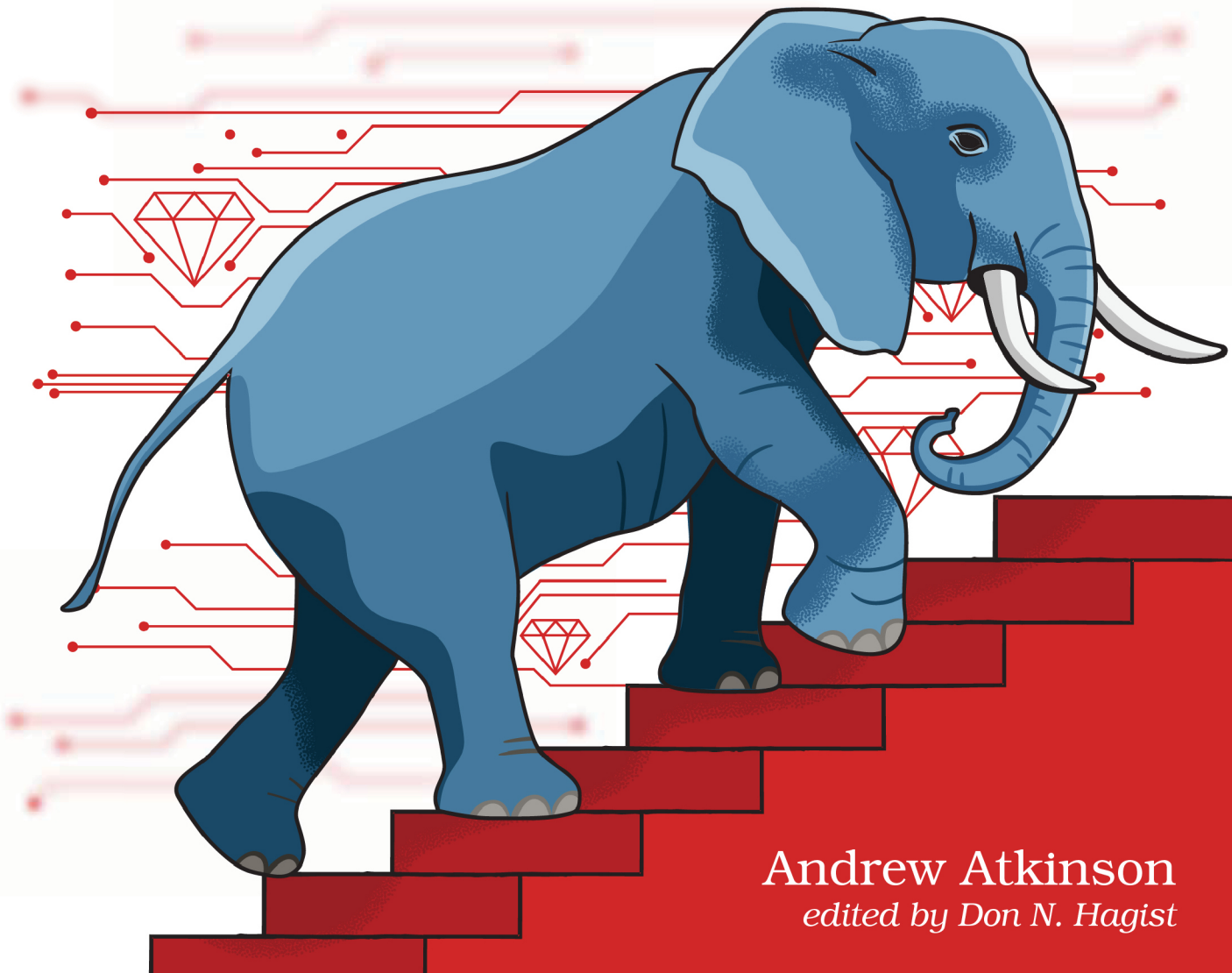


The
Pragmatic
Programmers

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable
Database Applications



Andrew Atkinson
edited by Don N. Hagist



Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at pragprog.com/my_account. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at pragprog.com/titles/aapsql/errata, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

The Pragmatic Bookshelf

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications

Andrew Atkinson

The Pragmatic Bookshelf

Dallas, Texas



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit <https://pragprog.com>.

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2023 The Pragmatic Programmers, LLC.

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

ISBN-13: 979-8-88865-038-7

Encoded using the finest acid-free high-entropy binary digits.

Book version: B1.0—August 30, 2023

Contents

Change History	ix
Preface	xi

Part I — Getting Started

1. An App to Get You Started	3
What Is Rideshare?	3
Active Record Schema Management Refresher	4
Installing Rideshare	6
Working With PostgreSQL Locally	10
Learning PostgreSQL Terminology	10
Learning SQL Terminology	11
Ruby on Rails Terminology	12
Conventions Used In This Book	13
You're Ready	15

Part II — Design and Build

2. Administration Basics	19
Learning Meta Commands In psql With Rideshare	20
Modifying Your PostgreSQL Config File	22
Getting Started With Query Observability	23
Glancing At Current Lock Behavior	25
Generating Fake Data For Experiments	26
Rolling Back Schema Modifications	28
Exploring and Experimenting Safely In Production	29
3. Building a Performance Testing Database	33
Downloading Production Data Dumps	34

Replacement Values That Are Statistically Similar	37
Tracking Columns With Sensitive Information	39
Comparing Direct Updates and Table Copying Strategies	39
Starting An Email Scrubber Function	40
Implementing the Scrub Email Function	42
Understanding Table Copying Trade-Offs	45
Speeding Up Inserts For Table Copying	47
Using Direct Updates For Text Replacement	53
Performing Database Maintenance	54
Performing Modifications In Batches	55
What's Next For Your Performance Database	57
4. Data Correctness and Consistency	59
Multiple Column Uniqueness	60
Fixing Constraint Violations	62
Enforcing Relationships With Foreign Keys	63
The Versatile Check Constraint	64
Deferring Constraint Checks	67
Preventing Overlaps With An Exclusion Constraint	68
Creating Active Record Custom Validators	69
Significant Casing And Unique Constraints	72
Storing Transformations In Generated Columns	73
Constraining Values With Database Enums	74
Sharing Domains Between Tables	77
Automating Consistency Checks In Development	79
 Part III — Operate and Grow	
5. Modifying Busy Databases Without Downtime	83
Identifying Dangerous Migrations	85
Learning From Unsafe Migrations	85
Learning To Use CONCURRENTLY By Default	88
Adopting A Migration Safety Check Process	89
Exploring Strong Migrations Features	89
Locking, Blocking, and Concurrency Refresher	92
Preventing Excessive Queueing With A Lock Timeout	95
Exploring Lock Type Queues	96
Setting Statement Timeout	97
Avoiding Schema Cache Errors	98
Backfilling Large Tables Without Downtime	98

Backfilling And Double Writing	99
Separating Reads and Writes for Backfills	99
Specialized Tables For Backfills	100
Practicing Backfilling Techniques	101
Wrapping Up	102
6. Optimizing Active Record	105
Preferring Active Record Over SQL	105
Query Logs To Connect SQL to App Code	106
Common Active Record Problems	107
Use Eager Loading To Reduce Queries	110
Eager Loading With .includes()	110
Prefer Strict Loading Over Lazy Loading	111
Optimizing Individual Active Record Queries	112
Save A SELECT With A RETURNING	113
Bounding Query Results Using LIMIT	114
Advanced Query Support In Active Record	115
Using Common Table Expressions (CTE)	116
Introducing Database Views for Rideshare	118
Creating the Search Result Model With Scenic	120
Improving Performance With Materialized Views	121
Reducing Queries With Active Record Caches	122
Prepared Statements With Active Record	123
Eliminating Slow Count Queries With Counter Caches	124
Performing Aggregations In the Database	126
Reduced Object Allocations With SQL In Active Record	127
Wrapping Up	128
7. Improving Query Performance	129
Logging Slow Queries With Active Support Notifications	130
Capture Query Statistics In Your Database	131
Rideshare Query Statistics	132
Introducing PgHero As a Performance Dashboard	133
Analyzing Query Execution Plans	135
Finding Missing Indexes	138
Logging Slow Queries	139
Automatically Gathering Execution Plans	140
Performing Maintenance First	141
Interpreting Index Scan Execution Plan Info	142
EXPLAIN Scan Nodes and Bitmap Scans	143
Adding Query Boundaries With Filtering and LIMIT	144

	Performing Fast COUNT() Queries	145
	Using Code and SQL Analysis Tools	147
	Wrapping Up	147
8.	Optimized Indexes For Fast Retrieval	149
	Generating Data for Experiments	150
	Transforming Ruby to SQL	153
	Why Are My Indexes Not Being Used?	154
	Using Single Column And Multiple Column Indexes	154
	Understanding Index Column Ordering	155
	Indexing Boolean Columns	156
	Transform Values with an Expression Index	157
	Using GIN Indexes with JSON	158
	Using GIN Indexes for Full Text Search	162
	Using Partial Indexes	165
	Using BRIN Indexes	167
	Using Indexes In Less Common Ways	170
	Using Covering Indexes	171
9.	High Impact Database Maintenance	175
	Basics of Autovacuum	176
	Tuning Autovacuum Parameters	177
	Rebuilding Indexes Without Downtime	179
	Running Manual Vacuums	180
	Simulating Bloat and Understanding Impact	181
	Removing Unused Indexes	183
	Removing Duplicate And Overlapping Indexes	184
	Removing Indexes On Insert Only Tables	186
	Scheduling Jobs Using pg_cron	186
	Monitoring pg_cron Scheduled Jobs	188
	Conducting Maintenance Tune-Ups	189
10.	Handling Errors From Increased Concurrency	191
	Monitoring Database Connections	192
	Managing Idle Connections	194
	Setting Active Record Pool Size	195
	Using the Active Record Connection Pool	195
	Running Out Of Connections	196
	Working With PgBouncer	199
	Choosing A PgBouncer Pooling Mode	201
	Identifying Connection Errors and Problems	202

More Lock Monitoring With pg_locks	204
Monitoring Row Locks	204
Finding Lock Conflicts	205
Using PgBadger For Lock Analysis	206
Active Record Optimistic Locking	206
Using Advisory Locks	207
Wrapping Up	208

Part IV — Optimize and Scale

11.	Scalability of Common Features	211
12.	Working With Bulk Data	213
13.	Scaling With Replication And Sharding	215
14.	Boosting Performance With Partitioning	217

Part V — Advanced Uses

15.	Advanced Usage and What's Next	221
A1.	Installation Guides	223
A2.	psql Client	225
A3.	Getting Help	227

Change History

The book you're reading is in beta. This means that we update it frequently. Here is the list of the major changes that have been made at each beta release of the book, with the most recent change first.

B1.0: August 30, 2023

- Initial beta release.

Preface

If you're looking to improve your skills with PostgreSQL for Ruby on Rails you've come to the right place. In this book you'll work on exercises from practical problems that will help you grow your career.

Maybe your application has grown in popularity and you've got performance problems. Maybe your database has ballooned in size and you aren't sure how to manage the growth. In the market, the traditional Database Administrator role is declining in popularity while there are more database-backed web applications than ever. Organizations from startups to huge companies choose PostgreSQL as a mission critical database to build their businesses on. Operator challenges, especially under high growth and high query volume, can put a lot of pressure on the engineering team. Why not be the team member who takes on these challenges?

Get ready to develop useful skills for operating PostgreSQL that you can immediately put to use. You'll use the latest versions of PostgreSQL and Ruby on Rails. You'll work on making your applications and databases faster, more reliable, and more resilient. As you work towards mastery of high performance operations, you'll raise the bar for operational excellence for your PostgreSQL database and your engineering team.

As you go through the book's exercises, you'll work with a real Rails application and database schema that you can see and continue to evolve as a test lab for your skill development. You'll populate millions of rows of data to work with, to help simulate your production workload. In addition to the core functionality of PostgreSQL and Ruby on Rails, you'll be working with 50 Ruby gems and PostgreSQL extensions.

Who Is This Book For?

This book is for developers looking to improve their skills with PostgreSQL and Ruby on Rails. Whether you're working on consumer scale Internet applications or enterprise B2B Software as a Service (SaaS), scaling PostgreSQL

and Rails application codebases is mission critical for the success of your business. Your team members expect you to both build with and operate these technologies, managing huge data growth, shifting business priorities, and operating reliable and cost efficient infrastructure.

If your job responsibilities or career aspirations include any of the following descriptions, you've come to the right place:

- Ruby on Rails Application Developers improving their PostgreSQL knowledge and skills
- PostgreSQL Database Administrators (DBAs) learning about Ruby on Rails and Active Record
- Infrastructure Engineers developing SQL knowledge and PostgreSQL knowledge for Replication, Parameters, Maintenance, and Partitioning
- Database Reliability Engineers (DBRE) implementing Sharding, Replication, Partitioning, and Parameter Optimization
- SQL developers experienced with other database engines looking to learn PostgreSQL
- Database engineers working with MySQL looking to learn PostgreSQL
- Web Application Developers using other full-stack web frameworks like Laravel¹ or Django,² looking to learn Ruby on Rails and Active Record
- Data Engineers working with OLAP databases interested in Replication and Change Data Capture (CDC) with PostgreSQL and OLTP use cases
- Engineers looking to get a promotion to senior levels with improved database skills

Here you'll work exclusively with the open source community distribution of PostgreSQL. Sticking with locally installed community distribution of PostgreSQL allows readers to gain confidence by developing and testing locally. After getting in some practice, readers can take their skills into production.

The focus is on Transactional workloads, also called Online Transaction Processing or (OLTP), and not Analytical workloads. OLTP workloads are the types of queries you'd see in a web application that is user facing.

What's Not Covered in This Book?

This book will help you a lot but it doesn't cover everything. It is not an introduction to PostgreSQL or Ruby on Rails. Books like *PostgreSQL: Up and Running*³ are intended to help get readers started.

1. <https://laravel.com>

2. <https://www.djangoproject.com>

3. <https://www.oreilly.com/library/view/postgresql-up-and/9781449326326/>

Pragmatic Programmers has lots of excellent books on Ruby on Rails. *Agile Web Development with Rails* ⁴ would be a great book to go through first if you're new to Ruby on Rails. This book will help you build a good foundation with Ruby on Rails and Active Record. For readers that wish to strengthen their skills with the Ruby programming language, considering purchasing and reading *Programming Ruby 3.2 (5th Edition)*.⁵

Internals of PostgreSQL are not covered beyond some basics as needed for a chapter. Multiversion Concurrency Control (MVCC) and Isolation Levels for example are introduced with some basic information and readers are direct to the PostgreSQL documentation for more information.

User Administration, Roles, and Privileges are not covered. The GRANT command and Security concepts like Row Level Security (RLS) and Policies are not covered.

Readers will set up multiple PostgreSQL instances, however High Availability (HA) concepts like replication with Availability Zones (AZ) and Regions, or Disaster Recovery (DR) implementations are outside the scope of this book.

For concepts like Recovery Time Objectives (RTO) and Recovery Point Objectives (RPO), consider the book *Database Reliability Engineering*⁶ which focuses on building reliable database systems.

Database Backups are critical. Your organization must collect Snapshots automatically and be able to restore a database server instance to a previous backup if disaster strikes. However, backups and restores are also outside the scope of this book beyond some basics.

In the broader PostgreSQL ecosystem, there are many options beyond what is supported in the open source community distribution where you'll run PostgreSQL on your local server instance. There are PostgreSQL forks, PostgreSQL compatible databases, and PostgreSQL extensions that make very significant modifications to how PostgreSQL operates. For example, the *Citus 11.1 open source release*⁷ modifies PostgreSQL as an extension. Although extensions like Citus do offer portions of their offerings as open source software, in general forks and different databases engines with compatibility are outside the scope of this book.

4. <https://pragprog.com/titles/rails7/agile-web-development-with-rails-7/>

5. <https://pragprog.com/titles/ruby5/programming-ruby-3-2-5th-edition/>

6. <https://www.oreilly.com/library/view/database-reliability-engineering/9781491925935/>

7. <https://www.postgresql.org/about/news/announcing-citus-111-open-source-release-2511/>

Ruby on Rails Skills Are In Demand

In the piece called “Big Transitions in the Tech Industry” covering Hired’s *2023 State of Software Engineers Survey*,⁸ Ruby on Rails was the most sought-after skill.

Ruby on Rails surfaced as the most in-demand skill for software engineering roles, creating 1.64x more interview requests for the developers proficient in it.

Ruby on Rails is a great way to build PostgreSQL database applications. Ruby on Rails has a Rails Guides page that is dedicated to showing how to use Active Record with PostgreSQL.⁹ Some of the capabilities are exclusive to PostgreSQL.

In the earlier days of Ruby on Rails and the Active Record ORM there was an emphasis on portability, where web applications might migrate from one database management system to another. Portability has been deemphasized over the years. Community blog posts and conference talks regularly feature capabilities that are exclusive to PostgreSQL.

Rails continues to add support for PostgreSQL capabilities to each new release. Generated Columns from PostgreSQL 12 were added as Virtual Columns¹⁰ to Active Record.

In Rails 7.1 support for Common Table Expressions (CTE) queries were added to Active Record.

These are exclusive or advanced database features that weren’t supported in the past, but have been added to the core framework. These are the kinds of topics you’ll find as you read on.

With native support for Multiple Databases, Ruby on Rails and Active Record can be used with multiple PostgreSQL databases working together. Multiple databases can be paired up using physical replication for Read and Write splitting, or multiple writeable databases can be used from the application for application level sharding. You’ll see and configure both of these.

We know skills with Ruby on Rails are in demand. What about PostgreSQL?

PostgreSQL Is A Popular Award Winner

You’ve made a great choice to invest your time learning PostgreSQL.

8. <https://hired.com/state-of-software-engineers/2023/>

9. https://guides.rubyonrails.org/active_record_postgresql.html

10. <https://blog.saeloun.com/2022/01/25/rails-7-postgres-support-for-generated-columns.html>

For deployed Ruby on Rails applications, PostgreSQL is the #1 most popular database according to the 2022 Ruby on Rails Community Survey with data from 2009 through 2022.¹¹

Besides the Rails Hosting Survey, the *DB-Engines Ranking*¹² ranks the most popular databases in the world. PostgreSQL has been a three-time #1 Winner there in 2017, 2018, and 2020.

In 2021, PostgreSQL was the second most popular database, being a runner-up to Snowflake.¹³

In 2022, PostgreSQL was the winner of the third most popular database. Snowflake and Google BigQuery¹⁴ were #1 and #2 but neither are used for OLTP.

The 2022 Stack Overflow Developer Survey¹⁵ gathered input from nearly 50,000 Professional Developers. When the developers were asked what database they used most, PostgreSQL took the #1 spot.

Your investment in yourself building skills with PostgreSQL and Ruby on Rails is a smart career move. These are very popular technologies that are in high demand, used by small startups to giant corporations, being continually improved year after year.

The future is very bright for PostgreSQL and Ruby on Rails!

11. <https://rails-hosting.com/2022/#databases>

12. <https://db-engines.com/en/ranking>

13. <https://www.snowflake.com>

14. <https://cloud.google.com/bigquery>

15. <https://survey.stackoverflow.co/2022/#most-popular-technologies-database-prof>

Part I

Getting Started

An App to Get You Started

Welcome! PostgreSQL and Rails are mature, powerful, and modern technologies. If you're here to level up your skills using these technologies together, you've picked up the right book.

The best way to learn is from hands on practice and experimentation. Here you'll work on exercises using practical examples, with a Rails API app connected to PostgreSQL. You may want to challenge yourself on topics you're familiar with to build your own examples. Growth happens when solving challenges just beyond your current skill level.

In this chapter, you'll primarily get things installed and set up. You may have a lot of experience with Ruby on Rails and less or none at all with PostgreSQL. On the PostgreSQL side, you may be an experienced administrator but know little about Ruby on Rails, or perhaps use another popular web framework. Here you'll get information wherever you fall in those skill areas.

The Ruby on Rails app you'll use here is called *Rideshare*, and it's proudly configured to work with PostgreSQL. This is an open source Rails application that's available to download from GitHub.

Rideshare serves as a demonstration application both for Rails and Active Record concepts, and for PostgreSQL concepts. When the topic involves working directly with PostgreSQL, you'll use the Rideshare database. When the topic is a Rails topic, you'll work with the Rideshare Ruby codebase.

What's the Rideshare app all about?

What Is Rideshare?

Rideshare is an API-only web application that implements a portion of a fictional ridesharing service. Think of Uber or Lyft. Imagine there could be mil-

lions of riders booking trips provided by thousands of drivers in cities all around the world.

Although a global ridesharing application would likely have a very large codebase covering dozens or hundreds of models and database tables, Rideshare has less than ten models and tables. The portion of the domain model is a small core or sample that's needed mainly for demonstration purposes in examples and exercises you'll see.

The core nouns you'll work with are *Drivers*, *Riders*, *Trips*, and *Trip Requests*. These are Active Record models that are backed by database tables.

As a refresher or if you're newer to Rails, Active Record is the Object Relational Mapper (ORM) framework for Ruby on Rails. Active Record connects object oriented Ruby classes as application code, to the persistence layer. Active Record Objects are instantiated in memory and when they're saved or updated, a corresponding INSERT or UPDATE is sent to PostgreSQL.

Active Record works with many naming conventions that are important to know. Models use a singular form for names like "Driver" in memory but by convention are persisted into a table with a pluralized name like `drivers`. The *Convention Over Configuration* design principle that's a code aspect of Ruby on Rails means that things like a table name in a model does not need to be configured explicitly.

Rideshare has an *Entity Relationship Diagram* (ERD) available as a PDF in the root directory of the project. The ERD is automatically updated when schema changes are made. See the footnote with a link to the ERD where you can visualize the database tables and their relationships.¹

Active Record Schema Management Refresher

Active Record tracks database schema changes in a file. The file may be in a Ruby format or a SQL format. Rideshare uses the SQL format. After every schema modification is made, the Rideshare database schema is dumped into the file `db/structure.sql`, which is version controlled. This file will always have the latest version of the schema.

When you're working with a new Rails application, explore the `db/structure.sql` file and any available diagrams like ERDs, to begin to learn about the tables, indexes, and constraints.

1. <https://github.com/andyatkinson/rideshare/blob/master/erd.pdf>

If you're newer to Rails, you might be wondering how changes are captured in the `db/structure.sql` file. When a developer makes a schema modification, behind the scenes Ruby on Rails when configured for PostgreSQL uses `pg_dump` to dump the table structure to `db/structure.sql`.

If you've used a schema modification management tool like Flyway or Liquibase, Active Record Migrations are going to be somewhat similar. In Ruby on Rails, Active Record is built-in and is the way developers evolve the schema.

Each modification is called a "Migration", which is an overloaded term. In Rails a Migration is a versioned Ruby file in the `db/migrate` directory. Rails Migrations use an Active Record API that maps Ruby methods to SQL DDL changes.

Within the Ruby source code file, schema modifications can be written in SQL as well. Do Rails Migration files only contain structural changes, aka Data Definition Language (DDL)? No, there aren't any limitations on them, so developers can perform Data Modification Language (DML) statements in the form of inserting, updating, or deleting data. However, it's considered a good practice to use Migrations only for DDL and use Rake tasks or other mechanisms for DML.

Does the `db/structure.sql` contain only structural data about the schema? Although nearly all of the content of the file is purely structural, there is one exception. At the bottom of the file there is "data". The versioned Migration files each have a unique number. The number is inserted into the developer's database when the Migration runs.

How is this tracked then? When the structure file is dumped, it's dumped based on the developer's database, so any migrations that have been applied there are dumped as INSERT statements. The schema migrations are dumped as INSERT statements so that the database structure can be replicated on another database by running `bin/rails db:schema:load`.

What this task does is load the content from the structure file and then insert all the same migration versions into the destination database. That way if another migration comes in after that one with a version number that doesn't exist, the database knows which versions have already been applied and will apply only the new ones.

That was a quick refresher on how Active Record Migrations manage the PostgreSQL database structure, how developers make incremental schema modifications, and how those modifications are tracked.

When you clone the Rideshare application and “migrate” (a verb in Rails speak) the database, you’ve evolved your database schema to be in sync with the latest production version.

Now that you have an idea of what the Rails application is, how the database structure evolves and how to get a copy of the structure, you’ll want some data.

You’ll use a couple of ways to get data into your database. To make it easy to get started, you can use a pre-made data dump that can be downloaded and loaded.

Then you’ll learn how to populate even more data via scripts that are part of the Rideshare codebase.

It’s time to get Rideshare set up locally so you can explore the source code and start the database server and web server.

Installing Rideshare

Begin by installing the dependencies needed to run Rideshare. Once the dependencies are installed you’ll install the Rideshare application.

