`file_fdw`	A foreign data wrapper (FDW) that allows querying data from the file system.

	`postgres_fdw`
	`mysql_fdw`
	`oracle_fdw`
	`sqlite_fdw`
		Foreign data wrappers that let Postgres connect and query other SQL databases, such as a remote Postgres server, MySQL, Oracle, or SQLite.

	`redis_fdw, parquet_s3_fdw` `kafka_fdw`	Extensions that enable Postgres to retrieve data from non-SQL data sources such as Redis, S3, or Kafka.
Query and Performance Optimization (Extensions that help you analyze query execution statistics and improve performance when necessary)	`pg_stat_statements`	Helps to track the planning and execution statistics of all SQL statements executed by Postgres.
	`auto_explain`	Automatically logs execution plans for slow queries, making it easier to diagnose performance issues without manually running `EXPLAIN ANALYZE`.
	`hypopg`	Adds support for hypothetical indexes, allowing us to test different indexing strategies without creating real indexes.
Tools and Utilities (Extensions that simplify common Postgres tasks, automate operations, and enhance database usability)	`pg_cron`	A simple cron-based scheduler that runs within Postgres, helping automate routine tasks associated with the database.
	`PostgreSQL Anonymizer`	Lets you anonymize or mask personal and sensitive data using techniques like dynamic masking and pseudonymization.
	`pgaudit`	Provides detailed session and object-level audit logging through Postgres's standard logging facility.
	`pg_partman`	Simplifies the creation and management of both time-based and number-based table partitions.

SOFTWARE DEVELOPMENT/DATABASES

Just Use Postgres!

Denis Magda • Foreword by Josh Long • Afterword by Vlad Mihalcea

You know that PostgreSQL is a fast, reliable, SQL-compliant RDBMS. You may not know that it's also great for geospatial systems, time series, full-text search, JSON documents, AI vector embeddings, and many other specialty database functions. For almost any data task you can imagine, you can use Postgres.

Just Use Postgres! covers recipes for using Postgres in dozens of applications normally reserved for single-purpose databases. Written for busy application developers, each chapter explores a different use case illuminating the breadth and depth of Postgres's capabilities. Along the way, you'll also meet an incredible ecosystem of Postgres extensions like pgvector, PostGIS, pgmq, and TimescaleDB. You'll be amazed at everything you can accomplish with Postgres!

What's Inside

- Generative AI, geospatial, and time-series applications
- Modern SQL including window functions and CTEs
- Full-text search and JSON
- B-trees, GIN, GiST, HNSW, and more

For application developers, software engineers, and architects who know the basics of SQL.

Denis Magda is a recognized Postgres expert and software engineer who worked on Java at Sun Microsystems and Oracle before focusing on databases and large-scale distributed systems.

For print book owners, all digital formats are free:
https://www.manning.com/freebook

"I was pleasantly surprised to learn many new things from this book."

—From the Afterword by Vlad Mihalcea

"An excellent guide covering everything from basics to cutting-edge features."

—Dave Cramer
PostgreSQL JDBC Maintainer

"Pleasant, easy to read with tonnes of great code."

—Mike McQuillan
McQTech Ltd

"Well-organized and easy to search."

—Edward Pollack
Microsoft Data Platform MVP

"The missing guide to understanding and using Postgres."

—Mehboob Alam
POSTGRESNX, Inc.

FREE eBook
see first page

ISBN-13: 978-1-63343-569-8

MANNING