The examples here were developed on Mac OS, using the latest released versions of Ruby on Rails and PostgreSQL at the time of writing; these are listed below.

- Mac OS Ventura
- Ruby on Rails 7 (Released 2021)
- PostgreSQL 15 (Released 2022)

Run the rails executable that’s part of the application by running `bin/rails` from the root directory of Rideshare. You’ll use `bin/rails` to run commands like `console`, `server`, and `runner`. You might see instructions to install Rails using `gem install rails`. This method will make a rails executable available, but don’t use this version for these exercises.

If you’re an experienced Rails application developer and PostgreSQL user, you may be able to skip the next few sections.

If you’re less experienced with Ruby on Rails or with PostgreSQL, need a refresher, or prefer to make sure everything is installed upfront correctly, go through the instructions below to get everything configured on your local development computer.

By the end you’ll have Rideshare fully running locally for development, connected to a PostgreSQL database server instance.

Installing PostgreSQL on Mac OS

PostgreSQL can be installed in a variety of ways on Mac OS. The recommended method is to use PostgresApp <https://postgresapp.com>. This is a native Mac OS application. The installation process is the same for any installation with a .dmg extension. Download the file using a browser, and double click it to start the installation process.

Installing PostgreSQL using Homebrew or another method is also perfectly fine. If you're comfortable doing that, go for it.

Along with the PostgreSQL server installation, you'll get additional PostgreSQL Client Applications like `pg_dump` and `psql`².

After installing PostgreSQL in one of the ways above, check the versions of your client programs. This helps you verify they're installed properly. Run the following commands from your Terminal.

```
psql --version
pg_dump --version
```

With your PostgreSQL server instance started, open up your terminal and type `psql` to connect to a default database that matches your OS username, as a user (role) that's your username. If your Mac OS username is `andy` (type: `whoami`) without any arguments to `psql`, it will try and connect to a database called `andy` with a user `andy`. If you see a `psql` prompt, you know you're connected. You may type `SELECT current_database();` to view the current database you're connected to. Type `\q` to quit.

With your PostgreSQL server instance running, and being able to connect to it for any database, you're ready to proceed to work on getting the Ruby language installed, and then the Rideshare Rails app installed.

Installing and Configuring Your Ruby Environment

Rideshare is a Ruby application, so you'll need Ruby running on your computer. From your Terminal, type `ruby -v` to see if you already have a version of Ruby installed. Regardless, you'll need the specific version that's supported by Rideshare. To find the specific version, Ruby apps have a convention to place the version in a file in the root directory called `~/ruby-version`. Ruby Version Managers were created to help you install and manage multiple versions of Ruby installed at the same time. When a version is detected, the version manager should automatically switch to that version.

2. <https://www.postgresql.org/docs/current/reference-client.html>

The Ruby version manager called `rbenv`³ is a good choice. In this section you'll install `rbenv` and the version of Ruby that's needed for Rideshare.

After Ruby is installed you'll install the Bundler gem.⁴ Bundler will take care of all the other dependencies needed. If you're new to Rails it may be a little daunting to get started, but this combination of Ruby Version, Bundler, and running `bundle install` to install all the other gems needed is a common pattern.

In Ruby, third party shared library code is packaged and managed as a Ruby gem. Ruby gems have an official website and repository, <https://rubygems.org>, where you'll find loads of gems and some statistics about their usage.

The instructions below are minimal. Extended instructions can be found in the Installation Guides Appendix. Instructions will be maintained at https://github.com/andyatkinson/development_guides as well.

Ruby on Rails and PostgreSQL are supported in local development for both Linux and Windows, but you'll need to find your own installation and development guides for those.

On Mac OS, a popular package manager called Homebrew⁵ is the recommended way to install the Ruby Version Manager `rbenv`.

Install Homebrew using the latest available instructions at <https://brew.sh>. For Homebrew you'll work from your Terminal program running commands that start with `brew`. When Homebrew is installed run `brew -v` and verify that it is version 4 or newer.

Now you're ready to install the Ruby version manager.

Run the following `brew` command from your terminal to install `rbenv` and `ruby-build`.

```
brew install rbenv ruby-build
```

To verify these are installed, you may run `rbenv --version` and `ruby-build --version`.

With `rbenv` installed and Ruby versions available, you're ready to install the Ruby version needed for Rideshare.

3. <https://github.com/rbenv/rbenv>

4. <https://bundler.io>

5. <https://brew.sh>

Installing the Rideshare Application

To install Rideshare, work from your terminal and use git to clone the application source code. Run `git --version` to confirm you're running at least version 2.

1. Download the application source code by typing `cd` to navigate to your project source code directory. For example, that might be `cd ~/Projects`. From there, run `git clone https://github.com/andyatkinson/rideshare.git`
2. Once cloned, change into the Rideshare by typing `cd rideshare`

If you type `pwd` you should now be located in a directory like `/Users/andy/Projects/rideshare`.

3. From there, you're ready to install the Rideshare gems.

Type `ruby -v` and confirm the version matches the version in `.ruby-version`, which was 3.2 at publication time. Install the bundler gem by typing `gem install bundler`. This is the one gem you'll install using `gem install`. Type `bundler --version` to confirm version 2 or greater is installed.

With bundler installed, you're ready to use it to install the Rideshare gems. From the directory, type `bundle install`. This command installs Rails and all the gems needed to run the application.

If you run into errors, please open an Issue on the project. A common error occurs when installing gems like `pg` that have native code dependencies, and the native code is not accessible. If you run into issues installing the `pg` gem, see the Appendix for more information.

4. If all gems were installed successfully, you're ready to set up the database. From your terminal:

Use `bin/rails db:create` to create the development and test databases. The PostgreSQL server must be running. Using this technique will create a database role (user) named the same as your Mac OS user.

5. You now have the development database `rideshare_development` installed locally and ready to use.

Installation is now complete. To further verify your installation, you may run `bin/rails db:migrate` and confirm no pending migrations exist. You may also wish to run the test suite for Rideshare for additional verifications.

6. To run the test suite for Rideshare, run `bin/rails test` in the Rideshare directory from your terminal.

With Rideshare now installed and configured, you can begin to work with your local PostgreSQL database.

Working With PostgreSQL Locally

If you're an experienced PostgreSQL user you may wish to skip this section.

In the exercises here you'll work exclusively using your local version of PostgreSQL running the open source community distribution. You'll get some repetition in by regularly developing and testing in your local database. Your local PostgreSQL database will have less functionality compared with modern cloud hosted PostgreSQL instances, but it will be less complex and you'll have full control over it. You'll use your local instance to develop skills you can take with you into your production instance.

You'll view log files, modify parameters, and start and stop your server instance. Since you're working locally on PostgreSQL, any queries or processes are in your control and you may restart it at will. In production a database restart requires careful planning and is generally avoided. Thus, you'll practice "reloading" configuration changes where possible to avoid restarts.

PostgreSQL is an extensible database. Here you'll be limited to open source PostgreSQL only, and you'll work with extensions from the PostgreSQL "ecosystem". Extensions expand the core functionality. The extensions featured here have broad support from the major PostgreSQL cloud providers. If you're self-hosting PostgreSQL you can of course install and configure any extensions that you wish.

With that in mind, let's jump into some PostgreSQL terminology.

Learning PostgreSQL Terminology

PostgreSQL is an object relational database. If you're a programmer and are familiar with Object Oriented Programming (OOP) then you might think of objects like Ruby classes and inheritance hierarchies.

In PostgreSQL, database objects are things like tables, indexes, schemas, column defaults, or constraints. Objects in Ruby and objects in PostgreSQL refer to different things.

Databases have a LOT of terminology to familiarize yourself with. Newer database users may benefit from a terminology glossary to begin to digest some of the database specific terminology. Early and frequent exposure to terms is important, so there are terminology guides in most chapters that provide a short, simple definition for each term.

Some terms you'll run across are *relations*, *tuples*, *bloat*, and the *optimizer*.

A *relation* refers to a table. *Tuples* are versions of rows in tables. *Bloat* has to do with inefficient data layout, where data is stored in a fragmented way and this fragmentation can add latency to operations. These very brief definitions are a starting point for exposure to these terms.

Some basic information on PostgreSQL data storage internals is useful. There's a lot more that we won't discuss.

In PostgreSQL, data is laid out in *pages*. Data cannot be split across pages. Data is stored efficiently when there are minimal gaps and data that is accessed resides within a single page. Data that's in a single page requires fewer disk operations compared with accessing data that is split across multiple pages.

Another concept on data layout is *correlation*. Correlation refers to the relationship between how data resides on disk and how the data is ordered in a query. High correlation means they're similar and low correlation means they are dissimilar.

PostgreSQL works whether data is stored efficiently or not. Data may not fit in a single page, may be split across pages, and there may be a low correlation for a query. You will explore these concepts.

Next let's look at some SQL terminology.

Learning SQL Terminology

SQL has loads of terms as well, and much of it is not specific to PostgreSQL.

SQL is a *declarative* language. As a programmer, you *declare* to PostgreSQL what results you want and PostgreSQL will figure out the best plan to get them.

You specify the columns from the tables you want, and PostgreSQL determines an optimized retrieval algorithm using knowledge of the data layout, database objects, available hardware resources, and more.

PostgreSQL creates *query execution plans* on the fly to fulfill what you've asked for. Multiple plans are pitted against one another. The plans are formed using estimates about data in tables, so the plans aren't always flawless but are generally pretty good. As the programmer, you can see what PostgreSQL is planning and influence it. You'll learn more about that later on.

This plan evaluation process is conducted nearly instantly by the query execution planner, which is also referred to as the *optimizer*.

The optimizer uses a *cost-based algorithm* to compare costs and select the lowest-cost plan. Plan selection is a very interesting and deep topic that you'll also get some experience with.

Besides writing SQL, you'll run statements and commands for PostgreSQL, sometimes using built-in *functions*. Functions enhance your database; there are built-in functions and you may create your own as *User Defined Functions*. Functions can be written using SQL or other programming languages.

Statements you write in PostgreSQL can be classified in a couple of ways. A statement that changes the structure of the database, like creating a table, is called a Data Definition Language (DDL) statement. An example would be CREATE DATABASE which is not a full statement and won't run as-is, but meant to be an example.

Another type of statement modifies data and not structure. This statement type is called a Data Manipulation Language (DML) statement. An example is an INSERT statement that inserts data into a table. Database users might use these acronyms DDL and DML in conversation to categorize statement types when the categorization is relevant.

PostgreSQL is a transactional database, which means most operations take place inside of a *Transaction*. Transactions are used to group together multiple operations that should occur as one unit. Transactions have an isolated view of data and the level of isolation can be configured, which you'll learn about.

Let's recap some of the terms you've seen so far.

Glossary of Databases Terms




- Relation or rename — a table
 - Data Manipulation Language (DML) — e.g. INSERT, UPDATE, and DELETE operations
 - Data Definition Language (DDL) — e.g. Structure modifications like ALTER TABLE or CREATE INDEX statements
 - Transaction — A unit of work
-

Ruby on Rails Terminology

Active Record is the Object Relational Mapper (ORM) for Ruby on Rails. Active Record classes are Ruby classes. A class also serves as an interface to a table where model instances are persisted as table rows. Thus an Active Record Ruby class also acts as a persistence interface for application code to write

or read data from database tables even when no other configuration within the class is written.

Active Record goes beyond simple persistence and includes lots of data modeling and data validation features. Active Record objects have *Associations* to link models together and *Validations* for validating the correctness of input data.

Active Record Terminology	
	• Object Relational Mapper (ORM) — Maps relational database tables and to Active Record model classes in Ruby on Rails
	• Migrations — In Rails, Migrations are incremental modifications to the database structure.
	• Associations — Expressing data model relationships with Active Record using methods like <code>has_many</code> , <code>belongs_to</code>
	• Validations — Application level consistency checking, <code>null/nil</code> , <code>format</code> , model relationships

Now that some of the standard PostgreSQL, SQL and Rails terms are clear, let's look at conventions used in the coming chapters.

Conventions Used In This Book

Code snippets are used extensively for a variety of types of code like shell scripts, SQL, Ruby scripts, and more. Code snippets are available for download. The language is selected to help with coloring the keywords when viewing the text with color.

For CLI programs with options that support short and long option names, like `-v` or `--version` for version, the longer syntax is used.

Shell scripts start with `sh` in the filename. These are meant to be run in your Terminal program. On Mac OS, use either the Terminal.app terminal or a third party terminal like iTerm.⁶

For example, a shell script called `load_data.sh` would be run from your terminal by typing `sh load_data.sh`.

SQL keywords are capitalized and monospaced like `SELECT`, `WHERE`, and `LIMIT`. Portions of a query that are incomplete are referred as *clauses* or *fragments*, although full queries to run in a SQL client are preferred.

6. <https://iterm2.com>

In the SQL query `SELECT * FROM trips WHERE id = 1`; the fragment `WHERE id = 1` might be called the *where clause*.

To help identify SQL keywords, when the SQL statements operate on tables and fields, the tables and fields are lowercased.

In the example above, `trips` is a table name and `id` is a field name so they are lowercased.

SQL queries are meant to run in a SQL client. In this book `psql` is the recommended client, but any SQL client will work.

Once you're connected to PostgreSQL, copy and paste or type the SQL query into your client to run it.

Rails code is written in the Ruby programming language. Rails code will have rails in the code sample. Ruby is a interpreted language so you can run any Ruby code from the `bin/rails` console or the Ruby interpreter at any time. The `bin/rails` console is where you can run Active Record code and other Rails application code to see how it behaves. Copy and paste the code and or type it in and press Return to run it.

Note that both SQL and Ruby code can be run from the command line as well.

For SQL queries, `psql` supports a `--command` option. For example, type `'psql --command "select 1"'` to run a SQL query from your terminal.

In Ruby you can run `ruby -e "puts 'hello'"` to run Ruby code from the command line. Rails extends this with `bin/rails runner` which allows you to run Rails code from the command line. From the Rideshare root directory for example, try this out by running `bin/rails runner "ActiveRecord::Base.connection.execute('select 1')"`. This command doesn't depend on you yet having the Rideshare data loaded, and is meant just to show that Active Record is connected to PostgreSQL and can execute a simple query.

Ruby and Rails methods will be lowercased and have explicit parentheses around them like `.average()`.

Database functions will be capitalized and have parentheses as well, like `SELECT AVG()` to help show that they are functions.

Ruby on Rails Frameworks with multiple words a space separated, for example "Active Record" or "Action Pack".

You're Ready

You're now ready to jump into some real work.

One of the main goals is to help you improve your PostgreSQL operator and administration skills. Many Rails developers do not typically work on operation and administration directly with PostgreSQL.

To improve your skills you'll need to get comfortable performing tasks, modifying configuration files, viewing logs, and understanding how to observe what's currently happening inside the database.

It's time to start building those operator and administration skills.

Part II

Design and Build

Administration Basics

Welcome! In this chapter you'll get hands on with some basics of PostgreSQL operation and administration.

In the last chapter, you installed and configured the Rideshare Rails application connected to a database running on your local PostgreSQL server. If you skipped the Introduction and aren't familiar with Rideshare or don't have it installed, consider going back and reading that section. At the end you'll have a full development environment with the correct version of Ruby, Rails, and PostgreSQL installed, running on Mac OS. You'll be in a spot to run any Active Record code from `bin/rails` console for Rideshare, or run SQL statements for the Rideshare database.

PostgreSQL Administration is a broad subject and in this chapter you'll be getting an introduction to a variety of topics to help you branch out and go into greater depth.

There may be a lot of new terminology to learn. You'll work with meta-commands, `psql`, database activity observability, locks, database parameters, and extensions.

Most Rails developers don't write a lot of SQL directly to modify the database schema. Instead they write Active Record *Migrations* that are Ruby files that generate SQL DDL statements in order to evolve the database schema.

Since our goal is about operations and administrations knowledge, you'll begin to write more SQL modification statements and run them directly using a SQL client. You'll modify the database structure, populate data rows, and you'll write these statements with SQL.

Let's dive in using the Rideshare development database.

Learning Meta Commands In psql With Rideshare

When connecting to PostgreSQL you'll need a client application. GUI applications are great, but in this book you'll work solely from your Terminal and run the `psql` client that comes with PostgreSQL.

`psql` is recommended for several reasons. To learn more about the thinking behind recommending `psql`, refer to the Appendix “Why `psql`?”

To start, type `psql` with the `--dbname` option specifying the Rideshare development database:

```
sh/psql_dbname.sh
psql --dbname rideshare_development
```

A backend process ID corresponds to your `psql` session. Run `SELECT pg_backend_pid();` to see the value.

The command above doesn't set the host of the server, or any user, password credentials, or a connection string URL. If the command doesn't work as-is, review the “Development Guides” Appendix. Make sure you're able to easily connect to your running PostgreSQL database server before proceeding.

With that out of the way, you're now staring at a blank `psql` prompt. Now what?

Try typing `\s` and hitting Enter. This might be the first *meta-command* you've used if you're new to `psql`. `psql` has loads of meta commands. Some are listed in the post *17 Practical psql Commands*¹ and you'll see a lot of them here. This meta-command lists the history of SQL statements or commands you've run in `psql` before. If you run a statement like `SELECT 1;` and then run `\s` again, you'll see the statement in the history.

Administration Terminology



- Meta Commands — `psql` commands that start with a backslash
 - Backends — A backend process with a pid and query
 - System Catalog — A PostgreSQL table or view you might use to inspect internal state information
-

In the same way the history command works in your Terminal, you can do a “reverse search” through your commands by typing “`ctrl-r`”. After typing `ctrl-`

1. <https://www.postgresqltutorial.com/postgresql-administration/psql-commands/>

r, type `SEL` to find the `SELECT` statement you'd just run. Reverse searching for past queries helps, and you can run the `\g` to run the last statement again.

The history is kept in the file `~/.psql_history` by default. The history file is meant to be read within `psql`² although it's a plain text file.

If you'd like to customize the location or to generate a distinct file for each database, you may customize the `HISTFILE`. You can increase `HISTSZ` to keep more information in the file.

Any of these customizations can be added to a special `~/.psqlrc` file that's read when `psql` starts up. The Thoughtbot post *An Explained psqlrc*³ shows how to customize `~/.psqlrc` with lots of great tips.

You may also want to edit queries directly in a text editor from `psql`. To do that, use the `\e` meta-command.

Within the text editor, type out SQL queries, wrapping lines and doing formatting as you like. When you save your changes you're returned to `psql` and the query executes.

To save your SQL query output to a file from `psql` use the `\o` meta command, like `\o file.txt`, where `file.txt` contains the query results.

With this option, when you run a query the results are written directly into this file and no output is visible in `psql`. To toggle it back type `\o` without a filename. You should now see output again in `psql` when you run `\g`.

To exit `psql`, type `\q`. View the contents of the output file by running `cat file.txt` from your Terminal. The file will show the query results in the same way as they'd appear in `psql`.

Refer to the meta commands recap below.

Command	Description
<code>\set</code>	Set a variable
<code>\s</code>	See history of commands
<code>\e</code>	Launch a text editor from The code tag should not be here.
<code>\o</code>	Toggle output between The code tag should not be here. and a file query output to a file from

Great, hopefully you're picking up some new `psql` tips.

2. <https://stackoverflow.com/a/31649807>

3. <https://thoughtbot.com/blog/an-explained-psqlrc>

Next you'll use `psql` to modify your PostgreSQL configuration.

Modifying Your PostgreSQL Config File

By getting comfortable modifying your local PostgreSQL installation, you'll be well positioned to confidently edit your production configuration whether you're self-hosting PostgreSQL or using a cloud provider. With cloud providers, you may use a web UI or CLI program to modify parameters, but the basic idea is the same. You modify configuration and either reload the configuration without restarting PostgreSQL or restart PostgreSQL when it's required by a specific parameter.

For your local PostgreSQL installation, you'll need to find the path to the configuration file. Run `psql` from your terminal, and use the following statement to find the path:

```
sql/show_config_file.sql
SHOW config_file;
```

On Mac OS, the path includes a directory with a space in it ("Application Support") so you may need to surround the path in double quotes.

Open the file with your text editor.

You should now be looking at the contents of `postgresql.conf` in your text editor. It's a long file. Search within the file for the section called `shared_preload_libraries`.

The default value might be an assignment to an empty string:

```
shared_preload_libraries = "
```

Next you'll set a value for `shared_preload_libraries` by setting a value inside the single quotes. Before going too far, you may wish to make a backup copy of this file or even check it in to version control. If something goes wrong, you'll want to roll back to a known good version and restart PostgreSQL.

You'll enable two extensions that are included with PostgreSQL by configuring them in this section of the config file. These extensions will be used later, but for now you're just looking to enable them.

First up is the `pg_stat_statements`⁴ extension. This extension is used to collect statistics about queries that are executing on the server. It will be abbreviated as PGSS. The goal here is simply to practice configuring an extension to learn the process.

4. <https://www.postgresql.org/docs/current/pgstatstatements.html>

Type `pg_stat_statements` inside the single quotes using your editor. Once you've done this, save the changes to the file and exit. After your changes the line should look like this:

```
sh/shared_preload_libraries.sh
shared_preload_libraries = 'pg_stat_statements'
```

This change does require a restart. In your production database server, a restart needs to be carefully planned. Other customizations require only a “reload” and not a restart which is much better because it doesn't interrupt any running database queries or activity.

Since you're working locally and are the sole user of the database, restarting is no big deal. If you're using PostgresApp on Mac OS, you can restart PostgreSQL using the GUI. Click “Stop” and “Start” from the GUI app.

To restart PostgreSQL from your terminal, use the `pg_ctl` program and restart argument. This command expects `PGDATA` to be set either as an environment variable, or as a command argument. The value should be the path to your data directory for your installation.

To get the path to your data directory, run the following command from `psql`:

```
SHOW data_directory;
```

Type `\q` to exit `psql` and return to your terminal.

You can now run `pg_ctl restart` and use `--pgdata` with a directory path like `"/Users/andy/Library/Application Support/Postgres/var-15"`.

The full command to restart PostgreSQL in your terminal is:

```
pg_ctl restart -D "/Users/andy/Library/Application Support/Postgres/var-15"
```

Once PostgreSQL is restarted, run `psql` again and run the following query to confirm `PGSS` is listed:

```
sql/select_pg_extension.sql
SELECT * FROM pg_extension;
```

You should see `pg_stat_statements` in the `extname` column.

Great. You've now tried some basics on configuring an extension. Next you'll start to explore what's currently happening in your database while it's running.

Getting Started With Query Observability

By looking at the schema, tables, relationships and structure, you can get an idea of what's being written into the database.

What about what’s being queried? How do you determine which queries are currently running?

PostgreSQL makes current activity available from a *system catalog* view called `pg_stat_activity`. This might be the first system catalog view you’ve worked with.

PostgreSQL keeps metadata internally and bookkeeping information in the `pg_catalog`⁵ System Catalogs. System Catalog tables start with `pg_*` and shouldn’t be modified. They’re very useful to inspect more about what’s happening inside PostgreSQL.

To view current database activity, launch `psql` and run this SQL query:

```
sql/pg_stat_activity.sql
SELECT * FROM pg_stat_activity;
```

Try toggling `\x` (another meta-command) to make the output more readable. Even if no application queries are running, you might see things like “auto-vacuum launcher” or “logical replication launcher”, which are background processes running within PostgreSQL.

To inspect details about the `psql` session you’ve opened, you can query `pg_stat_activity` by the `pid`, using the `pid` for your own current backend process from `SELECT * FROM pg_backend_pid();`:

```
SELECT * FROM pg_stat_activity WHERE pid = (SELECT * FROM pg_backend_pid());
```

It is safe to run this command in your production database to view the activity from system catalogs. In production on a live system you may find an overwhelming amount of information when querying the `pg_stat_activity` view. If you see `INSUFFICIENT PRIVILEGES` when querying it, you’ll need to connect to your database using an administrator role.

Take a look at the `pid` Process ID field.⁶ If you have a query that’s running too long and you want to stop it, you’ll need the `pid`.

From `psql` with the `pid`, the PostgreSQL System Administration functions describe how a *backend* has a query. To stop a query you’ll cancel the backend using System Administration Functions.⁷

```
SELECT PG_CANCEL_BACKEND(pid);
```

With the cancel form, a `SIGINT` signal is sent which attempts a graceful shut-down.

5. <https://www.postgresql.org/docs/current/catalogs-overview.html>

6. <https://www.postgresql.org/docs/current/monitoring-stats.html#MONITORING-PG-STAT-ACTIVITY-VIEW>

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

If the graceful shutdown does not work, the next option is to terminate the query which sends a SIGTERM signal. These are server signaling System Administration Functions.⁸ Terminating a running query may cause data loss.

Use the following function to terminate the backend. Multiple queries in progress may be associated with the backend process so only use this option when it's really needed.

```
SELECT PG_TERMINATE_BACKEND(pid);
```

Later you'll learn about safeguards you can put in place to help reduce the likelihood of getting into a state where you need to terminate a query.

Besides current query activities, sometimes queries can get blocked on each other. One of the key design features of PostgreSQL is how it locks and unlocks resources to handle a lot of concurrent writing and reading activity.

In the next section you'll peek into some of the details on lock activity.

Glancing At Current Lock Behavior

PostgreSQL uses pessimistic locking, which means lockable resources are locked up front before they're modified. When queries try to access locked resources they can either share access or they are blocked and must wait to acquire a lock that is held from a single statement. Lockable resources include tables and also rows within tables.

Although a deep dive into locks is beyond the scope of this chapter, knowing some of the basic lock types can help you design solutions that minimize lock contention.

Exclusive locks should be minimized because they block other queries trying to access the same resources. The length of the time the lock is taken can grow with the row count of the table. For example when a table modification is happening that locks the table, the lock duration will be longer for a table with more rows because each row needs to change. If the same table is queried in high volume during a long lock duration period, it is very likely those queries will be blocked and result in being cancelled and causing application errors.

Although Locks can be created explicitly, usually resources are locked from implicit locks taken by various SQL statements. PostgreSQL documentation

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

shows conflicting lock modes on the Explicit Locking Documentation⁹. You will learn to map the SQL statements you send to the lock types they require in future chapters.

Although lock durations are typically very short, for high volume queries there can still be blocked queries or even *Deadlocks*. Deadlocks happen when two processes block each other permanently.¹⁰

For live information about locking activity, from psql query the pg_locks system catalog view as follows.

```
sql/view_pg_locks.sql
SELECT * FROM pg_locks;
```

That concludes the extremely brief introduction to lock activity. To recap, pg_stat_activity and pg_locks are system catalogs that can be queried to view live activity. As we move forward, you'll learn various ways to use this information. Lock durations are typically very short. Sometimes you'll need to simulate behaviors that you wish to monitor, like slow queries or long running locks. To do that, you'll create various experiments.

You're ready now to get started on setting up experiments.

Generating Fake Data For Experiments

You've installed and configured the Rideshare application and connected it to a database.

Did you know that on your database server instance, you can have multiple databases running? By default PostgreSQL actually installs several databases. In a default installation, you'll have databases called postgres, template0, and template1. You've created databases called rideshare_development and rideshare_test on your instance.

To verify all of these are there, run psql and type the \l meta-command to *list* the databases.

Type \c postgres to *connect* to the postgres database.

You may have thought “the database” was your PostgreSQL installation, but a database is really something that exists on your PostgreSQL server or instance. Each server has one or more databases.

9. <https://www.postgresql.org/docs/current/explicit-locking.html>

10. <https://www.postgresql.org/docs/current/explicit-locking.html#LOCKING-DEADLOCKS>

Although you'll primarily work with Rideshare, try creating another database. Make one that will be dedicated for experimentation. This will give you another place to experiment with schema modifications, bulk loading rows, or query performance.

From `psql`, type `CREATE DATABASE experiments;` to create the new database.

Next you'll create a table in your `experiments` database and populate it with fake data.

Try writing a SQL DDL statement that creates a table `tbl` with a single column `col` of type `smallint`. Normally you might use Active Record to create a table. It's ok to look this up, but try to write it in SQL if you can. If you're making mistakes that's ok, you're practicing and improving, and you're working locally where there's no negative consequences.

Try it yourself or refer to the SQL below for the answer.

How might you insert 10 rows into your new table? You'll need to be familiar with the built-in `GENERATE_SERIES()` function. In PostgreSQL, this function is available and generates integers that you can use for whatever purpose. You might want the integers to act as values for the column. Using arguments, you can populate as many rows as you wish.

The SQL statements below create the table and populate it with rows:

```
sql/create_table_generate_series.sql
-- create the table
CREATE TABLE tbl (col smallint);

-- populate the table
INSERT INTO tbl(col) SELECT GENERATE_SERIES(1, 10);
```

You've now created a new `experiments` database, a table, and populated it with rows.

What else could you practice as SQL?

As a Ruby on Rails developer, you may not have written `CREATE INDEX` statements in SQL. You've relied on the Active Record `create_index` method to generate the SQL statement for you.

Try creating an index using SQL. In query performance emergencies or even for single-use indexes, connecting to a production database with `psql` and creating an index `CONCURRENTLY` can really help quickly improve query performance and end an emergency. Of course a direct production modification should not be the normal way of working, but it's a good skill to have for when it's needed.

Deploying an Active Record Migration may take a lot of time depending on how fast your code review and software release process is, or on how many database changes the DDL statement needs to run for.

If you are very uncomfortable with the idea of direct database modifications, that's ok, your intuition is right. You want changes to be managed with a change management process, reviewed by team members. However, here you're looking to shortcut all of that and build your SQL DDL skills.

From `psql`, create an index called `test_index` for the `tbl` table you created.

This index should cover the `col` column on the `tbl` table and have the default B-Tree type. Don't worry about the purpose of the index; right now you're just getting comfortable developing and running the statements.

Type `\timing` to toggle on timing.

With timing enabled you'll see how long the `CREATE INDEX` statement runs. On large tables it can take a long time to create an index, so it's good to have an idea of how long this will take or sometimes test it in a lower environment.

Adding indexes on live systems in production should be done using the `CONCURRENTLY` keyword most of the time, which means that the index creation won't lock the table. However, using `CONCURRENTLY` makes the operation take about twice as long!

Try and write the SQL on your own first. When you're done or get stuck, refer to this SQL statement:

```
sql/create_index_basic.sql
```

```
CREATE INDEX test_index ON tbl (col);
```

Try dropping the index you just created using `DROP INDEX`. From `psql`, run `DROP INDEX test_index;`. Note that indexes can be dropped concurrently as well, which is a good idea on a live production system!

Rolling Back Schema Modifications

PostgreSQL has a feature called *Transactional DDL*.¹¹ Transactional DDL means that when structure changes are made, they either fully succeed or are fully rolled back depending on whether the transaction ends in a `COMMIT` or `ROLLBACK`. Transactional DDL is a key feature of PostgreSQL because it means that Active Record Migrations that change the database will succeed or fully roll back, and avoid a partially applied modification.

11. https://wiki.postgresql.org/wiki/Transactional_DDL_in_PostgreSQL:_A_Competitive_Analysis

Speaking of transactions, what is a *database transaction*? Rails developers may be less familiar with database transactions. Here are the basics.

Transactions can be explicitly created, but are usually implicit based on the statements you're running. Transactions help facilitate making multiple modifications to structure or data as one unit.

Try out Transactional DDL a bit with an experiment. Try making a DDL change and use an explicit transaction that you create.

Inside psql, type `BEGIN;` to open your transaction.

Inside the transaction create a DDL change like adding an index, like you did earlier. When you aren't using an explicit transaction, an implicit transaction is created for the operation.

When you use an explicit transaction you have "transaction control." With transaction control you may decide to roll back the transaction. After typing the `CREATE INDEX` statement and a semi-colon to end the line, on a new line type `ROLLBACK;`.

The `ROLLBACK` keyword rolls back a transaction, which means it's not applied or saved to the database. When this happens, thanks to the Transactional DDL feature the index creation is not applied. From psql, confirm that the index does not exist.

The following statements show this scenario where an index is added inside a transaction that can be committed or rolled back:

```
sql/transactional_ddl_changes.sql
BEGIN;
CREATE INDEX test_index ON tbl (col);
ROLLBACK;
```

Now that you've seen Transactional DDL, it may be tempting to experiment using your production database. Is this a good idea?

Exploring and Experimenting Safely In Production

Most companies have a staging environment that's running the application with a PostgreSQL database separate from production.

A staging environment is a great place to perform some kinds of experimentation. Take advantage of this if it's available to you!

Write down the operations you're performing in a ticket tracking system and share it with a teammate who can double check it. Create a backup before

running any data or structure modifications. Discuss a rollback plan when that's appropriate.

For queries, consider doing all exploration queries on a replica database. Replicas run in a *read only* mode so destructive DML operations like TRUNCATE aren't possible and result in an error.

Consider creating a read only PostgreSQL user on the primary database using the `pg_read_all_data` or `pg_monitor` role. The article *Creating a Read-Only Postgres User*¹² from Crunchy Data describes how to do this. By regularly using a read only user, you'll add another safety mechanism that prevents accidental modifications.

If you're creating database tables that are temporary and should be removed later, create the tables in their own schema. You'll use that technique in this book by creating tables within a temp schema.

A schema named temp helps reveal the temporary nature to other team members, which helps them more easily clean it up later if you forget to!

Almost all of the content in your production PostgreSQL database for your Rails application should have a corresponding Active Record Migration file. Besides the separate temporary schema, tables created in the application schema (public by default) that do not have corresponding migrations can probably be removed.

One technique to assist in removing tables is to rename them first using an ALTER TABLE statement. A renamed database table that doesn't match the table name expected by the application will produce errors. To roll this change back quickly, run an ALTER TABLE again that renames it back to the original name.

For example, from psql connect to your experiments database and rename the tbl you created earlier.

Run this statement:

```
ALTER TABLE tbl RENAME TO tbl_pending_delete;
```

This could be an intermediate step that allows you to quickly rename it back if needed. If no application errors are observed after a period of time, you can confidently drop the table by running `DROP TABLE tbl_pending_delete;`.

You now have several techniques to add safeguards when working directly with your production database.

12. <https://www.crunchydata.com/blog/creating-a-read-only-postgres-user>

While these techniques are useful, none allow you to evolve the schema of the database.

New tables, new constraints, new data relationships, and new indexes are common ways that applications evolve as developers add more features or improve the data model over time. While schema evolution happens locally during the development cycle, it's not always practical to populate huge amounts of data in local databases.

Your staging environment or even a dedicated performance testing environment helps you analyze and test performance problems and allows you to safely modify the structure when needed. Wouldn't it be nice to have a performance database to test on? How might you set that up?

In the next chapter you'll set up a Performance testing database where you can make these kinds of changes. You'll build up a Performance database by starting from the Rideshare database and making it safe to use.

Let's get started!

Building a Performance Testing Database

In the previous chapter you started working directly with PostgreSQL from `psql` and learned about the Rideshare application. You ran meta commands, inspected system catalogs, and configured an extension.

Moving along, you're about to learn to develop your own copy of your production database where you can begin to experiment with more extensive types of changes including structure modifications. You can also use this database for Performance testing, so it will be called the Performance database.

You'll begin to use advanced PostgreSQL concepts like *Functions* and *Procedures* by solving the very common problem faced by development teams, which is how they can get production-like data in their pre-production environment, without compromising any sensitive customer data. Some of the reasons to have production-like data are for performance testing of queries or for experimenting with indexes or different data model designs.

You'll write custom functions using SQL and learn about the *PL/pgSQL* Procedural Language. If you've never worked with this before don't worry, no prior knowledge is expected.

As your application scales up to higher amounts of queries and tables with greater row counts, having a place to test query changes and structure modifications becomes even more valuable.

To get started you'll download a pre-made database dump file for Rideshare. The database dump includes the structure and database objects but it also includes simulated sensitive customer information. Rideshare is fictional and the generated users are fictional as well, but you can treat these as if they are real users like you might find in a production database dump from your application.

When columns have values that uniquely identify individuals they're considered to be sensitive. A good security practice is to limit access to sensitive data. To do this, you'll build a *scrubbing* process that replaces sensitive values with data that is representative of real data, but benign.

A design goal of the scrubbing techniques you'll use is that they'll work quickly even on millions of rows. This provides an opportunity to discuss some of the ways that row modifications can slow down in PostgreSQL.

Another design goal is that the *scrubbed* data maintains the statistical properties of the original data as much as possible. This way your query performance analysis in your Performance database will very closely match your production database.

Performance Database Terminology



- Schema — In PostgreSQL it's a namespace
 - Scrubbing — Replacing sensitive text in your database fields, as a good security practice
 - Functions — Database functions in SQL or the built-in procedural language pl/pgSQL
 - Procedures — An extended form of functions
 - Cardinality — Unique values in a column
-

The scrubbing process works with all modern versions of PostgreSQL and has no external dependencies. To achieve that level of flexibility, this means you'll be building all of the pieces yourself!

To get started, first you'll download a copy of a pre-made data dump. Read on to find out how to do that.

Downloading Production Data Dumps

The Riders and Drivers in Rideshare were generated using fake values from the Faker gem¹ which produces values designed to look real.

A pre-made dump of one million rows was made using `pg_dump` as below. You don't need to run this now, it's just here to show you what was done earlier.

```
sh/dump_rideshare_database.sh
# create the dump file
pg_dump --dbname rideshare_development \
```

1. <https://github.com/faker-ruby/faker>

```
--no-owner \
--no-privileges \
--file tmp/rideshare_database_dump.sql
# compress the file, force overwrite
gzip -f tmp/rideshare_database_dump.sql
```

Once the dump file was created it was uploaded to GitHub.

The amount of records was chosen to create a dump file that was large enough to be “interesting”, but small enough to download quickly. For performance testing it helps to have large amounts of row data to work with, so after downloading the dump you may wish to generate even more data for your local Rideshare database.

Dumping the Database Schema

You may have been using `pg_dump` and not realized it. As the database schema is modified, changes show up as diffs to one of two possible files in your Rails application. The file is called `db/schema.rb`, when the Ruby format is used. Or the file is `db/structure.sql`, when the SQL format is used. Rideshare uses the SQL format. The SQL format uses `pg_dump` behind the scenes, but dumps only schema changes and not data. Explore `pg_dump` options to learn how to dump both schema and data.

In this section you’ll download and load the data file. The steps are all written out in a shell script.

You downloaded and installed Rideshare already. `cd` into the source code `rideshare` directory.

From the Rideshare directory, run `sh reset_and_load_data_dump.sh`. If you prefer to run the commands one by one, that’s ok too.

```
sh/reset_and_load_data_dump.sh
#!/bin/bash

# Important Note:
# Run this script from parent directory
#
# cd ..
# sh scripts/reset_and_load_data_dump.sh

echo "Make tmp dir"
mkdir -p tmp

echo "Download dump file"
curl -L \
https://github.com/andyatkinson/rideshare\
/raw/master/rideshare_database_dump.sql.gz \
-o tmp/rideshare_database_dump.sql.gz
```

```

echo "Decompress file"
gzip -d tmp/rideshare_database_dump.sql.gz

echo "Decompressed file is around 100 MB"
du -h tmp/rideshare_database_dump.sql

echo "Re-create the empty database"
bin/rails db:drop:all
bin/rails db:create

echo "Load the dump file using psql. \
This will take a minute..."
psql --set ON_ERROR_STOP=on --quiet \
  --no-psqlrc \
  --output /dev/null \
  rideshare_development \
  --file tmp/rideshare_database_dump.sql

echo
echo "done!"

```

After the script runs, the `rideshare_development` database should be populated with one million records for the `users` table. The `Rideshare` `users` table is a polymorphic table and stores `Drivers` and `Riders`.

Run `psql --dbname rideshare_development` and count the `users` table records to verify that everything was loaded.

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

Describe the `users` table by running `\d users` from `psql`. Notice `users` has fields like `first_name` and `email`. The values in these fields are unique to individuals are considered sensitive. These are some of the columns you'll scrub!

How Do I Identify Sensitive Data?

To identify sensitive data, you'll look through all columns in all tables, identifying column names that describe personal information, and collect some sample rows. At most businesses, you'd then work with stakeholders like Product Owners, Business Analysts, and Security team members, to help classify the sensitivity of the data, and implement access controls.

Your `rideshare_development` database now has data that simulates a production database that is not scrubbed. This will be your starting point for developing your scrubbing process. If you wish to start over at any point, recreate the database (`bin/rails db:reset`) and download or load the dump file.

Now you're ready to jump into the scrubbing process itself. The first concept to tackle is how to keep the statistical properties the same when scrubbing data. What does that mean?

Replacement Values That Are Statistically Similar

The statistical properties of the data in tables, columns, and indexes all affect query execution performance.

Query execution plans are determined dynamically at execution time. PostgreSQL uses estimates and statistics about the data to estimate the cost of alternative plans. The query planner chooses the lowest cost plan that produces the fastest data retrieval. We'll get into details of query execution plan generation and plan selection later; for now, it's enough to know that table statistics are important to query planning.

What are some of the statistics collected by PostgreSQL?

As you've already learned, PostgreSQL keeps system catalog tables for internal bookkeeping. Statistics about the table rows that PostgreSQL keeps can be queried using the `pg_stats` System Catalog view.² The statistics include things like the most common values or how many rows have NULL values in their columns.

For the Rideshare database, take a look at the statistics captured for the `users` table by running the following query from `psql`. An example result row is shown below.

```
sql/table_statistics_users.sql
```

```
SELECT
  attname,
  n_distinct,
  most_common_vals
FROM pg_stats WHERE tablename = 'users'
AND attname = 'first_name';

-- # SELECT attname, n_distinct, most_common_vals FROM
-- pg_stats WHERE tablename = 'users' AND attname = 'first_name';
--
-- -[ RECORD 1 ]-
-- attname          | first_name
-- n_distinct       | 5122
-- most_common_vals | {Elroy,Maurice,Tristan,Dion,Ariel,Angel,Chi,
-- Chris,Frances,Luis,Mark,Ray,Fermin,Guadalupe,Logan,Perry,Royce,
-- Scott,Vernon,Carey,Carson,Cary,Dalton,Erin,Eugene,Frank,Glenn...
```

2. <https://www.postgresql.org/docs/current/view-pg-stats.html>

The commented out portion shows an example record.

[RECORD 1] shows that 5122 distinct first names are captured in the `n_distinct` column.

You can see some of the most common first names like Elroy, Maurice, and Tristan in the `most_common_vals` column. “Elroy” is one of the most common first names. The `most_common_freqs` column shows the frequency at which the most common values occur.

Calculate the frequency of Elroy as a first name. The frequency is the total number of occurrences of “Elroy” divided by the total number of rows in the `users` table. Since Elroy appears 405 times, divided by 1,000,000 total rows, it has a frequency of 0.000405.

The *cardinality* of a column describes the distinct values the column has.

For query performance analysis it’s ideal to closely match replacement text with the original text to keep characteristics like cardinality and frequency similar to what’s in production. Besides that, the space consumed within the column is worth keeping as close as possible.

“Elroy” is a string of five characters. “Elroy” is stored in a character varying column. A good replacement for Elroy would be five characters in length and stored in a column of the same type.

Another statistic for columns is the percentage of NULL and non-NULL values. PostgreSQL tracks this as the `null_frac` or null fraction.

NULL values in columns consume less space but are still maintained in indexes.

When replacing text content, consider the column cardinality (*What Does Cardinality Mean in a Database?*³), frequency distributions, and the proportion of NULL values as you design your text replacement functions.

You’ve now had a brief overview of some of the statistics PostgreSQL tracks about the data.

As you begin to identify sensitive columns like email that you want to scrub, how will you keep track of all the columns over time?

3. <https://vertabelo.com/blog/cardinality-in-database/>

Tracking Columns With Sensitive Information

To maintain a list of columns with sensitive values, you can track that information within PostgreSQL.

PostgreSQL gives users the ability to add database-level comments to tables and fields. These comments are then dumped just like other structural details when `pg_dump` runs. You'll use the column commenting capability to track your sensitive columns.

Connect to the `rideshare_development` database from `psql` and run this SQL statement:

```
sql/create_comment_for_table_column.sql
COMMENT ON COLUMN users.email IS 'sensitive_data=true';
```

The statement adds a comment to the `users.email` field using a simple key and value structure to mark a sensitive field.

To confirm the comment was added, run `\d+ users`. This expanded form of `\d` includes additional information like comments. If the output from `\d+ users` is difficult to read, try running `\pset format wrapped to toggle wrapping` and run again.

To the very right of the email column you should now see `sensitive_data=true` as the column comment.

For each sensitive field you've identified, create this type of comment.

To keep all databases in sync, create the database comments using a Rails Migration app. Database comments will then be included in `db/structure.sql` and become part of your database structure.

Now that you know you've started to identify and track columns to scrub, how will you perform the scrubbing?

Comparing Direct Updates and Table Copying Strategies

For your scrubbing system you'll use two techniques which can be called "Direct Updates" and "Table Copying."

Direct Updates are simply a SQL `UPDATE` statement that overwrites column values. Direct Updates are a strategy recommended for smaller tables with fewer than one million rows, because `UPDATEs` can become slow on very large tables.

Some database maintenance is required after Direct Updates as well. Due to the design of PostgreSQL, an UPDATE statement for a row leaves around a former version of the row that's now considered “dead”; dead rows can also be called *bloat*. Database maintenance on a table removes bloat, attempting to reclaim the space consumed by dead rows. Direct Updates cause bloat and Table Copying doesn't.

For Direct Updates, you'll run the VACUUM and REINDEX maintenance commands after scrubbing.

A goal of a the Performance database scrubbing system is that it runs fast. With a fast running process, your team will have an easier time automating the steps and repeating the whole process periodically.

To avoid the slowdown caused by Direct Updates with large tables, you'll use a second technique called Table Copying for very large tables with millions of rows. Table Copying uses a trick to replace the original table with a copy of the table and then swap the names around.

Table Copying doesn't require post-copy database maintenance either.

Table Copying also runs faster by strategically deferring some things like constraint checks. Database constraint checks from DML operations like UPDATE statements normally run on updates and provide an important data consistency benefit. Table Copying temporarily removes the constraints and then adds them back later. This speeds up the process.

You've now seen a brief comparison of *Direct Updates* and *Table Copying* but haven't yet put these into practice. First you'll need a way to scrub the sensitive text.

Starting An Email Scrubber Function

For your first scrubber function you'll scrub the value of the users table email column. This column has a varchar type and the email addresses have a variety of domains like gmail.com. Your design goal is to preserve the email domain portion because it does not uniquely identify a person, but replace the first portion before the “@” symbol.

To do this you'll create a database function called scrub_email() using built-in SQL functions. As a Rails developer you may not have written database functions before. Don't worry, you'll be gradually introduced to it.

PostgreSQL functions can be written with SQL (LANGUAGE SQL). For more advanced needs, there is a built-in Procedural Language called *PL/pgSQL*.

The procedural language is available as an extension. By running from psql \dx you'll see the plpgsql extension listed.

For SQL Functions, PostgreSQL calls these Query Language (SQL) functions.⁴ SQL Functions have some capabilities that are similar to writing Ruby methods like variable numbers of arguments and default arguments. It's worth starting out with SQL functions because they're less complicated, and then moving to PL/pgSQL if you need to set variables and have conditional branching and loops in your functions.

Start with a skeleton structure without an implementation so that you can learn the basics.

The *signature* of the function accepts a single argument. The argument must be a varchar type and return a varchar text string.

Run psql from your terminal and type out this function structure:

```
sql/scrub_email_v1.sql
CREATE OR REPLACE FUNCTION scrub_email(email_address varchar(255))
RETURNS varchar(255) AS $$
SELECT
    email_address;
$$ LANGUAGE SQL;
```

CREATE OR REPLACE FUNCTION creates a named function or overwrites one that was defined with the same name and arguments. The function accepts an argument email_address with the type varchar(255). The argument type is important because your argument will be database column values with specific types, and the column type must match the accepted argument type exactly.

This function doesn't do anything yet. Notice it just calls SELECT on your input value.

Once you've pasted your function body into psql, you can now call your function using SELECT:

```
sql/select_scrub_email.sql
SELECT scrub_email(email) FROM users;
```

Nice! That's the basics of a SQL function. You can use functions to build up SQL statements. You can even take things to another level with variables and loops.

Functions you define are source code just like your Rails application code, so you'll want to keep them in version control and manage their lifecycle.

4. <https://www.postgresql.org/docs/current/xfunc-sql.html>

You've got some options. You could keep your functions in the Rails app in a directory called `db/functions`. From there you can manage changes but the functions won't automatically be added to your database.

Since you're working with Rails, take advantage of the `fx` Ruby gem (*Versioned database functions and triggers for Rails*⁵) to help you manage the lifecycle of your functions. `fx` provides a Rails generator that generates a SQL file where you can place your function, as well as a Migration file to add it.

You don't need to run this now, but an example of the `fx` function generator is:

```
sh/rails_fx_gem_generate_function.sh
bin/rails generate fx:function scrub_email
```

`fx` provides a `create_function` method as well as Active Record Migration methods to update and remove functions.

Functions are included in the application `db/structure.sql` file after running `bin/rails db:migrate`.

The `fx` gem is already added to Rideshare so you can explore it there.

The first time it was run it created a SQL file `db/functions/scrub_email_v01.sql`. Check out the `db/functions` directory in Rideshare to see revisions to the function.

The Migration file in Rideshare looks like this:

```
ruby/migration_db_function.rb
class CreateFunctionScrubEmail < ActiveRecord::Migration[7.0]
  def change
    create_function :scrub_email
  end
end
```

Now that you have seen the basics of creating a database function and managing the lifecycle of it in a Rails app, you're ready to return to filling out the function implementation.

Implementing the Scrub Email Function

For the scrubber function implementation you'll build on top of built-in PostgreSQL functions.

You'll use PostgreSQL `SPLIT_PART()`⁶ function to split a string into parts.

5. <https://github.com/teoljungberg/fx>

6. https://www.postgresqltutorial.com/postgresql-string-functions/postgresql-split_part/

Pass an email address like `SPLIT_PART(email_address, '@', 1)` to return the first part before the @ symbol. The first part has a position of 1 so 1 was provided as the third argument.

Run the `SPLIT_PART()` example below in psql to get familiar with the function.

```
sql/select_split_part_function.sql
SELECT split_part('bob@example.com', '@', 1);
-- split_part
-- -----
--      bob
```

Calling `SPLIT_PART()` returns bob which is visible in the commented out portion. bob is the text you'll replace with your function.

How long is bob? Use the built-in `LENGTH()` function. bob has a length of 3.

```
sql/select_length_function.sql
SELECT length('bob');
-- length
-- -----
--      3
```

This is the basic process to build on top of built-in functions as needed.

Next you'll need some replacement text.

Use the built-in PostgreSQL MD5 hash string generation functions. The `MD5()` function generates 32-character strings from an input string. You could use the text you want to hash as input but MD5 hashed text is not considered secure cryptographically. The original text could be guessed by comparing hashed text with it.

For the Performance database, it's OK to lose the original values. Use the `RANDOM()` function as input to `MD5()` but coerce the input to `::text`. That provides a semi random input to the `MD5()` function that isn't connected to the original source text.

The `MD5()` function generates long strings but you want to take a slice out that matches the length of your source email address, which is typically going to be less than 32 characters for the part before the "@" symbol.

One gotcha when generating MD5 text strings and comparing the first few characters is that there can be overlaps on those first few characters. You'll want to store unique values for the email address, so use a workaround.

The workaround is to make the generated replacement at least 5 characters long. Use the built-in `GREATEST()` function to use either the length of the original email address input text or a length of 5, whichever is greater.

Use the `CONCAT()` function to assemble the string parts, crafting a full replacement email address from three parts.

In the function implementation below, everything is brought together to replace an email address. You can overwrite the earlier version that didn't have a body by copying and pasting this function into psql. If you wish to manage the function with Rideshare you may need to drop the function from psql and then `bin/rails db:reset` in Rideshare to restore the versions of the functions there.

```
sql/scrub_email_function_full.sql
```

```
-- replace email_address with random text that is the same
-- length as the unique portion of an email address
-- before the "@" symbol.
-- Make the minimum length 5 characters to avoid
-- MD5 text generation collisions
CREATE OR REPLACE FUNCTION scrub_email(
  email_address varchar(255)
) RETURNS varchar(255) AS $$
SELECT
CONCAT(
  SUBSTR(
    MD5(RANDOM()::text),
    0,
    GREATEST(
      LENGTH(
        SPLIT_PART(email_address, '@', 1)
      ) + 1, 6
    )
  ),
  '@',
  SPLIT_PART(email_address, '@', 2)
);
$$ LANGUAGE SQL;
```

Now that the function is defined, call it from a `SELECT` SQL statement as you did earlier from psql.

```
sql/scrub_email_short.sql
```

```
SELECT scrub_email('bob@gmail.com');
--      scrub_email
-- -----
--  5e9dd@gmail.com
```

You now have a mechanism to replace the sensitive parts of email addresses with scrubbed replacement text.

Try another email address that's longer and make sure the replacement text has the same length.

Run this SQL statement from psql.

```
sql/scrub_email_long.sql
SELECT scrub_email('bob-and-jane@gmail.com');
--      scrub_email
--      -----
--  39246b1ccf0a@gmail.com
```

With your scrubber function in place, the next phase in the development of your scrubber solution is performing the scrubbing across all the rows.

How might you go about that?

Understanding Table Copying Trade-Offs

From this section forward, you'll focus on scrubbing table rows using the scrubber functions you've developed, and the two scrubbing techniques called Direct Updates and Table Copying.

First let's focus on the Table Copying technique. With Table Copying, the first step is to create a new destination (or "target") table that is a clone of the original (or "source") table. The destination table will have no data rows.

Run `psql --dbname rideshare_development` to connect to the `rideshare_development` database. Run the following command to copy the `users` table structure to a new table `users_copy`.

To do that you'll run the following SQL DDL statement from `psql`. This uses the `CREATE TABLE ... LIKE` table creation form. Run this statement:

```
sql/table_copying_create_like.sql
CREATE TABLE users_copy (LIKE users INCLUDING ALL);
```

You'll now have a `users_copy` table that's a clone of the original `users` table with no data rows. Explore it with `\d users_copy`. Confirm that there are no rows by running `SELECT COUNT(*) FROM users_copy;`

The reason you didn't fill the table rows yet is very important and you'll find out why very soon.

You'll want to run these actions in Rideshare and you'll need data. If you have a messy database or want to start over, run the following commands from Rideshare to go back to a clean slate and with a significant volume of data.

To populate data you can either download the data dump as described in Chapter 1, or from an empty database you can use the built-in data generators Rake tasks in Rideshare. The example below resets the database to the original schema and uses the data generator tasks to load data.

```
bin/rails db:reset
```



```
bin/rails data_generators:generate_all
```

After running the generators above you should have at least 20,000 users.

To copy data rows, if you're unfamiliar with this technique review the SQL below. You may wish to run this and then TRUNCATE the users_copy table once you've done that, in order to test out the technique; or, you may skip running this if you're familiar with the INSERT INTO ... SELECT * FROM format being used.

The input to the INSERT statement is the output of the SELECT statement. Since the destination side table has all the database objects including indexes, constraints, defaults, triggers, and more, the insert process is slower than it would be without all of those objects. This will be significant later on!

The general pattern used for Table Copying is below. Running this is optional.

```
sql/table_copying_insert_into.sql
```

```
INSERT INTO users_copy(
  first_name,
  last_name,
  email,
  type,
  created_at,
  updated_at
)
(
  SELECT
    first_name,
    last_name,
    email,
    type,
    created_at,
    updated_at
  FROM users
);
```

If you ran the statement above in psql, you should see output something like INSERT 0 20300 indicating that around 20,000 rows were inserted.

You've now copied rows from one table to another. Rows were copied over as-is though, and no text was scrubbed.

Now you're ready to fold the scrubbing into your recipe.

As you did earlier, you'll call the scrubber functions using a SELECT statement. You'll replace the places where the table field was selected directly and instead you'll pass it through the function as input.

It's time to give that a try! Since you want to keep the users_copy table but remove the rows, from psql run TRUNCATE users_copy;.

Now you're ready to run the following SQL from psql. Note that the `SCRUB_EMAIL()` is now surrounding the `users.email` field. Type `\df scrub*` to make sure your scrubbed functions are listed.

`sql/scrubbing_on_the_fly.sql`

```
INSERT INTO users_copy(
  id, first_name, last_name,
  email, type, created_at, updated_at
)
(
  SELECT
    id, first_name, last_name,
    scrub_email(email), -- scrubber function
    type, created_at, updated_at
  FROM users
);
```

Now try comparing the email data stored for the same user in the `users` table and the `users_copy` table.

```
SELECT email FROM users WHERE id = 1;
      email
-----
Beau-Marvin-driver-0@email.com

SELECT email FROM users_copy WHERE id = 1;
      email
-----
16acaalaa55e432d69ea@email.com
```

The `users_copy` table email should be the scrubbed version!

Note that the SQL statement above has an explicit list of columns for the `users` table from Rideshare at the time of publication. Review the latest schema and make any necessary column additions or modifications to the statement by checking out the latest version of the project.

Inserting 20,000 rows with a value transformed on the fly above should have been pretty quick. You can actually speed things up further though. Speeding this process up is important when you're working with millions of rows of data.

Speeding Up Inserts For Table Copying

By using database functions inside PostgreSQL to perform scrubbing as opposed to scrubbing in client application Ruby code, you're removing the latency that's added when the client application communicates with the PostgreSQL server.

Although running the process above is fast, it can be made even faster. To do that, you'll use some tricks that are known in PostgreSQL to speed up the insert rate by taking away some of the database objects.

A note upfront though. These tricks speed up the Insert rate but add significant complexity to the process. For your needs try starting out with table copies that have all the database objects and run your scrubbing process to see if it's fast enough. If it's very slow, then proceed with these tricks, and understand you'll be putting in some more work to speed the process up.

Two of the tricks to speed up inserts are to remove database constraints and indexes on the destination table. After copying is completed you'll add them back. This can be a lot of direct manipulation and table modifications, which is a lot to learn! Don't worry, you'll conduct this work on your local PostgreSQL database and gradually build up your comfort with these techniques.

Open up the Rideshare database with `psql`. Drop the `users_copy` table since you're going to create it again. Earlier you created the `users_copy` table using the `INCLUDING ALL` keywords but you'll go in a different direction this time.

This time use the keywords `INCLUDING ALL EXCLUDING INDEXES`.

Run these statements:

```
sql/table_create_like_including_all_excluding_indexes.sql
DROP TABLE IF EXISTS users_copy;

CREATE TABLE users_copy (LIKE users INCLUDING ALL EXCLUDING INDEXES);
```

Explore the new version of `users_copy` with `\d users_copy`. This version has many of the database objects before except the table Indexes are missing and the Primary Key constraint on the `id` column is not there either.

With indexes on the table, each table write needs to be maintained in any Indexes covering columns being inserted. Without the indexes this index maintenance does not happen and inserts are faster. Later on you'll see how to run `CREATE INDEX` statements to recreate them all.

Run the "`scrubbing_on_the_fly.sql`" Insert statement again on the new `users_copy` table. If you've enabled `\timing` in `psql`, study the differences. Without indexes the newer version should be faster.

Next you'll consider what's missing in order to make sure a replacement table retains all of the original objects.

Three things you'll need to bring forward to the destination table are the Foreign Key Constraints, the Primary Key constraint and Sequence ownership and assignment, and the Indexes.

First, focus on the Foreign Key Constraints. You haven't worked yet with Foreign Key Constraints much. For now just consider them as database objects that you're bringing from the original table to the destination table.

The users table does not have Foreign Key constraints on the table so you'll use a different table from Rideshare to demonstrate how to handle this.

From psql, run the following SQL statement to list all Foreign Key Constraints in the public schema, taking note of the ones that are listed for the trips table. trips is another Rideshare table. This query uses the pg_constraint⁷ System Catalog:

```
sql/list_all_constraints.sql
-- list constraints in 'public' schema
SELECT
    conrelid::regclass AS table_name,
    conname AS foreign_key,
    pg_get_constraintdef(oid)
FROM pg_constraint
WHERE contype = 'f'
AND connamespace = 'public'::regnamespace
ORDER BY conrelid::regclass::text, contype DESC;
```

From the trips table, select the Foreign Key Constraint named fk_rails_e7560abc33. This constraint name was generated by Active Record and the name may change. Refer to the latest copy of Rideshare if it's changed. The constraint covers the driver_id field on the trips table referring to the users.id column. Remember that the users table is polymorphic and holds both Drivers and Riders.

To restore the constraint, if you were scrubbing the trips table, you'd use the query result above and then duplicate the constraint on the users_copy table with the exact same definition and name.

Creating Constraints manually can get very tedious, though, when there are dozens of constraints to create.

Is there any easier way? Yes! A better technique is to query within PostgreSQL itself to get the constraint definition DDL. You can format the constraints in the form of a creation DDL statement and make it easier to copy and paste in creation statements.

7. <https://www.postgresql.org/docs/current/catalog-pg-constraint.html>

To do that, run psql and enter the following query. Supply the constraint name from above. The result will be the constraint definition, showing which table and columns it covers.

```
sql/constraint_definition_ddl.sql
```

```
SELECT
  connamespace::regnamespace AS schema,
  conrelid::regclass AS table,
  conname AS constraint,
  pg_get_constraintdef(oid) AS definition,
  FORMAT(
    'ALTER TABLE %I.%I ADD CONSTRAINT %I %S;',
    connamespace::regnamespace,
    conrelid::regclass,
    conname,
    pg_get_constraintdef(oid)
  )
FROM
  pg_constraint
WHERE
  conname IN ('fk_rails_e7560abc33');
```

Copy the value from the rightmost definition column. You'll need a trips_copy table to test this on, similar to the users_copy table above. Run the following statements to create a trips_copy table, and then run the ALTER TABLE statement below to add the Foreign Key Constraint.

```
CREATE TABLE trips_copy (
  LIKE trips INCLUDING ALL EXCLUDING INDEXES
);

ALTER TABLE public.trips_copy
ADD CONSTRAINT fk_rails_e7560abc33 FOREIGN KEY (driver_id)
REFERENCES users(id);
```

You've now seen one way to add the constraint to the destination table. You can DROP TABLE trips_copy; to clean things up.

This is how you'll handle Foreign Key Constraints for columns. Next you'll consider how to handle Sequences used for the Primary Key column. Note that this table is not using an Identity column (See: *Serial type versus Identity columns*⁸) and is using a Sequence for Primary Key column values.

The users table id column is the Primary Key and has a unique value enforced by the Primary Key Constraint, handed out from a Sequence. The Sequence is another database object and has the name users_id_seq which follows the conventional naming format for Sequences from Active Record tables.

8. <https://wanago.io/2022/02/21/serial-type-identity-columns-postgresql-typeorm/>

Just like with Foreign Key Constraints you may list out all the Sequences in the database. Run the following “List all Sequences” SQL query⁹ from psql to list all Sequence objects.

```
sql/list_sequences_table_column_owner.sql
```

```
SELECT
    s.relname AS seq,
    n.nspname AS sch,
    t.relname AS tab,
    a.attname AS col
FROM
    pg_class s
JOIN pg_depend d ON d.objid = s.oid
AND d.classid = 'pg_class'::regclass
AND d.refclassid = 'pg_class'::regclass
JOIN pg_class t ON t.oid = d.refobjid
JOIN pg_namespace n ON n.oid = t.relnamespace
JOIN pg_attribute a ON a.attrelid = t.oid
AND a.attnum = d.refobjsubid
WHERE
    s.relkind = 'S'
    AND d.deptype = 'a';
```

Since there is no activity on the users table right now and since you’re working on a replacement table called users_copy, you’ll change the ownership of the sequence so that it’s owned by the users_copy table. This is because eventually the users_copy table will take over for users.

Warning: you wouldn’t want to modify the sequence this way in a live system that was being used by a users table creating new rows. It’s ok here though. The Sequence must be in the same Schema as the table and the table owner must be the same. users_id_seq is in the Public schema. Check the owner of users and users_copy with the \dt meta command. If the owners are different, to change the owner of users_copy to postgres, for example, run this statement:

```
ALTER TABLE users_copy OWNER TO postgres;
```

Once the owner is the same for both users and users_copy, run the following statement to set the ownership for the sequence.

```
ALTER SEQUENCE users_id_seq OWNED BY users_copy.id;
```

The sequence is now in place for the users_copy table. You’ve prepared the Foreign Key Constraints and the Primary Key Sequence for the upcoming planned table switch. The last missing item to bring forward are the Indexes from the users table.

9. https://github.com/andyatkinson/pg_scripts/blob/master/list_all_sequences.sql

To see the Indexes for a table from psql, use the \d users meta command. The indexes are listed at the bottom of the output.

Since you want to create the indexes and not just view their definition, you'll use the same trick as before to list the indexes in a DDL creation style, making it easier to create them.

Run the following SQL query to list the table indexes in this way.

```
sql/list_users_table_indexes.sql
SELECT pg_get_indexdef(indexrelid) || ';' AS index
FROM pg_index
WHERE indrelid = 'public.users'::regclass;
```

At publication time for Rideshare there are four indexes on the users table. Take a look at the Unique index on the email column.

```
CREATE UNIQUE INDEX index_users_on_email ON public.users USING btree (email);
```

The table name is “schema qualified” and shows the public schema name. Change the table name from users to users_copy. You'll also need to make the name unique since index_users_on_email already exists for the users table. Make it unique by adding “2” on the end. Real creative right?

Run the statement to add the Index. Repeat these general steps for the other three indexes.

You've now brought the Foreign Key Constraints, Primary Key column Sequence, and Indexes forward. The users_copy table can now take over the role of the users table. The last step in this process is to rename the users_copy table to users. You can perform this Drop and Rename inside a transaction and it even has a cool name: the “swap and drop.”

From psql, run the following SQL statements to DROP the source table and rename the destination table.

```
sql/finalize_table_copying_users.sql
BEGIN;
-- drop the original table and related objects
DROP TABLE users CASCADE;

-- rename the destination table to be the source table name
ALTER TABLE users_copy RENAME TO users;
COMMIT;
```

Note that CASCADE is used to drop all related objects. Rideshare has more objects connected to the users table, like Views and Materialized Views, that are being discarded with CASCADE. Those object types will be covered later. For

now it's OK to remove them. You can always reset the Rideshare schema from scratch to `bin/rails db:reset` which restores the Views.

One other note is that you want to rename the indexes back to their original name. For example, from `psql` you may run `ALTER INDEX index_users_on_email2 RENAME TO index_users_on_email`; if you'd added a "2" on the end earlier.

Let's take stock of where things are. You created a full copy of the users table including all the original database objects. The rows are all the same apart from the scrubbed columns.

From the perspective of the Rails application, the same table exists with the same schema, so it's all good, and it will work as-is with the Driver and Rider model classes. In a real application you wouldn't even need to restart it because the schema has not changed, although restarting is always a good idea for these types of changes to rule out cache issues.

You've now seen the Table Copying process and run through it end to end, to scrub a single column, and have an understanding of how to speed up the process if you'd like.

Moving all of the database objects around is a lot of work.

For smaller tables, directly updating rows will be a more straightforward technique.

Using Direct Updates For Text Replacement

Direct Updates are simply SQL `UPDATE` statements applied to a table like users, using the scrubbed functions you developed earlier.

From `psql`, run the following `UPDATE` statement to scrub the email address column for every row. Note that this mutates your users row data you'd generated earlier. This can be called a "destructive" modification compared with the Table Copying approach where you copied rows and mutated a copy of each row. That's perfectly fine here because you can always reset and generate new data if you wish, but it's worth noting the destructive aspect of the `UPDATE` operation.

```
sql/direct_updates_users.sql
UPDATE users
SET email = scrub_email(email);
```

Running the statement above should show something like below, indicating all rows were updated.

```
UPDATE 20300
```


If you were running this on hundreds of thousands of rows, you may wish to update a batch of, say, 10,000 rows at a time. Working on batches of rows will be covered later on. For now, consider how the Direct Updates technique can be as simple as a single statement for all rows.

Compared with Table Copying, you don't need to worry about Foreign Key Constraints, Sequence modifications, or Index creations.

However, to restore optimal performance, when using Direct Updates you'll want to perform some post-update database maintenance.

What does that look like?

Performing Database Maintenance

Because you've updated every row, a new row version has been created and taken over. Remember that row versions are immutable in PostgreSQL. The former version is now a "dead" row. The space consumed by the dead rows can be reclaimed. Normally the *Autovacuum* process helps keep things tidy and runs automatically for you. Autovacuum is a background scheduler system that runs a VACUUM operation for each of your tables.

For manual operations like Direct Updates where you're modifying many rows, you may wish to immediately perform an equivalent operation to restore optimal performance. As the PostgreSQL operator you're able to run a manual VACUUM on a table at any time, and this is safe to do.

To run a manual VACUUM, from psql run this statement:

```
VACUUM (ANALYZE, VERBOSE) users;
```

This command adds the ANALYZE and VERBOSE arguments.

With ANALYZE you're also updating the statistics that are collected for the table; PostgreSQL keeps track of some of the properties of the data in the table when the ANALYZE command runs.

The VERBOSE option adds much more output about what's happening, listing the number of pages being scanned, live rows, and dead rows.

Another important maintenance operation after a big update is to refresh Indexes for the table. This operation is called *Reindexing* and the Index is actually rebuilt instead of being updated directly. To rebuild indexes you'll use the REINDEX command.

For PostgreSQL version 12, the REINDEX command gained a new option called CONCURRENTLY. This is very significant because it makes it possible to rebuild

indexes online without stopping PostgreSQL, which is critical for a running application.

For the Performance database, it's being refreshed offline. The REINDEX command can be run without CONCURRENTLY because there are no other concurrent users of the indexes. Without that option the REINDEX operation will run faster.

Although it may not be strictly necessary, practice rebuilding the index `index_users_on_email` for the `users` table by running this statement from `psql`:

```
REINDEX INDEX CONCURRENTLY index_users_on_email;
```

You've now performed VACUUM, ANALYZE, and REINDEX operations following a big update.

With those maintenance operations, the `users` table is optimized again for use by the application.

Performing a single UPDATE statement on a large table with hundreds of thousands of rows may not be feasible. Instead you will likely want to break up the updates into batches. How might you do that?

Performing Modifications In Batches

You've written *Functions* in SQL and learned they can be written with the PL/pgSQL Procedural Language.

PostgreSQL offers another capability that extends functions further, called *Procedures*, introduced in version 11. You'll use a Procedure to perform batched updates, calling a function within the Procedure implementation body.

Procedures can be used to provide control flow constructs like looping. As a Ruby programmer, you've written a lot of loops before, but you may not have written loops in PostgreSQL code.

Below you'll find a Procedure that takes the Direct Updates UPDATE statement approach and breaks it up into batches.

The Primary Key `id` column is used to select with for a range of values. The variable for controlling the batch size can act as a configuration point depending on whether your server is capable of more updates at a time or fewer. The default size is 1000. You may wish to increase this to 10000 or more if you have a big server.

Read through the implementation. The details will be described in more detail below.

```
sql/scrub_batched_direct_updates.sql
```

```
CREATE OR REPLACE PROCEDURE scrub_batches()
LANGUAGE plpgsql
AS $$
DECLARE
    current_id int := (SELECT MIN(id) FROM users);
    max_id int := (SELECT MAX(id) FROM users);
    batch_size int := 1000;
    rows_updated int;
BEGIN
    WHILE current_id <= max_id LOOP
        -- the UPDATE by `id` range
        UPDATE users
        SET email = scrub_email(email)
        WHERE id >= current_id
        AND id < current_id + batch_size;

        GET DIAGNOSTICS rows_updated = ROW_COUNT;

        COMMIT;
        RAISE NOTICE 'current_id: % - Number of rows updated: %',
            current_id, rows_updated;

        current_id := current_id + batch_size + 1;
    END LOOP;
END;
$$;

-- Call the Procedure
CALL scrub_batches();
```

You may paste this procedure directly into psql for Rideshare, and note that it will run right away, updating batches of users rows.

A WHILE loop is being used which is similar to a While loop in Ruby. It will run until a condition is met.

The GET DIAGNOSTICS rows_updated = ROW_COUNT; clause captures the number of rows being updated in one loop iteration. The UPDATE statement updates all of the rows that are matched inside of the id column range. This will update 1000 rows with a batch size of 1000.

Values are printed out in a similar way to puts in Ruby using the RAISE NOTICE clause. The % symbol here is similar to String interpolation in Ruby.

Procedures can be copied directly into psql to run. However, similar to database functions, you'll want to treat procedures you use with your appli-

cation as source code and make sure they're added and managed in version control.

By pasting in the CREATE PROCEDURE block it becomes defined but it's not yet called. The last line `CALL scrub_batches();` is what calls the procedure.

If you ran the procedure you should see similar output to what's below. The `current_id` is printed starting from the `min(id)` value and starts updating rows until the `max(id)` is reached.

```
NOTICE: current_id: 1 - Number of rows updated: 1000
NOTICE: current_id: 1002 - Number of rows updated: 1000
NOTICE: current_id: 2003 - Number of rows updated: 1000
```

You've now seen a technique to break up the single UPDATE from earlier into batches of updates.

What's Next For Your Performance Database

In this chapter you've seen how you could build a scrubber system for your production data. Creating a sanitized copy of your production database unlocks Performance testing and trialing of destructive modifications for your team.

You compared Direct Updates and Table Copying strategies to perform the scrubbing process, reviewing the tradeoffs with each technique. You may wish to use a hybrid strategy with Direct Updates where Constraints and Indexes are temporarily removed and then brought back.

A Performance database like this could be part of a dedicated performance testing environment, or act as the data in your pre-production Staging environment. By running the database on an instance with similar power to your production instance, you'll have a great simulation environment your team can work with safely.

In the next chapter you'll begin working with database structure changes you make as you evolve your application schema, and how to do that safely.

In the old days, some types of structure modifications might have required downtime. In modern companies, stopping the database and application comes at a high cost as is to be avoided. You'll want to learn how to make structure changes while the database is serving other users, and what's safe and unsafe to do. The Performance database running in an environment with concurrent activity can help you out there too.

To get started, you'll first need to understand what makes a structural change safe or potentially unsafe.

Data Correctness and Consistency

Now you're ready to learn how to use Constraints to improve your data layer. PostgreSQL offers a variety of constraint types and options that can help you make sure your data is correct and consistent. These capabilities are up to you to implement.

The general technique you'll follow is to add Constraint objects at various levels such as at the table level or column level. In most Ruby on Rails applications, you'll also validate data at the application layer using the Active Record Validations framework.

You'll compare Database level and Application level constraints and validation approaches to better understand their trade-offs.

To help understand what consistent and valid data looks like, you'll start with examples of invalid and inconsistent data. The first example you'll look at is when data violates *Referential Integrity*. Referential Integrity refers to data in two tables with a referring or referenced relationship, and making sure both sides are intact.

When Referential Integrity is violated it means that one half of the relationship is missing. Referential integrity violations are a common type of application bug.

To begin enforcing referential integrity you'll add Foreign Key constraints. Foreign Key constraints work together with Primary Key columns on another table and this is the mechanism you use to get PostgreSQL to enforce the relationship.

In order to add a validated constraint, there must be no pre-existing violations. You'll see ways to work around this by modifying this behavior. Referential integrity can also be enforced at the application layer in Active Record model code.

Database constraints improve your data model but there are operational challenges to adding them to production databases. You'll learn about those challenges and how to solve them.

Correctness Terminology



- Referential Integrity — Ensuring rows in two tables with common values both exist
 - Constraints — Database object used to constrain data in a field or between tables
 - Enum — Database list object to constrain possible field values
 - Domain — Like an enum but can be shared across tables
 - Correctness — Ensuring that data is consistent with the constraints
-

PostgreSQL provides a variety of constraint types. They are listed below and will be covered in greater detail as the chapter unfolds.

- PRIMARY KEY
- NOT NULL
- UNIQUE (covered in [on page 60](#))
- REFERENCES (Foreign Key)
- CHECK
- EXCLUDE (Exclusion)

First up is the UNIQUE constraint.

Multiple Column Uniqueness

Understanding which fields or attributes should be unique is one of the interesting challenges in data modeling.

Some tables start out with one field defining uniqueness, but then the definition changes over time. For example a City name might be unique for a single state, but exist as a name in many states when all States in the country are considered. Within the United States a city named Portland could refer to Portland, Maine or Portland, Oregon. Another example is Hollywood, which probably refers to Los Angeles but could refer to Florida. To express a unique city name you might want to define uniqueness as the combination of city name and state name.

In Rideshare there is a locations table with city and state columns. The table has a UNIQUE constraint on the address column.

You'll add a new unique constraint that defines uniqueness on both City and State.

A UNIQUE constraint is added using an ALTER TABLE statement with the ADD CONSTRAINT keyword. PostgreSQL automatically adds an index when you add a constraint, to allow a fast lookup when enforcing the constraint.

Since PostgreSQL adds an index anyway, adding a Unique Index is a common way to add a Unique constraint. For a running system, adding the constraint via the Unique index makes it possible to add the index CONCURRENTLY and avoid a long exclusive lock of the table where the index is being added. This is a good design goal because a long-held lock might block other queries trying to write and read to the table.

If you're a Rails developer you've added Unique indexes before, so you can skip this refresher section. If you're newer to Rails, the conventional way to add a Unique index would be to use the CONCURRENTLY keyword and add these changes to a new Active Record Migration.

From your terminal, generate a new Migration by running `bin/rails generate migration` and give it a name. Below, "g" is short for "generate". Generation in Rideshare may fail because this Migration exists already.

```
bin/rails g migration AddLocationsCityStateUnique
```

Open the generated file and fill in the `add_index` method, setting the `unique` option and `concurrently` options as follows. This migration was added to Rideshare earlier.

```
class AddLocationsCityStateUnique < ActiveRecord::Migration[7.1]
  disable_ddl_transaction!

  def change
    add_index :locations, [:city, :state],
      unique: true,
      algorithm: :concurrently
  end
end
```

The migration was applied running `bin/rails db:migrate`. UNIQUE constraints are added smoothly when there aren't existing violations and there are multiple rows with the same combination of city and state.

Often when redefining uniqueness things don't go so smoothly!

When rows with duplicate values exist, PostgreSQL won't allow the constraint to be added. Thanks to Transactional DDL, the Constraint creation DDL rolls

back smoothly. Before it can be added again the violations need to be fixed up.

How can you do this?

Fixing Constraint Violations

When a constraint is added, like NOT NULL or FOREIGN KEY, all existing rows are checked. Constraint violations prevent their creation. This is true for adding all constraint types.

How can you ensure there are no violations in advance?

To do that, you'll need to query for violations and identify their presence or absence. When dealing with millions of rows it can be a pain to validate all of them.

One technique from the application level would be to iterate through rows in batches and look for problematic values. However, this process can be slow. Querying large data sets is a task PostgreSQL is designed for, and this discovery work can be done in the database.

In the SQL For Devs post “Delete Duplicate Rows”,¹ the author describes two methods to identify duplicates and delete them in one query.

In the simpler approach, rows are queried and grouped by the columns that make them unique, and then rows greater than the MIN(id) or less than the MAX(id) are deleted. With the duplicates removed, the constraint can be added.

In the more complex form, a couple of advanced concepts are used to add flexibility over which results to delete. This technique uses a Common Table Expression (CTE) for the main query and all rows get a value from the ROW_NUMBER() *Window Function*² with the PARTITION BY option. This partitions (or “segments”) the results based on the columns that define uniqueness. CTEs and Window Functions are described only briefly for now.

What's happening is that each row in the results gets a row number. The rows can be ordered in different ways to get a different row number. The rows are deleted based on a row number condition, such as rows numbered > 1.

You've now seen how a multi-column definition of uniqueness can be added to a table, and how to discover duplicates and remove them in advance.

1. <https://sqlfordevs.com/delete-duplicate-rows>

2. <https://www.postgresql.org/docs/current/tutorial-window.html>

What about data that's spread across multiple tables? This is where FOREIGN KEY constraints can help you keep data relationships intact.

Enforcing Relationships With Foreign Keys

The paper *Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity*³ covers how Active Record models are linked together using Associations and Validations within the Ruby application code layer.

Active Record uses association methods like `has_many` and `belongs_to` to express relationship types like one to one, and one to many. Validations are added to models to enforce validation that there aren't NULL values, and for many other things.

When a database table row depends on the existence of a row in another table, a column can be used to store a reference to the Primary Key id value of the other table being referred to.

That column can optionally have a Foreign Key constraint added to it. Trying to delete either row when a Foreign Key constraint is defined will be prevented by PostgreSQL.

Foreign Key constraints are not added by default so it's up to developers to add them.

For many years Rails did not support database Foreign Key constraints. This might be why they had less usage by Rails developers. Although support was added with a gem, native support for database Foreign Keys was added to Rails Version 4.2 released in 2014.

As a refresher, let's review FOREIGN KEY constraints⁴ in the Rideshare database.

A Rider creates a Trip Request. When a Driver accepts a Trip Request a Trip record is created. Trips are linked to Trip Requests.

Since a Trip depends on a Trip Request, the `trip_request_id` is a Foreign Key column and has a NOT NULL constraint and Foreign Key constraint. Trip Requests are also linked with Riders, and Trips are linked with Drivers.

The Migration in Ruby turns into a SQL statement similar to the ALTER TABLE DDL SQL below. You don't need to run this statement since this Foreign Key constraint already exists in Rideshare.

3. <http://www.bailis.org/papers/feral-sigmod2015.pdf>

4. https://guides.rubyonrails.org/active_record_migrations.html#foreign-keys

```
sql/foreign_key_constraint_trips.sql
```

```
ALTER TABLE ONLY trips
ADD CONSTRAINT fk_trips_trip_requests
FOREIGN KEY (trip_request_id)
REFERENCES trip_requests(id);
```

Duplicate Constraints and Indexes



Duplicate constraints and Indexes can be added that cover the same columns as long as they have unique names. Be careful not to add duplicate Constraints or Indexes because these are consuming space, slowing down write operations. A Duplicate constraint or Index is redundant and one of them should be removed.

You've now covered how to use Foreign Key Constraints to link data from multiple tables together. Next you'll work with a less common constraint type called a CHECK constraint. This versatile constraint type allows you to capture conditional statements in their definition.

The Versatile Check Constraint

With a CHECK constraint you can go beyond checking for NULLs and connecting Primary Key and Foreign Key columns, and write a custom expression.

This constraint type gives you versatility in what expressions you write to help you enforce correctness in your data.

CHECK constraints are owned by a table. For Rails developers, if you've never seen a Check constraint, think of them as similar to Active Record Model Validations. Validations have some overlap in their purpose. Validations are created in Active Record models while Check constraints are created within the database for a table.

Model Validations express conditions that must be met for data to be considered valid.

Like all constraint types, Check constraints can be used on their own or in tandem with Active Record Validations.

Besides their primary purpose of enforcing correctness, Check constraints have another very useful purpose. Check constraints can be added as a placeholder due to their flexibility.

Some constraints require a table lock to be added without the option to check only new row modifications. Check constraints can be used to work around that. What does that look like?

When adding UNIQUE or NOT NULL constraints to a table, all rows are checked immediately and the check process cannot be configured only for new modifications like new inserts, updates, or deletes.

On a large and busy database this can make it difficult to add UNIQUE and NOT NULL constraints since it might take a long time to check all of the rows. While all of the rows are being checked, the table is locked and blocking other writers and readers.

Those blocked queries will start to show up as application errors and cause user experience problems and possibly data loss. Thus adding NOT NULL and UNIQUE constraints to a large running database is considered a potentially unsafe operation. Is there a way to work around this? Check constraints to the rescue!

Instead of directly adding NOT NULL and UNIQUE constraints, first you'll add a CHECK constraint and mark it NOT VALID. Note that Foreign Key constraints also support the NOT VALID option.

To practice adding Check constraints, you'll add some to Rideshare. The examples were inspired by *Postgres Constraints for Newbies* from Crunchy Data.⁵

You'll add Check constraints to the Vehicle Reservations table.

To get started, take a look at the Check constraints on the trips table.

A Trip is completed and completed_at is set. This time value should always be *after* the time the trip was created, which is stored in the created_at column. This is enforced with a Check constraint in Rideshare.

Review the Migration to see the Check constraint definition, or review the equivalent SQL below.

Rails 6.1 natively supports Check constraints in Active Record syntax.⁶ Support for if_exists⁷ was added later.

[sql/check_constraint_trips.sql](#)

```
ALTER TABLE trips
ADD CONSTRAINT trips_completed_after_created
CHECK (completed_at > created_at);
```

An equivalent Active Record Migration would look like this:

5. <https://www.crunchydata.com/blog/postgres-constraints-for-newbies>
6. <https://blog.saeloun.com/2021/01/08/rails-6-check-constraints-database-migrations>
7. <https://github.com/rails/rails/commit/25f97a66bdae6efe788b2d0ab7ab9cef6fc5a23a>

```
ruby/migration_trips_check_constraint.rb
```

```
class AddCheckConstraintTripsCompletedAt < ActiveRecord::Migration[7.0]
  def change
    add_check_constraint :trips, "completed_at > created_at",
      name: "trips_completed_after_created"
  end
end
```

In the same way as the trips table, the Vehicle Reservation start and end times should make sense chronologically where the ends_at time should always be *after* the starts_at time.

This Check constraint for Vehicle Reservations has not been added to Rideshare. See if you can add it to your copy! You could even fork the Rideshare repo and continue to evolve your own database schema.

If there are rows that violate the constraint, PostgreSQL will display an error message:

```
sql/error_constraint_violation.sql
```

```
ERROR: check constraint "trips_completed_after_created"
of relation "trips" is violated by some row
```

You've now seen how Check constraints can help make sure your timestamps are in the correct order. This might help you avoid a data consistency bug where the front-end client application posts invalid data, by making sure the data is rejected.

These are relatively straightforward constraints, but what about one that's a bit more complicated?

Imagine that Vehicle Reservations must be a minimum of 30 minutes in length.

Since having a reservation that's too short is a significant business problem for Rideshare, you decide to create a Check constraint to help enforce minimum reservation lengths.

One strategy with a SQL expression in a Check constraint definition is below. This builds on what you did earlier, making sure the ends_at column is at least 30 minutes after the starts_at column value. Both of these columns prevent NULLs.

Put this SQL statement into psql for the Rideshare database:

```
sql/check_constraint_reservation_length.sql
```

```
ALTER TABLE vehicle_reservations
ADD CONSTRAINT vehicle_reservation_minimum_length
CHECK (ends_at >= (starts_at + INTERVAL '30 minutes'));
```

Earlier you learned that constraints can be NOT VALID. This is a way to enforce the constraint for new row modifications, but not for existing rows that aren't being checked.

Constraint enforcement can also be deferred which is slightly different. What does that mean?

Deferring Constraint Checks

PostgreSQL supports deferring constraint checks within a transaction. UNIQUE, PRIMARY KEY, Foreign Key (REFERENCES), and Exclusion (EXCLUDE) constraints are *deferrable* although they are all checked immediately by default. This means that if you wish to make the constraint deferrable, you'll need to create it that way from the beginning or drop it and create it again with the DEFERRABLE option enabled.

Foreign Key and Exclusion constraints can have their deferrable behavior modified using the SET CONSTRAINT⁸ operation within a transaction.

Where can this be used? In the Hashrocket post *Deferring Constraints in PostgreSQL*,⁹ the author walks through a use case for deferring constraints in a Rails app.

Items in a list have unique position values. To enforce unique position values a UNIQUE constraint is used on the position column. Without deferring the constraint enforcement, successive UPDATE calls would duplicate positions as list items are re-ordered, which would violate the Unique constraint definition.

To work around this, UPDATES must be made within a transaction, and the Unique checks are delayed until the transaction commits. This means it's possible to create duplicate position values, basically ignoring the Unique constraint, within the scope of the transaction before it's committed. They do need to all be unique by the time of commit, though.

Once all the positions are swapped around and unique again, the transaction is committed and constraint enforcement happens at that time.

By defining the constraint as DEFERRABLE INITIALLY DEFERRED it's possible to change the default behavior from being checked immediately to being checked at the end in a transaction.

Using that option would make the behavior described above the default behavior. If you realize later that this option would be better as the default

8. <https://www.postgresql.org/docs/current/sql-set-constraints.html>

9. <https://hashrocket.com/blog/posts/deferring-database-constraints>

but the constraint was not defined that way initially, you'll need to drop the constraint and create it again using that option.

Next, you'll try out a less common but powerful constraint type called the Exclusion (EXCLUDE) Constraint. The big difference with this type is that it's used to constrain data across multiple rows. How does that work?

Preventing Overlaps With An Exclusion Constraint

Instead of a condition to enforce for a single row, an Exclusion constraint works at the level of multiple rows and can be used to express conditions about when they shouldn't match or overlap.

In Rideshare, Vehicle Reservations should not overlap. Two Riders should not be able to reserve the same vehicle at the same time. Typically this business logic will be written in the application. However this is mission critical business logic and you've decided you'd like to enforce it within PostgreSQL.

Overlapping Vehicle Reservations would cause a very bad experience for each person involved. Not only would the reservation be impossible to fulfill for both people simultaneously, it would be really unclear which reservation to honor. The first one? The most recent one? Someone is going to be upset.

That's something you'd definitely want to avoid.

Take a moment to think through how to define this constraint. Only one Vehicle should be reserved by a Rider at a time. You'll use an Exclusion constraint (EXCLUDE) to detect when an overlap would occur and the constraint will prevent the overlap.

To create the Exclusion constraint you'll use a GiST index. You haven't yet worked with GiST indexes; for now it's enough to know that they are the Index type you need for your Exclusion constraint.

The SQL statement to add this constraint type is listed below. This Exclusion constraint has already been added to Rideshare, so you may explore the definition by running `\d vehicle_reservations` from `psql`. Each of the pieces will be explained in the next section.

```
sql/exclusion_constraint.sql
```

```
ALTER TABLE vehicle_reservations
DROP CONSTRAINT IF EXISTS non_overlapping_vehicle_registration;

CREATE EXTENSION IF NOT EXISTS btree_gist;

ALTER TABLE vehicle_reservations
ADD CONSTRAINT non_overlapping_vehicle_registration
EXCLUDE USING gist (
```

```

    vehicle_id WITH =,
    tstzrange(starts_at, ends_at) WITH &&
  )
WHERE (not canceled);

```

Let's break down what's happening in this Exclusion constraint.

The constraint is named “non_overlapping_vehicle_registration”. The btree_gist extension is enabled because a GiST index is needed.

Identical vehicle_id values are selected using the WITH = clause and operator.

For the same vehicle_id, the starts_at and ends_at column values are compared using the WITH && clause and operator.

The && operator is a special one that identifies overlaps. Overlaps are elements that are in common.

The last piece part of the SQL statement is a WHERE clause with the WHERE (not canceled) condition. This clause is important because it excludes canceled Vehicle Reservations from this constraint definition. You'd want to allow overlapping Vehicle Reservation for reservations that have been canceled so that the business can collect revenue during that time.

And there you have it!

The Exclusion constraint is a more complicated but expressive constraint type that helps you prevent bad data from entering your system.

In the next section, you'll switch gears away from database constraints and move into the Active Record application layer.

Active Record provides a powerful and flexible set of built-in Validation types. Besides the built-in types, the system is extensible allowing you to write Custom Validators. How might you go about that?

Creating Active Record Custom Validators

Besides the built-in Active Record validations, Active Record supports Custom Validators.

Custom Validators¹⁰ are created using object oriented design concepts.

A Ruby class that extends the ActiveRecord::Validator class and implements a validate() instance method is a Custom Validator. The .validate() method signature accepts a single record argument.

10. https://guides.rubyonrails.org/active_record_validations.html#performing-custom-validations

Individual attributes can be validated by extending the `ActiveModel::EachValidator` class and by implementing the `.validate_each()` method. This method has a signature of `validate_each(record, attribute, value)`.

Rails Guides cover Custom Validators in the section *Performing Custom Validations*. The email validator from Rails Guides has been copied below.

To use the validator, the `email: true` configuration is added to a model attribute that you want to validate. Model attributes support multiple validations, allowing you to add both built-in ones and Custom Validators.

The `EmailValidator` class below has been added to the Rideshare codebase and is in use on the User model email field.

```
ruby/email_validator.rb
class EmailValidator < ActiveModel::EachValidator

  EMAIL_REGEXP_FORMAT = /\A([^\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i

  def validate_each(record, attribute, value)
    unless value =~ EMAIL_REGEXP_FORMAT
      record.errors.add(
        attribute,
        (options[:message] || "is not an email")
      )
    end
  end
end
```

After studying the Validator class, which class it extends, methods it implements and the signature of the method, it's time for you to create one.

Imagine you're developing a new feature for Rideshare that accepts a Driver's License number from the State of Minnesota in the United States. The number is accepted as user input from a form. Driver's Licenses in the United States have different formats in each state and some states share their formats.

According to *U.S. Driver's License Numbers*,¹¹ Minnesota Driver's License numbers have one letter and 12 numbers. For example, P800000224322 starts with the letter "P" and then has 12 numbers.

A Regular Expression of `[a-zA-Z]\d{12}` is even provided on the website for reference. This Regular Expression will be useful for your custom validator.

Using what you learned in the Email Validator example above, try and create a Customer Validator for the Drivers License Number format. A starting point is listed below.

11. <https://success.myshn.net/Data-Protection/Data-Identifiers/U.S.-Driver%27s-License-Numbers>

```
class DriversLicenseValidator < ActiveRecord::EachValidator
  def validate_each(record, attribute, value)
    end
end
```

A completed Validator for validating the format of Drivers License Numbers has been added to the Rideshare codebase in the `app/validators` directory:

```
ruby/drivers_license_number_validator.rb
class DriversLicenseValidator < ActiveRecord::EachValidator

  DL_MN_REGEXP_FORMAT = /[a-zA-Z]\d{12}/i
  DEFAULT_MESSAGE = "is not a valid driver's license number"

  def validate_each(record, attribute, value)
    unless value =~ DL_MN_REGEXP_FORMAT
      record.errors.add(
        attribute,
        options[:message] || DEFAULT_MESSAGE
      )
    end
  end
end
```

Drivers are required to have a Driver's License and their license number should be unique. Use the built-in presence and uniqueness validations in addition to the Custom Driver's License Number Validator for the `drivers_license_number` field.

The Driver model in Rideshare combines these validators:

```
ruby/driver_model.rb
class Driver < User
  validates :drivers_license_number,
    presence: true,
    uniqueness: true,
    drivers_license: true
end
```

You've now seen how multiple Active Record validations can be used together. Pretty slick. You've made sure the data is populated, unique, and matches an expected format.

Although these rules are rigid; at other times you may want flexibility in how you accept input from the user.

One example is case sensitivity vs. insensitivity. Email addresses are not case sensitive. However, email addresses may be entered as user input using specific casing and you may wish to preserve that casing for presentation

purposes. You may have the goal simultaneously of ignoring the casing for uniqueness enforcement.

How can you accomplish both of those goals at once?

Significant Casing And Unique Constraints

One technique for email addresses and casing is to convert email addresses to lower case before they're stored. A downside of this technique is that it discards the original casing.

An alternative technique is to accept and store the input data in whatever casing the user has supplied at write time, but ignore the case information when the email address is presented at query time. How can this be accomplished?

The PostgreSQL module `citext`¹² provides a case insensitive character type. Using this character type as a column type allows you to achieve this write and read behavior.

To set this up, enable the `citext` extension in the Rideshare database. Enter this SQL statement into `psql`:

```
CREATE EXTENSION IF NOT EXISTS citext;
```

As you've done in earlier examples, enable the extension using a Migration. `citext` is not in Rideshare. Review extensions with `\dx`.

To test this out, create a new table in the Rideshare database but create it in a separate temp schema. Use the temp schema for tables that aren't part of the application schema. At the end you can drop the temp schema.

From `psql`, enter this statement to create Schema if it doesn't exist:

```
CREATE SCHEMA IF NOT EXISTS temp;
```

Generate a new migration and use the following example as the implementation. The Migration enables the `citext` extension and creates a customers table in the temp schema.

```
ruby/migration_add_customers_table.rb
class AddCustomersTable < ActiveRecord::Migration[7.0]
  def change
    enable_extension 'citext' unless extension_enabled?('citext')
    # Assumes a `temp` schema exists
    create_table 'temp.customers' do |t|
```

12. <https://www.postgresql.org/docs/current/citext.html>

```

    t.citext :email
  end
end
end

```

Or create the table with SQL directly by running `CREATE TABLE temp.customers (email citext);`.

Once you've added the `temp.customers` table, try inserting some data into it. Run the following Insert statement from `psql`.

```
INSERT INTO temp.customers (email) VALUES ('Customer1@example.com');
```

Now try querying for the row using a lower case “c” in the WHERE clause like `WHERE email = 'customer1@example.com'`.

```
SELECT * FROM temp.customers WHERE email = 'customer1@example.com';
```

You should have the row you inserted as a query result. Without `citext` there would have been no results.

For `citext` columns, Unique constraints enforce unique values regardless of their casing. Add a Unique constraint and try to insert a customer with an email address with the same text but different casing, and it will be prevented.

While `citext` is a great solution for retaining original casing and still enforcing Uniqueness, there may be other use cases where you wish to perform a normalization process on data prior to inserting it.

How can that be achieved?

Storing Transformations In Generated Columns

PostgreSQL 12 added a feature called *Generated Columns* and it's even supported by Ruby on Rails. The Rails support for Generated Columns¹³ is called virtual columns. These columns can be created using Active Record Migrations.

In this section you'll create a generated column to see how to use this feature.

Call your Generated Column `email_downcased`. If you've just added `temp.customers` in a Rails migration to Rideshare, roll it back by running `bin/rails db:rollback` in your terminal or drop the table with SQL.

Now you can replace the `create_table` portion from before adding the email column with the string type and also the `email_downcased` column.

13. https://guides.rubyonrails.org/active_record_postgresql.html#generated-columns

The `email_downcased` column will use the SQL `LOWER()` function to modify the input. The `create_table` snippet for the migration is shown below.

Generate a new Rails Migration and add the following implementation to the `change` method.

```
ruby/migration_snippet_virtual_column.rb
create_table 'temp.customers' do |t|
  t.string :email
  t.virtual :email_downcased,
    type: :string,
    as: 'LOWER(email)',
    stored: true
end
```

Run `bin/rails db:migrate` again to apply it. Launch `psql` and run `\d temp.customers` to view both columns.

The `email_downcased` will now have the following Default configuration:

```
GENERATED ALWAYS AS (LOWER(email::text)) STORED
```

The `GENERATED` keyword refers to the Generated Column. `ALWAYS AS ... STORED` shows that the transformed value is automatically stored that way. The `LOWER()` function is used as an expression to make the column data lowercase.

A benefit of the Generated Column approach is that it does not require enabling the `citext` extension.

On the other hand, it does require adding a another column to your schema. Consider whether you'd rather have an extension or the schema change.

In the next section you'll continue exploring consistency and correctness techniques you can add to your database.

PostgreSQL supports Enumerated Types that can be added directly to your database. If you're a seasoned Rails developer, you may be familiar with Enums at the application code layer but perhaps not in the database.

It's time to put those into action.

Constraining Values With Database Enums

Enumerated Types or *Enums* are a way to express a limited explicit list of possible values. These are the full comprehensive set of possible values that can exist. In a way, they are similar to Constraints in that they *constrain* the possible values.

When adding an Enumerated Type to your database it forms part of the database structure.

You may be familiar with Enums in Rails. In Active Record, an Array of Enum values can be used with an inclusion validation to ensure input data matches one of the types.

Although not an Enumerated Type specifically, the `VehicleStatus` class in the Rideshare codebase limits possible status options to Strings that are `DRAFT` or `PUBLISHED`.

Rideshare uses an Inclusion validation in the `Vehicle` model to constrain the possible values for status.

```
validates :status, inclusion: { in: VehicleStatus::VALID_STATUSES }
```

The status column also uses a NOT NULL constraint and a database column default of the String returned by `VehicleStatus::DRAFT`.

In the next section, you'll see how to use database enumerated types to constrain the input data.

The Rails Guides page on Active Record and PostgreSQL covers Enumerated Types.¹⁴

- Enums are a PostgreSQL type
- Additional enum values can be added either before or after existing values, but not in the middle
- Enum values cannot be dropped

The position of new values, and the inability to drop existing values, are significant trade-offs to be aware of.

If you require more flexibility than that, you may wish to leave the enumerated type in application code. If you do not imagine the values changing much, then putting them in the database provides stronger consistency guarantees at the cost of more rigidity.

First, the enum is added using the `create_enum` Ruby method. Second, the column is added and the type is set using the enum name. The default value and null handling are the same.

Copy and paste the following code into the `change` method in the Migration file you generated, and then run `bin/rails db:migrate` to apply it.

14. https://guides.rubyonrails.org/active_record_postgresql.html#enumerated-types

```
ruby/migration_snippet_vehicle_enum_usage.rb
```

```
create_enum :vehicle_status, [
  VehicleStatus::DRAFT,
  VehicleStatus::PUBLISHED
]

add_column :vehicles, :status, :enum,
  enum_type: :vehicle_status,
  default: VehicleStatus::DRAFT,
  null: false
```

After migrating, you've now created new database objects. These objects will be dumped into the db/structure.sql file. You'll see a CREATE TYPE statement that looks like this:

```
sql/create_type_as_enum.sql
```

```
CREATE TYPE vehicle_status AS ENUM (
  'draft',
  'published'
);
```

The pg_type system catalog may be queried to list Enums. Query it as follows, confirming the enum name and values match the Migration above. This query should be run from psql.

```
sql/view_pg_types.sql
```

```
SELECT
  n.nspname AS enum_schema,
  t.typname AS enum_name,
  e.enumlabel AS enum_value
FROM pg_type t
JOIN pg_enum e ON t.oid = e.enumtypid
JOIN pg_catalog.pg_namespace n ON n.oid = t.typnamespace;

-- enum_schema | enum_name | enum_value
-----+-----+-----
-- public      | vehicle_status | published
-- public      | vehicle_status | draft
```

At this point, there is a compatibility problem between the database enumerated type and the Active Record model code. Update the model code to include the enum information.

```
ruby/active_record_enum_usage.rb
```

```
enum status: {
  draft: "draft",
  published: "published"
}, _prefix: true
```

The `_prefix` option adds a `status_` prefix to the generated model methods. Try calling the method `status_draft?` on a `Vehicle Reservation` instance. You can instantiate model instances and call methods on them from `bin/rails` console.

To see all of the possible values for the enum, refer to the `db/structure.sql` file.

A benefit of using the enum as a column type is that it is now reusable for other tables. Another benefit of the enum is that it can be queried with only database access and without code access to see all possible values that a column could contain. This serves as a form of documentation for other users of your database that might not have code access.

In the article *PostgreSQL: ENUM is no Silver Bullet*,¹⁵ many examples of common enumerated types are listed along with their trade-offs.

The post *Enums vs Check Constraints in Postgres*,¹⁶ encourages skipping Enums entirely and using CHECK constraints with an explicit list of values to check against.

In this chapter you've seen how to use Check Constraints and Database Enums.

PostgreSQL has even more concepts for constraining values. In the next section, you'll look at how to use Domains which go beyond the limitations of Enums and Check Constraints.

Sharing Domains Between Tables

PostgreSQL provides another object type called Domains¹⁷ which have some similarities to both Enums and Constraints. In the post *Enforcing a Set of Values*,¹⁸ storing values as table rows, as an enum, or as a Domain are compared.

The Domain that's created is a Text type, and uses a CHECK constraint ensuring the value is one from an explicit list of values. The Domain name is then used as the column type on a table.

From `psql`, try creating a Domain by copying and pasting the following SQL statement for the `Rideshare` database.

```
sql/create_domain.sql
CREATE DOMAIN vehicle_statuses AS TEXT
```

15. <https://medium.com/swlh/postgresql-3-ways-to-replace-enum-305861e089bc>

16. <https://www.crunchydata.com/blog/enums-vs-check-constraints-in-postgres>

17. <https://www.postgresql.org/docs/current/sql-createdomain.html>

18. <https://justatheory.com/2010/01/enforce-set-of-postgres-values/>


```
CONSTRAINT valid_vehicle_statuses CHECK (
  VALUE IN ('draft', 'published')
);
```

After running the statement, from psql run \dD to “describe Domains”. You’ve now created a freestanding vehicle_statuses Domain that’s not attached to anything.

The Domain extends the Text type, and validates the Draft and Published statuses for a Vehicle.

The DOMAIN object is called vehicle_statuses and it uses a CHECK constraint to constrain the possible values.

To use the vehicle_statuses Domain for the status column, run the following statements. The first statement drops the existing status column so that it can be added back with the domain.

```
sql/vehicle_status_domain.sql
-- drop column so it can be added back
ALTER TABLE vehicles DROP COLUMN status;

-- add column with a Domain called "vehicle_statuses"
ALTER TABLE vehicles
ADD COLUMN status vehicle_statuses
NOT NULL
DEFAULT 'draft';
```

The vehicles.status column now uses a Domain called vehicle_statuses. The column does not allow NULL values and has a default value, and all of that configuration is defined in the database.

Row modifications must pass the CHECK constraint defined in the Domain. The main differences here compared with the Enum from earlier is that the NOT NULL constraint is packaged up along with the Domain. Enums do not have a way to attach additional constraint types to them in this way.

That means this Domain could be used for other tables that have a similar Draft and Published workflow and would benefit from similar Constraint checks. That’s pretty cool!

To reset the vehicles table back to the original configuration for Rideshare, run bin/rails db:reset.

You’ve now implemented a variety of mechanisms to help ensure that data is consistent, correct, and referentially intact. By putting these techniques into practice for your applications when they’re new, you can help avoid problems caused by bad data.

To scale this knowledge for your team, you may wish to implement some tooling in your project to help automate data-level checks. In the next section, you'll use an open source analysis tool that will help you achieve that.

Automating Consistency Checks In Development

To help quickly identify invalid or inconsistent data, you'll need the help of some tooling.

For Rideshare, use the `active_record_doctor`¹⁹ gem to analyze the database. If you're not familiar with adding Ruby gems to Rails projects, here's a quick refresher. Open the Gemfile and add the gem name to the development group. From your terminal, run `bundle install` to add it. This gem is already added to Rideshare and you can begin using it.

Active Record Doctor is a command line program. Run `bin/rake active_record_doctor` from the Rideshare directory in your Terminal.

One of the checks Active Record Doctor performs is to check that Primary Key and Foreign Key columns have matching types.

You may have a Foreign Key column with the integer type and a Primary Key using the bigint type. Change the Foreign Key column to use the bigint type so that they match.

Active Record Doctor also scans your Active Record source code, comparing the Validations with the database configuration and constraints. For example, when a uniqueness model validation exists, but no corresponding UNIQUE constraint exists, the tool suggests adding the UNIQUE constraint. The tool also suggests adding missing Foreign Key constraints.

Another gem to install and experiment with is called `database_consistency`.²⁰ Database Consistency makes suggestions by comparing Active Record models and database objects. This gem is also added to Rideshare.

For example, when a LIMIT is specified on a column in the database, but a similar length limit doesn't exist as a Active Record length validation, the tool suggests creating a length validation to match.

Using Active Record Doctor and Database Consistency as part of your Rails project will help improve consistency and correctness in your data and scale this knowledge to your development team.

19. https://github.com/gregnavis/active_record_doctor

20. https://github.com/djezzl/database_consistency

Now that you've built a solid foundation of consistent data, it's time to shift gears more towards the structure of your database, also often called the "schema".

Some types of changes to the schema or structure on a live and busy database can be potentially dangerous. Read on to find out how to detect those and how to make the changes safely.

Part III

Operate and Grow

Modifying Busy Databases Without Downtime

Over time your application codebase and database structure will evolve. Part of your development team's velocity hinges on being able to have a short time to delivery for both new code features and incremental database modifications. Rails developers use Active Record Migrations to evolve the database structure, constantly creating DDL changes as needed for features and as a change management mechanism to keep all databases in sync.

When your application is new and usage is low, modifying the database structure incurs almost no risk. Tables that are locked have few rows and lock times are short. There may not be much concurrent activity from customers or internal users. As an application grows in popularity and usage, data growth increases, query volume increases, and your team may begin to feel more pressure from database changes that become riskier and reduce the team development velocity.

One of the main challenges you'll face in evolving your database structure is performing operations that cause table locks, and how to do those safely and quickly.

How do you identify these riskier types of changes and how can you prevent them?

Structural changes from Active Record Migration methods that create `ALTER TABLE`¹ SQL statements can be riskier for tables with high row counts and high query volume. Most structural changes lock the table in Exclusive Access

1. <https://www.postgresql.org/docs/current/sql-altertable.html>

mode which means no other transactions can access the resource while the lock is in effect. How can you make any change to a table then for a busy database?


Don't worry. Most of the time the lock durations are short and clients can wait a bit if needed.

Some changes are less safe though, particularly for tables with very high row counts and queried heavily. In this chapter you'll learn to identify which changes are the most risky, and how to work around those risks using safer tactics.

Besides changes to the database structure, you may be conducting large scale *backfill* operations. Backfills populate one or more columns that didn't exist before and are otherwise null, to have useful values for the application.

Backfilling on tables with millions or billions of rows without stopping the server is another type of technical challenge that will be covered in this chapter.

Take a look at some of the terminology you'll be learning.

Busy Databases Terminology	
	• Strong Migrations — Library that adds more safety to Migrations
	• Multiversion Concurrency Control (MVCC) — PostgreSQL mechanism for managing row modifications and concurrent data access
	• ACID — Design properties of Atomicity, Consistency, Isolation, Durability
	• Isolation Levels — Configurable access level among transactions accessing the same data
	• Denormalization — Duplicating some column data
	• Backfilling — Populating a recently added empty column on existing table rows
	• Table Rewrites — Internal changes from some schema modifications that can cause a significant availability delay

Earlier you learned that some types of Migrations are riskier than others. How can you find the riskier ones?

Identifying Dangerous Migrations

As an experienced Rails developer and PostgreSQL user, you've likely had Migrations that didn't run correctly and possibly blocked your release process. Blocking releases slows down your development team velocity. Besides slowing the team down, Migrations could even cause application errors.

One solution would be to take your database down when you need to make schema changes. With a disconnected database there would be no concurrent modifications to worry about from application queries, so changes would be perfectly safe!

Unfortunately that strategy is impractical. Modern development teams don't take their databases offline to make changes. Modern teams are shipping code changes and evolving SQL database schemas every day, and fortunately PostgreSQL can keep pace with the needs of your team, allowing you to change the structure as often as needed.

How do you detect those scenarios?

When a developer creates an Active Record Migration for a DDL change, there is no built-in concept of safety or danger with Migrations themselves. All migrations are treated equally. This leaves the possibility open of deploying unsafe migrations that can harm team velocity and cause application errors.

Fortunately third party gems like Strong Migrations² fill this need. Strong Migrations has been added to Rideshare so you can experiment with it there. The main purpose of Strong Migrations is to identify potentially unsafe migrations during normal development, using well known patterns from DDL changes that can cause trouble. Strong Migrations even presents safer alternatives.

Read on to learn more.

Learning From Unsafe Migrations

To get some hands-on experience with how queries can get blocked, you'll create a long running modification and run a query during that time that gets blocked. You'll use the Rideshare database and the temp schema from earlier.

Launch psql from your terminal, connecting to the Rideshare database. To reset the temp schema, use the CASCADE option which drops any tables in that Schema. Create it again.

```
DROP SCHEMA IF EXISTS temp CASCADE;
```

2. https://github.com/ankane/strong_migrations

```
CREATE SCHEMA temp;
```

Within the recreated temp schema, create a slimmed down users table with an id and name column. This is a simplified form of a users table.

Next, populate the table with 10 million rows. Use CREATE TABLE AS to create the table and populate it in one statement.

Use GENERATE_SERIES() to get Integer values from 1 to 10 million for the id and to create a unique name.

Toggle timing to on with \timing to see how long the operation takes.

Run the following statements in psql. This will take a bit of time to populate.

```
sql/migration_dangerous_defaults_setup.sql
-- Enable timing
\timing

-- create users table (id, name)
-- populate it with 10,000,000 rows
CREATE TABLE temp.users AS
SELECT
    seq AS id,
    'Name-' || seq::TEXT AS name
FROM generate_series(1, 10000000) AS t(seq);
```

Verify the users table structure with \d temp.users and confirm it has 10 million rows.

Imagine you've made an application change and want to associate Users to Cities using a city_id column. You want all new and existing Users to have a City assigned by default.

In the application you might prompt users to change their default Cities, but you'd like to start with a default value.

To accomplish this, you'll add a new city_id column and give it a default value. This value will then be used for all new rows being inserted, but will also be applied retroactively to all 10 million existing rows.

For long time PostgreSQL users, adding a Default column value carries some baggage. Column defaults have been made mostly safe to perform on newer versions of PostgreSQL but there are still some gotchas.

Open up a second psql session connected to the Rideshare database. The example below uses "psql1" and "psql2" to refer to the first and second sessions.

In the first session add the default value and while that's being added, you'll run a query in the second session. Running a query simulates an application query running during a structural change.

You'll add the default value two times, a different way each time, demonstrating an important difference between them. The first one is relatively safe and the second one would be likely unsafe for a table with a lot of rows and in a high activity period.

The main difference is that in the first version, the column Default has a static value. This is very fast to add. Run the statement below which adds a static value of "1" to all rows.

```
sql/migration_safe_modification.sql
-- add column with a constant default value
-- This is safe to do and runs fairly quick
ALTER TABLE temp.users ADD COLUMN city_id INTEGER
    DEFAULT 1;
```

After adding the column, run `\d temp.users` and observe the `city_id` column and Default value.

Drop the column you just added to prepare for adding it a second time. You can run this from the "psql" session:

```
ALTER TABLE temp.users DROP COLUMN city_id;
```

Next is the unsafe version. In this version you'll run the ALTER TABLE statement, adding the column in the "psql" session again.

Run that now.

```
sql/migration_dangerous_modification.sql
-- !!! Dangerous Version !!!
-- Adds a "non-constant" or "volatile" DEFAULT
-- This takes a LOT longer
-- Table is locked in `ACCESS EXCLUSIVE` mode for duration
ALTER TABLE temp.users ADD COLUMN city_id INTEGER
    DEFAULT 1 + FLOOR(RANDOM() * 25);
```

While that's running, try selecting a single row from the same `temp.users` table in the "psql2" session. Run this SQL query:

```
SELECT * FROM temp.users LIMIT 1;
```

The DDL query in "psql1" might take ten seconds or more to run. The very simple query in "psql2", trying to select one row, will appear "hung" for the whole time until the modification in "psql1" completes, and then the query in "psql2" will finish instantly.

Now imagine this scenario in production for a table with hundreds of millions of rows, and thousands of queries per second trying to read from this table. This would be a disaster and cause huge amounts of errors, so it's something you'll want to avoid!

What steps can be put in place to help detect and avoid this type of scenario?

Learning To Use **CONCURRENTLY** By Default

PostgreSQL provides the **CONCURRENTLY** keyword as an option for a number of operations. Develop a mindset where **CONCURRENTLY** becomes a default option for you anytime you're working with a database connection to a Rails application that's running. Note that **CONCURRENTLY** is *NOT* added by default, and that there are scenarios where it's better to skip it. You'll learn some of the places where you can use it, and where it's safe to skip. It's up to you to put it into use as your default way of working if you aren't doing that already.

When **CONCURRENTLY** is used, statements generally take a **SHARE** lock type, meaning the operation has shared access to the resource. Without **CONCURRENTLY**, some operations will take an **EXCLUSIVE** lock on the table, meaning it's the only operation that can access the table for the length of time the lock is in effect.

Let's start with some of the places where you'll use it. Some of these operations you've not yet used here, but all examples will be covered.

- Adding an Index
- Dropping an Index
- Rebuilding an Index
- Adding a Partition
- Detaching a Partition

Once you've incorporated **CONCURRENTLY** on all your statements as your default way of working, you'll also want to know when it's not necessary and when not to use it.

Using **CONCURRENTLY** can make an operation like Creating an Index take twice as long. When adding an index to a table that's disconnected from any application or client queries, it's safe to create the Index without **CONCURRENTLY**, so that the creation is faster.

Adopting A Migration Safety Check Process

Each Active Record Migration file is a new source code file. At organizations that use Pull Requests or Merge Requests to introduce changes, a single change is prepared and a developer reviews it.

Experienced developers might identify dangerous database changes from their past experience, using their familiarity with table row counts and query pattern to notice changes that might cause long lock durations. Raising these concerns can help avoid errors.

On the other end of the spectrum, when Migrations are completely safe to perform at any scale, the authoring should release their change as quickly as possible.

While review feedback from team members is valuable, it requires team members to have skills to identify dangerous migrations and provide safer alternatives. Besides technical challenges, there are social challenges in providing this feedback in a graceful and timely fashion.

A good tool could help identify potentially unsafe Migrations just like a skilled developer and suggest safer alternatives. The tool could be used by any member on the team during their local development process, before publicizing their changes for review.

The tool could automatically block Migrations that are potentially unsafe but also provide an override mechanism.

Fortunately this tool exists and it's called Strong Migrations!

Exploring Strong Migrations Features

Strong Migrations strengthens the uptime and resiliency of Rails apps and engineering teams by adding safety checks for Migrations.

Most unsafe operations are detected and prevented by default. Besides the technical benefit, there is also an educational benefit for developers looking for feedback in a tight feedback loop. Any time Strong Migrations flags a Migration you've created, it's worth carefully reviewing the explanation and suggestions, and even reading more on the GitHub README.

Some of the operations that are detected are listed below.

1. When adding constraints (CHECK, FOREIGN KEY, and NOT NULL) to an existing column, Strong Migrations checks whether they're validated immediately or not

2. Strong Migrations checks for table modifications *and* data backfills that would extend the duration of an exclusive lock
3. When removing or renaming a column or table, Strong Migrations checks whether this will create a problem with the Schema Cache
4. Strong Migrations prevents changing the type of a column to an incompatible type.
5. When Adding or Dropping an Index without the CONCURRENTLY keyword, Strong Migrations prompts the developer to use CONCURRENTLY.

How does it work?

Strong Migrations hooks into the Rails Migration process. Once it's added to a Rails project, every time a developer runs `bin/rails db:migrate` it checks any pending modifications for safety.

A Ruby Exception is raised when a potentially dangerous migration is detected which prevents the developer from applying it.

This forces the developer to either abandon it, rewrite it, or explicitly opt out of the safety check. To opt out, the developer can wrap their changes in a `safety_assured` Ruby block. Review the `db/migrate` directory of Rideshare to see some places with opt out examples.

Most of the time, Strong Migrations helpfully catches unsafe migrations.

Because this whole cycle may be conducted before ever submitting a change for review, it creates an opportunity for developers to learn privately about this information. This tight feedback loop can contribute to scaling knowledge on the team around safe and unsafe database changes.

Next you'll look at a specific example from Rideshare.

Trips can have a rating of integer values between 1 and 5. As an Active Record Validation, Rideshare uses a "numericality" validator for the Trip model that makes sure the value is between 1 and 5. The rating value can also be nil since trips aren't rated until after the Driver completes the trip.

Review the rating field validation in `app/models/trip.rb`.

The rating validation is useful but it's implemented on the client side Rails application.

The rating validation can also be implemented in PostgreSQL using a CHECK constraint.³ The CHECK constraint used here has already been added to

3. <https://www.postgresql.org/docs/current/ddl-constraints.html#DDL-CONSTRAINTS-CHECK-CONSTRAINTS>

Rideshare. Review the following Migration file that added the constraint to the database.

```
ruby/migration_trip_rating_check_constraint.rb
```

```
class AddTripRatingCheckConstraint < ActiveRecord::Migration[7.0]
  def change
    add_check_constraint :users,
      "rating >= 1 AND rating <= 5",
      name: "rating_check"
  end
end
```

The Migration uses `add_check_constraint` (added in Rails 7) for the trips table and rating column. The constraint boolean logic is very similar to what's in the Active Record Ruby code although it's written in SQL.

Besides boolean logic, the database column itself acts as a form of validation since it only allows compatible values for the type. For example, a floating point value cannot be stored in the rating column with an integer type.

Strong Migrations flags adding a CHECK constraint and validating it in one operation.

As a safer variation, it suggests adding the constraint without validating it. You'll look at how to do that next.

Review these Rideshare migrations:

```
sh/migration_check_constraint_dangerous.rb
```

```
# Example message from Strong Migrations:
#
# === Dangerous operation detected #strong_migrations ===
#
# Adding a check constraint key blocks
# reads and writes while every row is checked.
# Instead, add the check constraint without
# validating existing rows, then validate them
# in a separate migration.
#
# Add a migration with "validate: false"
class AddTripRatingCheckConstraint < ActiveRecord::Migration[7.0]
  def change
    add_check_constraint :trips,
      "rating >= 1 AND rating <= 5",
      name: "rating_check",
      validate: false
  end
end

# Second migration, validates the constraint
class ValidateAddTripRatingCheckConstraint < ActiveRecord::Migration[7.0]
```

```
def change
  validate_check_constraint :trips, name: "rating_check"
end
```

In the first Migration, you'll see that Strong Migrations prints that it detected a potentially dangerous migration.

Remember that constraints like CHECK may validate all rows or only new rows. Adding a CHECK constraint to a table requires an Exclusive Access lock while being added. Normally this is quick, but when all rows need to be validated, this extends the length of the lock. As you saw earlier, with a lock in place even for ten seconds many queries in your application might be blocked, causing application errors.

Instead, Strong Migrations suggests using "validate: false" in your Migration, which means the constraint is added without validating all rows; only new rows will be validated when they are created. This means the exclusive lock is very brief causing little or no interruption.

But what about the existing rows that aren't being modified, that existed before the constraint was added?

Those rows have not yet been validated for compatibility with the CHECK constraint condition. This is not a good final state to be in, because you're not able to validate all table data.

To do this you'll need to validate the constraint. This is accomplished with a second Migration that calls the `validate_check_constraint` method.

Prior to running this, in psql run `\dt trips` and confirms the constraint is visible but has the keywords NOT VALID:

```
Check constraints:
"rating_check" CHECK (rating >= 1 AND rating <= 5) NOT VALID
```

With the second migration, you'll validate the constraint. After running it you should no longer see the NOT VALID keywords.

With a validated constraint, you can be confident that all values for the rating match the CHECK constraint definition.

Locking, Blocking, and Concurrency Refresher

As described earlier, one of the operational challenges with PostgreSQL is when tables are locked for modifications and queries get blocked.

This section will go into a bit more depth on that.

All SQL statements, including SELECT statements (DQL), DML, and DDL, run inside a transaction.

Transactions are normally automatic and implicit, although as the developer you may use explicit transaction blocks with the keywords BEGIN and then COMMIT or ROLLBACK. You may also lock resources explicitly.

PostgreSQL reports which lock types conflict with each other in the *Conflicting Lock Modes* table on the *Explicit Locking*⁴ documentation page. Familiarize yourself with different DDL and DML statement types and which locks they take.

If you're a Rails developer who has written Migrations, you'll first need to translate the Active Record migration methods like `add_column` into the SQL statements like ALTER TABLE that they generate. Once you have a SQL statement, you can see what the needs are from the resource being changed, like a table or row, and whether it needs access exclusively for itself, or shared access.

With this level of understanding, you can create less disruptive modifications and SQL statements, and less lock contention from the code and SQL you write. The benefit of reducing lock contention code is that you might achieve fewer errors and higher concurrency.

If you do have lock contention and lock issues, how can you gain more insights into what's happening?

As you saw earlier, PostgreSQL system catalogs do internal bookkeeping and thus provide visibility into what's happening including for lock activity. You can query the `pg_locks` catalog views. Review the *Lock Monitoring*⁵ PostgreSQL wiki for more information.

To get some hands on experience with lock contention, one way is to simulate it using your local database. You'll do that next!

In this exercise you'll create two transactions that contend with one another.

Follow the instructions in the SQL comments (they start with two dashes) below, opening two `psql` sessions connected to the Rideshare database.

```
sql/locks_and_blocks.sql
```

```
-- Start "psql1" and "psql2" in two Terminal windows
-- psql --dbname rideshare_development
-- "psql1"
BEGIN; -- start a transaction
```

4. <https://www.postgresql.org/docs/current/explicit-locking.html>

5. https://wiki.postgresql.org/wiki/Lock_Monitoring

```

-- create explicit EXCLUSIVE LOCK of trips table
LOCK trips IN ACCESS EXCLUSIVE MODE;

-- "psql2"
-- Inspect pg_locks view
-- Find the "AccessExclusiveLock" for "trips"
SELECT
    mode,
    pg_class.relname,
    locktype,
    relation
FROM pg_locks
JOIN pg_class
ON pg_locks.relation = pg_class.oid
AND pg_locks.mode = 'AccessExclusiveLock';

-- -[ RECORD 1 ]-----
-- mode      | AccessExclusiveLock
-- relname    | trips
-- locktype   | relation
-- relation   | 461492

-- "psql1", inspect the current lock_timeout;
-- A lock_timeout of 0 means it is disabled, so
-- a statement waiting on a lock will run forever
SHOW lock_timeout;

-- "psql2"
-- Create the DDL statement from earlier.
-- DDL statement acquires an exclusive lock
-- But it will get blocked!
ALTER TABLE trips ADD COLUMN city_id INTEGER DEFAULT 1;

-- Since this hanging go back to "psql1"

-- Query pg_stat_activity from "psql1"
-- Look for waiting transactions
-- Confirm the ALTER TABLE is listed and blocked
SELECT
    wait_event_type,
    wait_event, query
FROM pg_stat_activity
WHERE wait_event = 'relation'
AND query LIKE '%ALTER TABLE%';

-- -[ RECORD 1 ]---+-----
-- wait_event_type | Lock
-- wait_event      | relation
-- query           | ALTER TABLE trips ADD COLUMN city_id INTEGER DEFAULT 1;

-- NOTE: UNBLOCK
-- To unblock, COMMIT or ROLLBACK the original transaction from "psql1"

-- "psql1"

```


ROLLBACK;

The ALTER TABLE statement gets blocked trying to modify the trips table because the trip table is locked with exclusive access within a transaction.

Use the condition LIKE '%ALTER TABLE%' and wait_event = 'relation' to identify the blocking query.

The purpose of this exercise was mainly to simulate how a query or DDL modification can be blocked. How might you protect against that?

Preventing Excessive Queueing With A Lock Timeout

One protection you can add to your database is to set a lock_timeout. The purpose of this timeout is to set an upper bound on the amount of time a transaction can wait to acquire a lock. Once the timeout is exceeded, PostgreSQL will cancel the query.

This is a little different than a statement_timeout, but they both share the cancellation behavior. PostgreSQL tracks the duration of these queries and cancels them if they exceed the value you set.

The lock_timeout⁶ parameter can be set database wide, within a session, or for a single SQL statement.

In the example from the last section, the lock_timeout was not set.

Try running the experiment again but this time you'll set a lock_timeout. You'll need two psql sessions, "psql1" and "psql2", again each connected to the Rideshare database.

Use the statements below. The main difference compared with the earlier example is that you're running the SET LOCAL lock_timeout = '5s'; statement which sets a lock_timeout for that transaction of 5 seconds.

```
sql/set_lock_timeout.sql
-- create psql1
-- start a transaction
BEGIN;
LOCK trips IN ACCESS EXCLUSIVE MODE;

-- create psql2
-- set a transaction level lock_timeout
BEGIN;
SET LOCAL lock_timeout = '5s';

-- Run the modification
```

6. https://postgresqlco.nf/doc/en/param/lock_timeout/

```
-- It should hang since the table is locked for exclusive access
-- But it should get canceled after 5s
ALTER TABLE trips ADD COLUMN city_id INTEGER;

-- In psql2 notice the statement is canceled
-- ERROR: canceling statement due to lock timeout

-- run rollback in both psql1 and psql2
ROLLBACK;
```

You’ve now seen how to set a max allowable time for lock acquisition. What other kinds of safeguards can be added?

Exploring Lock Type Queues

Statements requesting access for a lock type are put into a queue where they wait in order.

If transaction B is waiting to acquire the lock type that transaction A has exclusive access to, B must wait for A.

As described in *PostgreSQL and the Lock Queue*,⁷ the pids for the backend processes are logged including the statement holding a lock type and the statements waiting to acquire it.

Some helpful lock parameters are to set `log_lock_waits` to on and setting `deadlock_timeout` to get more visibility into locks or blocked queries. When these timeouts cause cancellations they’ll be logged to the `postgresql.log`. See the post “Cost of `log_lock_waits` and `deadlock_timeout`.”⁸

Take a look at the following example. This example shows the pid holding a lock and the pids waiting for the lock type.

```
DETAIL: Processes holding the lock: 29386. Wait queue: 31025, 30551, ...
```

The pids 31025 and 30551 are blocked waiting on pid 29386. Once the query associated to the backend pid 29386 completes, the pids that are waiting will be processed in the order of how they appear.

Besides using the `pg_stat_activity` system catalog you learned about earlier to view lock related queries, you may want to use a log analyzer that analyzes your `postgresql.log` file.

The log analyzer program `pgBadger`⁹ organizes lock related query information it parses from your log file. The lock information is put into categories like

7. <https://joinhandshake.com/blog/our-team/postgresql-and-lock-queue/>

8. <https://dba.stackexchange.com/a/249904>

9. <https://github.com/darold/pgbadger>

“Most frequent waiting queries” and “Queries that waited the most” that can be helpful for investigations.

Setting Statement Timeout

In this section you’ll follow a similar pattern to how you set a `lock_timeout` but focus on the `statement_timeout`.

Note that queries getting canceled for taking too long cause an application error and a bad experience, especially when they’re related to a user-facing Rails application query compared with a non-user facing background job query. In both cases the associated queries need to be fixed so that they don’t exceed the timeout. The solutions will depend highly on the circumstances.

Setting too low of a value for `statement_timeout` cancels queries unnecessarily. Not leaving it set at all allows queries to run forever and consume resources that are finite, such as a database connection. A happy middle ground is to use the `statement_timeout` and select a value that balances these two extremes.

In a Ruby on Rails web application, user-facing queries should run in less than 250ms for ecommerce or consumer web sites and applications. In B2B Enterprise SaaS applications, there may be less appetite for query optimization, or tolerance for slower queries. However, fast queries are a good goal regardless of the type of application, because in the long run fast queries make it easier for the same server to achieve higher concurrency, and for your team to avoid costs related to scaling up.

If your application does not have a statement timeout set at all, try starting with a very high one like 10 minutes or 5 minutes. As you fix queries and some time passes without observing cancellations from timeouts, lower the timeout to 5 minutes or 1 minute. You may want to allocate some database connections within your limited set for especially slow background queries. You may also want to allow a higher timeout value for background job queries that aren’t user-facing.

Lock Timeout and Statement Timeout are two options worth setting.

In the next section you’ll consider how Active Record caches your schema.

Avoiding Schema Cache Errors

When Active Record starts up, for each model that's backed by a database table, the table fields are scanned and cached as the *Schema Cache*¹⁰ for as long as the application is running.

Dropping a database column without considering Active Record and active instances of the application means that a schema change may cause an application error where there is code that tries to write or read from a column that no longer exists.

This isn't required for every change; refer to the documentation for specific examples. This section is intended to familiarize you with the Schema Cache concept and how it works so that you can determine whether this risk exists for your change. This type of error would be more common at a higher scale as well, where there may be multiple running copies of the application.

To drop a column safely, first work in Active Record. Use the Active Record `ignored_columns`¹¹ mechanism to add the field you're planning to drop first. Add the field there, commit and release your change, making sure the application restarts.

A field added to `ignored_columns` can safely be dropped from the database without concern that it's still in the Schema Cache.

Backfilling Large Tables Without Downtime

Over time your application structure will evolve. You may introduce a new column that will be NULL by default, and then populated with a value for the purposes of the application.

Populating values like this is usually called *backfilling*. As your table row counts grow large, backfills take longer and can become more complicated and risky from an operational perspective.

When possible, backfills could be made optional if the application can fall back to a default value or some other value when the column is NULL. When that's not possible, you'll need to backfill.

While you may conduct backfills from a Ruby application using a Rake task, in this section you'll conduct backfills using SQL, connected to your database using `psql`.

10. <https://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/SchemaCache.html>

11. <https://www.bigbinary.com/blog/rails-5-adds-active-record-ignored-columns>

Backfills should not harm client application queries that are running at the same time. To accomplish that, your backfill operations need to run efficiently and consume few resources. Ideally your server instance is over-provisioned as well, meaning it has a lot of spare compute and disk IO capacity.

From a timing perspective, running backfill queries during a low activity period is a good idea. From an operations and team coordination perspective, capture the SQL being run in a SQL file or script, share it on your team, and conduct the work in a pre-production location first to validate the approach.

An overview of some backfilling concepts is presented below.

- Consider using a *Double Writing* or “Dual Writes” strategy
- Consider a purpose-built intermediate table to assist in the backfill
- Use batched operations to limit resource consumption
- Add a throttle mechanism to slow down row copying if needed. This allows time for Index Maintenance and Replication.
- For a purpose-built intermediate table, create specialized indexes based on how the table is queried. Make the table Unlogged and disable Auto-vacuum

In the next sections you’ll explore some of these techniques in greater detail.

Backfilling And Double Writing

Modern engineering organizations are able to perform data migrations online in most cases, to avoid taking down the database server which halts business operations.

This does make the process more challenging and risky compared with a backfill process where the database is fully stopped. This means that online migrations will typically require careful planning and execution.

After introducing a new database column, you may want to plan out multiple phases where first the write operations are set up, and then the reads.

This technique can be called *Double Writing*. Double writing can help as a transitional technique to minimize the time needed in a cutover where reads are switched from the original location to a target location, by ensuring the data population is conducted for both locations at the same time.

Separating Reads and Writes for Backfills

Backfills are straightforward when they can SELECT from one table and UPDATE another table.

When the table being queried has poor query performance, it will be inefficient to regularly query it. Determine if you can modify the table with special-purpose indexes to speed up the query you'll need for the backfill.

On the destination side, slow updates can cause excessive lock durations and block other readers. In an upcoming chapter you'll see how to use EXPLAIN to analyze both the read and write.

If the queries to read and write aren't fast enough, you may want a special-built table to assist in the backfill process. Read on for more information.

Specialized Tables For Backfills

When optimizing the read side and write side queries is not enough, you might want to create a specialized table to assist in the backfill process.

The data in this table will be temporary in nature, and will duplicate the source data. A benefit of creating a special-purpose table with duplicate data is that there will be no other connections or contention for these table rows. This separate table can also be used to track the backfill state.

A downside of duplicating data is that more space is consumed.

With the intermediate table scenario, rows selected from the source table are written into a new intermediate table instead of into the target or destination table.

Since this table is designed specifically for the backfill, you'll add indexes to it or remove items from it to help speed the process up. This solution adds more effort earlier but is designed to speed up the total time of the operation.

For the specialized table, now you can start selecting what's needed and not needed. Start by reducing the columns from the source table to the bare minimum, and create the table without any indexes or constraints.

To make writes to the table even faster and avoid having this table replicated, set the table to "Unlogged".

To do that, from psql run the SET UNLOGGED statement for the table if it's already been created. An unlogged table does not have write protection from the Write-Ahead Log (WAL). If PostgreSQL restarts unexpectedly, you may lose data for this table. Since this table only has duplicated data though, having it unlogged is a reasonable trade-off since it could be re-created if necessary.

Once the data is populated, add indexes that are as narrow and as special-purpose as possible, closely matching the query you're using.

Since there is no concurrent activity, skip the `CONCURRENTLY` keyword when building indexes. Autovacuum can also be disabled for this table. If the table was called `trip_requests_intermediate` then you'd run this SQL statement to disable Autovacuum:

```
sql/alter_table_disable_auto_vacuum.sql
```

```
ALTER TABLE trip_requests_intermediate
SET (autovacuum_enabled = false);
```

Next you'll put intermediate table techniques into practice with Rideshare.

Practicing Backfilling Techniques

Let's practice some of the backfill techniques to develop your skills in this area.

Using the example where you added a `city_id` column to the `users` table, imagine that you were creating a purpose-built intermediate table that would act as the data source to select from.

This table could have some of these characteristics:

- A minimal set of columns
- Purpose made index for a single query
- No constraints
- Unlogged
- Autovacuum disabled

Run these statements in `psql`:

```
CREATE SCHEMA IF NOT EXISTS temp;

CREATE UNLOGGED TABLE temp.users_intermediate (
  user_id bigint, city_id integer
);

ALTER TABLE temp.users_intermediate
SET (autovacuum_enabled = false);
```

Generate some rows:

```
INSERT INTO temp.users_intermediate (user_id, city_id)
SELECT GENERATE_SERIES(1, 10, 1), GENERATE_SERIES(1, 10, 1);
```

For your backfill, you'll `UPDATE` the `users.city_id` column, using matching rows from the `temp.users` table. If the `users.city_id` column does not exist, you can add it by running this statement in `psql`:

```
ALTER TABLE users ADD COLUMN IF NOT EXISTS city_id INTEGER;
```

Imagine that `temp.users_intermediate` had millions of rows. You want to select rows as fast as possible.

Add a Multicolumn index covering both columns:

```
CREATE INDEX temp_users_user_id_city_id
ON temp.users_intermediate (user_id, city_id);
```

Since you have few rows in the table, PostgreSQL will choose a Sequential Scan. You want to confirm the index is posted, so disable Sequential Scans for your session by running `SET enable_seqscan = OFF;`.

From `psql`, run the following query and add `EXPLAIN` in front of it. Confirm the index is named and an Index Scan is used.

```
UPDATE users
SET
    city_id = temp.users_intermediate.city_id
FROM temp.users_intermediate
WHERE users.id = temp.users_intermediate.user_id
AND users.id > 0
AND users.id < 10000;
```

For your batch size, use a range that is sized appropriately for your database server, and iterate through the `users` table using the primary key `id` column. There may be deleted rows in the range, but you can plan on there being up to 10,000 rows updated at once for this batch size.

Use the PL/pgSQL Procedure that you've already learned earlier to turn this single statement into a loop.

Besides batched updates, consider adding pauses within your updates to add time for index writes and replication. Monitor server resource consumption while your statements are running.

Once the destination side table is updated, you can safely remove the `temp.users` table.

Wrapping Up

In this chapter you considered how to safely make database structural changes on large and busy databases. You saw lock types and queues, and learned how queries can become blocked. You simulated lock behavior and saw how to set a lock timeout and statement timeout to add more protections to your database

You saw backfills and learned techniques to make them run faster, so that they can run online while other application queries are running at the same time.

You worked a lot in PostgreSQL. In the next chapter you'll work more with Active Record and learn to optimize your application queries.

Optimizing Active Record

Now the focus switches away from PostgreSQL and SQL and to the Active Record query layer in your Rails application.

You'll see a variety of ways to improve the performance of your application by eliminating unnecessary queries, using caches, and using advanced query designs.

Active Record is the *Domain Specific Language* (DSL) that Rails developers use to create SQL queries from Ruby code. Queries are generated and fragments of queries as Ruby code can be composed together.

If you've written SQL before and are newer to Rails, you may wonder whether writing your queries as Active Record is worth it. You may prefer to write SQL directly. What are some of the reasons to prefer Active Record?

Preferring Active Record Over SQL

Active Record has always been the sole built-in ORM framework for Ruby on Rails. Ruby on Rails is one of the most popular open source Ruby projects, meaning many developers learned Ruby primarily in working with the Ruby on Rails framework.

As a full-stack framework, Ruby on Rails has always been opinionated or “prescriptive”, including all of the major layers needed to build web applications, especially an ORM that connects Ruby language objects and instances to database tables and rows.

Rails advocates for *Convention Over Configuration* as a design philosophy. Active Record is the conventional way to describe your application data model and do write queries. Whether a developer is familiar with SQL or not, by embracing the conventional way to write queries, a team may have an easier time growing their contributor base and maintaining their application.

You'll see how to write basic and advanced Active Record queries, as well as write SQL within Active Record. Active Record doesn't limit you to writing SQL queries solely in Ruby, so you've got options.

In Active Record, fragments of SQL queries like filter conditions in a WHERE clause are called *Named Scopes*. The Named Scopes pattern provides a conventional technique to identify most of the “filtering code” by reading through application models. Named Scopes can be shared among models as well and there can be default scopes that get applied to all queries for a particular model.

One example where common filtering occurs and reuse helps is with “soft deletes.” In a conventional soft deletes design, users delete records which are then hidden from the application but not physically removed as rows. Instead they have a `deleted_at` timestamp column that's set. This pattern is implemented in a common location as Ruby code leveraging the Active Record Named Scopes and Default Scope concepts.

Active Record allows a developer to start with a model class and gradually add methods to it. For this chainable design, query generation is deferred until the end, which is called lazy evaluation.

While this design is a good developer experience, it comes with a trade-off: developers can easily write inefficient queries, sometimes unknowingly.

Before you can fix inefficient queries, you'll need to see where they are coming from in the application.

Query Logs To Connect SQL to App Code

Although you may analyze slow queries in an Application Performance Monitoring (APM) tool, when analyzing SQL queries in the `postgresql.log` you can trace them back to the Active Record that generated them.

In this section, you'll learn how to link SQL queries to application code. Prior to Rails version 7, third party gems fulfilled this purpose, but eventually support was added directly to Rails for this very useful functionality.

Prior to Rails version 7, the `Marginalia`¹ gem added comments to SQL queries that showed where in the application the query was generated. The `Marginalia` comment lists the application name, controller name, and controller action name. An example comment looks like this:

```
/*application:Rideshare,controller:trips,action:index*/
```

1. <https://github.com/basecamp/marginalia>

This comment shows that the query was generated from the Trips Controller Index Action in the Rideshare application. The Marginalia documentation advises against adding unique value components when using Prepared Statements because a unique value will prevent re-use of them. You'll learn about Prepared Statements soon.

Since knowing where the query was generated is extremely useful, this capability was added directly to Rails 7 and is called "Query Log Tags."

Rideshare enables Query Log Tags by setting `config.active_record.query_log_tags_enabled` to `true` in `config/application.rb`. Note that Prepared Statements is incompatible with Query Log Tags (See the Warning in the `config.active_record.query_log_tags_enabled` section²).

For this section, Prepared Statements were disabled in the application by putting `prepared_statements = false` in `config/database.yml`.

With Prepared Statements disabled and Query Log Tags enabled, Rideshare will show similar comments to the Marginalia gem in the logged SQL queries. You'll even notice these in the logged SQL in the server log or console log output. For example, notice `/*application:Rideshare*/` comment and run the example below from `bin/rails console`.

```
irb(main):020:0> Trip.first
Trip Load (0.5ms) /*application:Rideshare*/ SELECT "trips".*
FROM "trips" ORDER BY "trips"."id" ASC LIMIT $1 [["LIMIT", 1]]
=>
```

Now you've got a new query observability tool that can help you as you work on improving your application queries.

What are some of the common query problems that can happen?

Common Active Record Problems

The following terminology guide can be used to help learn terms or refresh your knowledge on terms used in this chapter.

Active Record Optimization Terminology



- Normalization — Removing duplication from fields
- Lazy Loading — Loading data "late" and on-demand
- Eager Loading — Loading data "early" or upfront

2. <https://edgeguides.rubyonrails.org/configuring.html#config-active-record-query-log-tags-enabled>

Active Record Optimization Terminology

- **Strict Loading** — Preventing Lazy Loading, requiring Eager Loading
 - **Counter Cache** — Storing a count in a column and incrementing it at insert time
-

While in-depth coverage of data normalization is beyond the scope of this chapter, the basics will be covered here including how they were put into practice for Rideshare.

Good data normalization suggests that your design should eliminate duplication of attributes between tables when possible. This means that each database table has a unique set of fields. When data needs to be linked together, Foreign Key columns refer to Primary Key columns and this relationship is both described and enforced using constraints.

Data relationships might take the form of *one to one*, *one to many*, or *many to many*.

Most Rails applications have many data models and those models are related to each other. Active Record code loads data from many tables, which means rows are joined together in various ways. This is where one of the pitfalls can come in, when rows from multiple tables are not joined together efficiently.

In the earlier section you learned about lazy evaluation of Active Record code, which causes Lazy Loading of data. Lazy Loading means that the data is not loaded upfront. Rails has a different term for loading data upfront: *Eager Loading*.

What's the relevance of Eager Loading? The Eager Loading technique is commonly the solution to one of the most common and well known types of inefficient queries, the *N plus one (N+1)* query pattern. As a refresher on the N+1 pattern, it involves queries for at least two tables, where an outer query runs once, and an inner query runs N times.

Let's break it down further. Querying the right side or inner side repeatedly is a problem because usually it's from Lazy Loading data that could otherwise be Eagerly Loaded.

Let's look at an example from Rideshare. Open `bin/rails` console from your Terminal, connecting to the Rideshare database.

Make sure SQL queries are logged to your console and run this code:

```

ruby/n_plus_one_example.rb
# Show SQL if not already enabled in bin/rails console
ActiveRecord::Base.logger = Logger.new(STDOUT)

Vehicle.all.each do |vehicle|
  vehicle.vehicle_reservations.count
end; nil

```

The count of vehicle_reservations for a Vehicle is queried for every Vehicle.

The SQL queries will look like these:

```

ruby/n_plus_one_queries.rb
Vehicle Load (2.1ms) \
  SELECT "vehicles".* FROM "vehicles" /*application:Rideshare*/
SELECT COUNT(*) ... WHERE "vehicle_reservations"."vehicle_id" = 1
SELECT COUNT(*) ... WHERE "vehicle_reservations"."vehicle_id" = 2
SELECT COUNT(*) ... WHERE "vehicle_reservations"."vehicle_id" = 3
SELECT COUNT(*) ... WHERE "vehicle_reservations"."vehicle_id" = 4

```

N+1 queries can happen in any Active Record whether it's in the model, views, or serializers. You'll need to discover N+1 instances by monitoring your Rails log or test log, or a team member may spot them during code review.

Manual checks are useful, but automated tools exist that can help you identify N+1 queries as well. The Ruby gem *prosopite*³ is an N+1 detection gem and it's been added to Rideshare. *Prosopite* depends on *pg_query*⁴ for PostgreSQL which was also added.

Let's see if *Prosopite* picks up this N+1. Run the code again in Rideshare in bin/rails console inside of a *Prosopite* block. Set *Prosopite*.rails_logger = true.

```

ruby/n_plus_one_detection.rb
Prosopite.rails_logger = true

Prosopite.scan do
  Vehicle.all.each do |vehicle|
    vehicle.vehicle_reservations.count
  end
end

```

Running this code shows “N+1 queries detected” in the Rails Console. *Prosopite* can even be configured to detect N+1s in tests and fail the tests.

Now that you can detect N+1 queries manually and automatically, how do you fix them?

3. <https://github.com/charkost/prosopite>

4. https://github.com/pganalyze/pg_query

Use Eager Loading To Reduce Queries

To fix N+1 queries, use Eager Loading and eagerly load associated data. You'll need to test to make sure you get the same query results. Usually one query is more efficient even if it takes a little longer, but you'll need to test a bit to confirm.

Active Record Associations (See: *Types of Associations*⁵) like `has_one`, `has_many`, or `belongs_to` to describe different types of relationships and generate different SQL queries to join rows of data from tables together.

Models can be configured for one of six supported types. When you see repeated queries for the same table there's a good chance all of the data can be queried for that table in one query.

Preload data using the `.preload()`⁶ method to combine multiple queries into a single statement.

In bin/rails console, run the same code again but add `.preload()`.

```
Vehicle.preload(:vehicle_reservations).all.each do |vehicle|
  vehicle.vehicle_reservations.size
end; nil
```

This code avoids the N+1 query pattern but it also adds one more trick. `.count()` was replaced with `.size()` which means that when all of the Vehicle Reservations were loaded at once for all of the Vehicles, the size of the loaded collection was used instead of issuing a SQL COUNT() query to count how many there were.⁷

Preloading isn't the only method of Eager Loading data. Read on to learn more.

Eager Loading With .includes()

Besides `.preload()`, `includes()` is a second technique to Eager Load data and performs an equivalent SQL query.

Try running the following query with `.includes()` and compare the SQL queries.

```
Vehicle.includes(:vehicle_reservations).each do |vehicle|
  vehicle.vehicle_reservations.size
end; nil
```

5. https://guides.rubyonrails.org/association_basics.html#the-types-of-associations

6. <https://api.rubyonrails.org/classes/ActiveRecord/QueryMethods.html#method-i-preload>

7. <https://www.speedshop.co/2019/01/10/three-activerecord-mistakes.html>

When using `.preload()` the WHERE clause conditions for the second table being joined cannot be modified. The resulting SQL query uses an IN clause for the top level query with a list of primary keys from the second table.

If `.includes()` generates an equivalent SQL query then what value does it add?

With `.includes()`, Active Record can pick between two strategies. `.includes()` will work equivalently to `.preload()` or it will generate a LEFT OUTER JOIN to join the tables together. How does it choose which one to use?

Active Record chooses a LEFT OUTER JOIN when there are conditions on the associated table. What do conditions on the associated table look like?

Modify the example from earlier that fetches Vehicle Reservations, but now fetch the active and not-canceled reservations.

Run the following code in bin/rails console and take a look at the generated SQL query.

```
ruby/n_plus_one_detection.rb
Prosopite.rails_logger = true

Prosopite.scan do
  Vehicle.all.each do |vehicle|
    vehicle.vehicle_reservations.count
  end
end
```

Notice that it's used a LEFT OUTER JOIN in order to accommodate the query conditions for the vehicle reservations.

About about traditional inner joins?

Active Record supports the `.joins()` method and when a traditional inner join is possible, this will likely be the most efficient type of join.

You've now seen a couple of ways to avoid N+1 queries by using Eager Loading.

What if you could configure your application to make it impossible to write N+1 queries altogether?

Prefer Strict Loading Over Lazy Loading

Active Record introduced a new feature called *Strict Loading* in Rails 6.1⁸ that can be used to prevent lazy loading.

Strict Loading can be specified on a per query basis, or globally for an application.

8. <https://github.com/rails/rails/pull/37400>

To better understand how Strict Loading works, take a look at an example using the Rideshare application, continuing to work with the Vehicles and Vehicle Reservations.

Run the following code in bin/rails console.

```
ruby/vehicle_reservations_strict_loading.rb
vehicles = Vehicle.strict_loading.all

vehicles.each do |vehicle|
  vehicle.vehicle_reservations.first.starts_at
end

# `Vehicle` is marked for strict_loading. The VehicleReservation association
# named `:vehicle_reservations` cannot be lazily loaded.
# (ActiveRecord::StrictLoadingViolationError)
```

This time when you run the code you'll get this error:

Vehicle is marked for strict_loading. The VehicleReservation association named :vehicle_reservations cannot be lazily loaded. (ActiveRecord::StrictLoadingViolationError)

When Strict Loading is enabled, Eager Loading must be performed in order to run this query. Try modifying the query above using the Eager Loading techniques you learned earlier to avoid the Strict Loading exception being raised.

You've now worked through one way of removing the possibility of N+1 queries from your Rails application. What other kinds of query-level optimizations can you make?

Optimizing Individual Active Record Queries

Database connections are a finite resource, and all queries require a database connection to run. Reducing unnecessary queries avoids excessive use of limited database connections which can help the scalability of your database server.

What kinds of changes can you make on an individual query basis?

Limiting the fields needed can help speed up queries. One way to do that in Active Record is to use `.select()`, which does not generate a SQL query and uses loaded data. (See: *Rails Pluck vs Select and Map/Collect*⁹)

Another method is `.pluck()`, which generates SQL that specifies limited fields.

9. <https://www.rubyinrails.com/2014/06/05/rails-pluck-vs-select-map-collect/>

Try running this code from bin/rails console:

```
Vehicle.includes(:vehicle_reservations).
  pluck('vehicles.name',
        'vehicle_reservations.starts_at',
        'vehicle_reservations.canceled')
```

.pluck() is even smart enough with invalid SQL, which requests fields that don't exist on a table, to figure out which tables to get the fields from.

The above query code can be rewritten as below and produce an equivalent result. This may be confusing to maintainers though, and a more the more explicit version above that selects the fields from the appropriate tables is clearer.

```
Vehicle.includes(:vehicle_reservations).
  pluck(:name, :starts_at, :canceled)
```

A major benefit of limiting the fields at query time is unlocking the possibility of Index Scans, and even Index Only Scans, to make retrieval very fast. You'll learn about those soon.

In the next section, you'll see how to save a round trip query by asking for fields to be returned before inserting data.

Save A SELECT With A RETURNING

PostgreSQL has a capability called the RETURNING keyword. It can be used to ask for fields to return before inserting data.

For fields that are populated by the database like a primary key field populated from a Sequence, you might write code to insert data and then query for the id value. Those second queries needed to look up the database supplied value can be replaced using the RETURNING keyword.

Try using the temp.users table you set up earlier. The table does not currently have a sequence associated with the id column so you'll set that up by running these statements from psql:

```
CREATE SEQUENCE temp_users_id_seq INCREMENT 1 START 1;
ALTER TABLE temp.users ALTER COLUMN id
SET DEFAULT nextval('temp_users_id_seq');
```

The first statement adds a sequence and the second statement assigns it to the id column. Now id will be populated by the sequence when you insert rows.

Try inserting a row, and add the RETURNING keyword asking for the id column.

```
INSERT into temp.users (name) VALUES ('bob') RETURNING id, name;
```

You should see the id values in the response without needing to query for them.

Fortunately Active Record supports RETURNING and you can experiment with this using bin/rails console. Insert a single Driver using `.insert_all()` and ask for the id column value:

```
ruby/insert_all_with_returning.rb
Driver.insert_all([
  {
    first_name: 'Andrew',
    last_name: 'Atkinson',
    email: "andrew.atkinson@example.com",
    password_digest: SecureRandom.hex
  }],
  returning: [:id]
)

_.first["id"]
```

You've now used RETURNING to eliminate a query.

How else might you speed up queries?

Bounding Query Results Using LIMIT

Another common way to resolve a poorly performing query is to add boundary conditions. Bounds can be added by filtering to fewer rows or by adding a LIMIT to the amount of rows that are fetched at one time using the Active Record `.limit()` method.

Most queries should be bounded when they are first written, but for small row counts unbounded queries will typically perform OK, and then slow down or not run at a certain row count size. The tipping point will depend on the queries and the server resources.

Another way to add boundaries is to use the Active Record batch operations like `.find_each()`, `.find_in_batches()`, and `.in_batches()` (See: *Rails — `find_each` v.s `find_in_batches` v.s `in_batches`*¹⁰).

These methods are typically used to reduce the amount of memory consumed at once, since each result row turns into an Active Record instance. However these methods also add limits to queries that can help them run reliably even as row counts increase.

10. <https://medium.com/lynns-dev-blog/rails-find-each-v-s-find-in-batches-v-s-in-batches-d5ca9bfe37d>

You've now seen some basics for individual query optimization. As your queries become more complicated, you want to reach for more advanced techniques.

Advanced Query Support In Active Record

*Subquery Expressions*¹¹ or *Subqueries* are what PostgreSQL calls multiple SQL queries combined into one statement. The general pattern is that an outer query uses a second inner query as a way to filter rows down. Filtering is done using a WHERE clause or IN clause. Subqueries are common in SQL and recently support was added to Active Record.

You'll adapt an example from the PostgreSQL Tutorial on Subqueries¹² to Rideshare creating an analytical type of query to show what's possible in a single statement. You'll first write the query in SQL and then convert it to Active Record. The query has these goals:

- Find the average number of Trips for all Drivers
- Find Drivers with more Trips than average
- Order above average Drivers by Trip count in descending order

To find the average in SQL, use the AVG() aggregate function. If you'd been working in bin/rails console, go back to psql for these since you're running SQL queries.

```
sql/select_avg_trips_count.sql
SELECT AVG(trips_count) FROM users
WHERE type = 'Driver';
```

Use the average Trip Count as *input* to a WHERE clause outer query, by making the Trip Count query an inner query or subquery. Modify the query below to find drivers above and below the average.

```
sql/trips_count_subquery.sql
SELECT
  id, trips_count
FROM users
WHERE type = 'Driver'
AND trips_count > (
  -- the earlier AVG(trips) query
  SELECT AVG(trips_count) FROM users
  WHERE type = 'Driver'
)
ORDER BY trips_count
DESC LIMIT 5;
```

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

12. <https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-subquery/>

You now have a list of your above average drivers by trip count, in descending order based on the trips they've provided.

How can you adapt this Subquery to Active Record?

Taking inspiration from *Advanced Active Record: Using Subqueries in Rails*¹³ on the PgAnalyze blog, take the techniques and write or review the Active Record code below. Run this code in bin/rails console and confirm that the generated SQL is the same.

```
ruby/active_record_subquery.rb
Driver.where('trips_count > (:avg)', avg: Driver.select('avg(trips_count)').
  order(trips_count: :desc).
  limit(5))
```

In this code, :avg is a placeholder and the value is supplied in a Hash key and value style. Instead of SQL query result rows, you're now working with Active Record objects. Pretty cool.

Subquery Expressions in SQL are very useful but they can become difficult to read. Another advanced query design technique called Common Table Expressions (CTEs) helps make multi-query statements better organized and easier to read and maintain.

Using Common Table Expressions (CTE)

Common Table Expressions abbreviated as a CTE,¹⁴ are a technique in PostgreSQL to combine two or more queries into a single statement with named parts.

CTEs are also called WITH queries because they use the WITH keyword. They'll be referred to as CTEs in this chapter.

CTEs combine queries that are independent but can be used to work together to pull data from multiple places. CTEs can be used recursively by combining a non-recursive portion and a recursive portion and then using a UNION to combine them. Recursive CTEs are beyond the scope of this chapter, but the PostgreSQL Tutorial has a nice example showing how to find direct reports for a manager using the technique.¹⁵

In this section, you'll create a CTE to answer questions about Rideshare Drivers. You'll look at new Drivers on the platform. As you did in the last

13. <https://pganalyze.com/blog/active-record-subqueries-rails>

14. <https://www.postgresql.org/docs/current/queries-with.html>

15. <https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-recursive-query/>

section, you'll write the CTE first with SQL and then look at how to translate a CTE into Active Record.

First, write a SQL query without a CTE that finds Drivers created in the last 30 days. From psql connected to the Rideshare database, run this SQL query:

```
sql/trips_new_drivers.sql
-- Drivers created in the last 30 days
SELECT *
FROM users
WHERE created_at >= (NOW() - INTERVAL '30 days');
```

Imagine a Rideshare feature that shows ratings for new Drivers to driver operations managers. To fetch the data, use the rating field on the trips table.

Your CTE will have named query parts so that the query stays organized and maintainable. To populate a lot of rated trips, run `bin/rails data_generators:trips_and_requests` from your terminal. Once you've got a lot of rated trips available, enter this SQL statement into psql:

```
sql/trips_top_drivers_cte.sql
WITH new_drivers AS (
  SELECT *
  FROM users
  WHERE type = 'Driver'
  AND created_at >= (NOW() - INTERVAL '30 days')
),
rated_trips AS (
  SELECT id, driver_id
  FROM trips
  WHERE rating IS NOT NULL
)
-- Display Driver names and
-- the Average rating for their trips
SELECT
  trips.driver_id,
  CONCAT(users.first_name, ' ', users.last_name) AS driver_name,
  ROUND(AVG(trips.rating), 2) AS avg_rating
FROM trips
JOIN users ON trips.driver_id = users.id
JOIN new_drivers ON trips.driver_id = new_drivers.id
JOIN rated_trips ON trips.id = rated_trips.id
WHERE users.type = 'Driver'
GROUP BY 1, 2
ORDER BY 3 DESC
LIMIT 10;
```

Let's break down what's happening in the CTE above. The CTE has named parts `new_drivers` and `rated_trips` which are each independent queries being brought together to combine Driver data and Trips data. With the data sources

available, the Drivers are then presented with their names and ratings, in descending order showing the top rated drivers first.

Support for PostgreSQL CTEs has been added to recent versions of Active Record in Ruby on Rails. Rails 7.1 adds a `.with()` method (See: *Rails 7.1 Construct Common Table Expressions*¹⁶) to generate a CTE statement. Prior to Rails 7.1 creating a CTE could be done using the `activerecord-cte`¹⁷ gem.

To get some hands on experience with that, you'll adapt a simpler CTE to Active Record. From your terminal, run `bin/rails console` and use this Active Record code:

```
ruby/active_record_cte_with.rb
# use the "." endless range
Trip.with(recently_rated:
  Trip.where.not(rating: nil).
  where(created_at: 30.days.ago..)
).
from("recently_rated").
count
```

Notice how this example combines multiple Active Record style queries into one. The `.with()` method is used as an outer query portion. The name `recently_rated` is given as a name to an inner query portion. Review the generated SQL. This query only counts rows, but consider using the techniques to create the earlier Driver SQL CTE as Active Record.

You've now seen how to perform Subqueries and CTEs in both SQL and Active Record.

As queries become more complex, it can be nice to encapsulate the details of complex queries and provide a simpler interface. Fortunately such a capability exists in PostgreSQL and it's called Database Views. Read on to see how those work and how to manage them with Ruby on Rails.

Introducing Database Views for Rideshare

In this section you'll work with *Database Views* which are a built-in object type for PostgreSQL.

Briefly, database views are a way to name and encapsulate a query and store that as an object within PostgreSQL.

16. <https://blog.kiprosh.com/rails-7-1-construct-cte-using-with-query-method>

17. <https://github.com/vlado/activerecord-cte>

Database Views are not currently supported natively by Active Record. However, in this section you'll create views to use with Rideshare. Since views have a definition that you may want to evolve over time, you'll manage the lifecycle of views using Rideshare. PostgreSQL does not offer a built-in way to manage versioned views over their lifecycle, so using the client application to do this provides a solution.

For Rideshare you'll use a third party Ruby gem that extends Active Record, adding support for Database Views.

Some of the new terminology in this section is listed below.

Administration Terminology



- Database View — Encapsulates a SQL query
 - Materialized View — A View with pre-calculated results to query
-

A database view encapsulates (Wikipedia: *Encapsulation*¹⁸) a SQL query and gives it a name. Imagine that you frequently queried New Drivers in Rideshare and wanted to have a shared definition of what a new driver is as a query that could then be re-used for other purposes. You might create a view called “new_drivers” with a query like this one:

```
CREATE VIEW new_drivers AS
  SELECT * FROM users
  WHERE type = 'Driver'
  AND created_at >= (NOW() - INTERVAL '30 days');
```

With that view created in PostgreSQL, you're now able to write a very simple query, `SELECT * FROM new_drivers;`

Unless views are Materialized, they execute live queries. What does Materialization mean? A Materialized view is a type of View that means the query can be run in advance and then queries to the view – like the one above that queries the pre-calculated results instead of running a live query.

Since the query results are pre-calculated with materialization, querying a materialized view is very fast. The trade-off is that the query results become stale. Fortunately Materialized Views can be refreshed `CONCURRENTLY`. Let's see how to create both types of views and keep materialized views refreshed.

Database views have a definition and you may want to evolve the definition over time. While there's no built-in solution to PostgreSQL or Ruby on Rails

18. [https://en.wikipedia.org/wiki/Encapsulation_\(computer_programming\)](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming))

for this, you'll use a third party gem to create and manage versioned views. Read on to learn more.

Creating the Search Result Model With Scenic

The third party Ruby gem scenic¹⁹ can be used to create and manage versioned views.

This gem works in the same way as Active Record where the developer writes Ruby code like `create_table` which then produces SQL DDL like `CREATE TABLE`.

The Scenic gem includes a generator that generates two files. The first file is a versioned plain text file with a `.sql` extension where you'll place the SQL definition for your view. The second file is a Migration that adds the view to the database.

For Rideshare, Scenic was used to create a database view to help when searching for Drivers. Open `psql` and run the `\dv` meta command to list the existing views.

Rideshare has a `search_results` database view that's connected to a `SearchResult` Ruby object in the codebase:

```
ruby/search_result.rb
class SearchResult < ApplicationRecord
  belongs_to :searchable, polymorphic: true

  # this isn't strictly necessary, but it will prevent
  # rails from calling save, which would fail anyway.
  def readonly?
    true
  end
end
```

The `SearchResult` class acts as an interface to the database view. To try it out, load `bin/rails console` and run `SearchResult.first`. This will execute the view with a live query and map results into Ruby object instances. The SQL query will be logged in the `bin/rails console`.

The `SearchResult` model interaction for querying it works the same way as other Active Record classes that are backed by database tables and not views.

Using scenic, the definition of the view is versioned and can be changed over time.

19. <https://github.com/scenic-views/scenic>

Views can also be materialized and refreshed concurrently. Read on to learn more.

Improving Performance With Materialized Views

Views do not have a performance advantage when they're not materialized. They do help encapsulate complex SQL queries and can be used to create Active Record object instances backed by a view instead of a database table using Scenic. These benefits are great, but there's also another very big benefit from a performance perspective you'll work with next.

Materialized Views have great performance advantages. When application code can use Materialized Views, very high performance query code can be written because the views have pre-computed query results. Query result execution can be done concurrently as well, which you'll see how to do. This means clients can keep querying the pre-computed results while the view is recalculated in the background and replaced.

To improve performance even further, as is the case with database tables, Materialized views can have Indexes added to them. By writing queries that use Indexes on your Materialized Views you can build very high performance query code.

In the last section you worked with the `SearchResult` model and `search_results` view but this view was not materialized.

In this section you'll work with a new model called `FastSearchResult`.

This model was generated using the `--materialized` flag provided by the `scenic` gem that creates a Materialized View to back the model. The view was generated by running this command:

```
bin/rails g scenic:model fast_search_results --materialized
```

Both views have been added to the `db/structure.sql`. Search in the file for `CREATE MATERIALIZED VIEW` to view the definition. The `db/structure.sql` file shows only the definition of the view and not the precomputed query results.

Viewing the definitions of Materialized Views from `psql` is a bit different. Check the `pg_matviews`²⁰ System Catalog to inspect the definition:

```
SELECT * FROM pg_matviews;
```

The field `ispopulated (bool)` shows whether the view has been populated, and `hasindexes` shows whether any indexes have been added.

20. <https://www.postgresql.org/docs/current/view-pg-matviews.html>

To get the performance advantages you'll want to regularly refresh your materialized views, which can be done both from the application code or with SQL.

The `FastSearchResult` model has a `refresh` method. From `bin/rails` console, refresh the view like this:

```
FastSearchResult.refresh
(20.7ms) REFRESH MATERIALIZED VIEW "fast_search_results";
=> #<PG::Result:0x0000000115bc40d8 status=PGRES_COMMAND_OK \
    ntuples=0 nfields=0 cmd_tuples=0>
```

Compare the response times between `SearchResult.first` and `FastSearchResult.first`. The materialized view will return much faster!

Rideshare keeps the materialized view results refreshed using a cron job. Ruby on Rails does not offer native cron job scheduling support so the third party gem `whenever`²¹ is used to generate and manage jobs.

Review `config/schedule.rb`. This code uses the Rails runner²² command to invoke the same code you called earlier from `bin/rails` console.

```
every 15.minutes do
  runner "FastSearchResult.refresh"
end
```

Materialized Views can be refreshed concurrently as well, although they do require a Unique index to be added. Rideshare added a Unique index to `fast_search_results` to enable concurrent refreshes.

To refresh concurrently from `psql`, run this statement:

```
REFRESH MATERIALIZED VIEW CONCURRENTLY "fast_search_results";
```

You've now seen how to leverage database views with high performance materialized views, that can be indexed and refreshed concurrently.

Besides these techniques, Ruby on Rails adds caching layers for database queries that help improve your application performance. Let's see what types of caches are available and how to leverage them.

Reducing Queries With Active Record Caches

Active Record has a variety of caches that run to improve the performance of your Rails application code.

21. <https://github.com/javan/whenever>

22. https://guides.rubyonrails.org/command_line.html#bin-rails-runner

The first one is the *Query Cache*²³ which is a cache of the SQL statements. Query Cache entries only live for the duration of a Rails controller action.

Loading a Rider's first Trip Request at the beginning of an action with `TripRequest.where(rider_id: 1).first` will place the query and result into the Query Cache. The second time the same query is run inside the same controller action, the query will match against the cached version and the result will be returned from cache instead issuing a second database query to PostgreSQL.

In this way, Active Record helps you avoid unnecessary queries with the Query Cache. To see this in action, take note of CACHE entries in your Rails server log and execution times printed as (0.0ms). The CACHE statements in the Rails log can also be disabled (See: *Disable Rails CACHE Logging*²⁴).

A trade-off with the Query Cache is that more memory is used hold the query and the result. Memory is used to instantiate Active Record objects and their relationships populated from the Query Cache (See: *Cached queries guidelines*²⁵).

While being a useful feature, an over reliance on the Query Cache usage from excessive memory consumption could be an opportunity to improve your database queries making them faster and lighter.

In the next section you'll see PostgreSQL Prepared Statements²⁶ and how to work with them in Active Record.

Prepared Statements With Active Record

Prepared Statements²⁷ are a type of object in PostgreSQL that take a specific query and separate the parameters that are bound to it.

Prepared Statements give a name to the query without parameters. Prepared statements can be viewed by querying the system view `pg_prepared_statements`. Try running `SELECT * FROM pg_prepared_statements;` from `psql` to see what's stored in there.

Prepared Statements are enabled by default and store up to 1000 statements per connection with Active Record.

23. https://guides.rubyonrails.org/caching_with_rails.html#sql-caching

24. <https://heliom.ca/blog/posts/disable-rails-cache-logging>

25. https://docs.gitlab.com/ee/development/cached_queries.html

26. <https://www.postgresql.org/docs/current/sql-prepare.html>

27. <https://guides.rubyonrails.org/configuring.html>

To use Prepared Statements, do not set `query_log_tags_enabled`.

From `bin/rails` console try running some Active Record queries.

```
User.first
```

Explore the saved prepared statements for your connections by running this code:

```
ActiveRecord::Base.connection.execute(
  'SELECT * FROM pg_prepared_statements'
).values
```

Prepared Statements have some incompatibilities so you may need to disable them. Besides being incompatible at publication time with Query Log Tags, they also don't work with some types of connection pooling and PgBouncer, which you'll work with in an upcoming chapter.

Ruby on Rails keeps innovating with prepared statements though. Prior to Rails 7, `SELECT*` was cached when fields weren't requested. In Rails 7 a setting `enumerate_columns_in_select_statements` was added²⁸ that enumerates the list of columns for `SELECT` statements, helping increase re-use of the prepared statement cache.

You've now seen how to use the Query Cache and Prepared Statements. Let's continue the cache theme, and look at Counter Caches.

Eliminating Slow Count Queries With Counter Caches

Another type of Active Record cache is aimed at improving slow `COUNT()` queries by using a pre-computed result instead.

This cache is called the *Counter Cache*. Counter Caches in Active Record keep a running tally of counts and store the tally in an integer column on a table.

In Rideshare, an example `COUNT()` query might be the number of trips a Driver has performed. This query could be done in high volume if many Riders are trying to access the trips count. Take a look at the following Active Record code to get the count of trips for a Driver.

```
Driver.first.trips.count
```

This code can be improved with a counter cache. First, this code is using `.count()` which always issues a SQL `COUNT()` query. Instead of `.count()` prefer the `.size()` method because it counts objects in memory when they're already loaded instead of issuing a SQL `COUNT()` query.

28. <https://www.bigbinary.com/blog/rails-7-adds-setting-for-enumerating-columns-in-select-statements>

Besides counting objects in memory, `.size()` also uses a Counter Cache column when it's available which avoids issuing a `SQL COUNT()` query.

Counter Cache columns include various trade-offs. They require a new column be added and they also add a small bit of latency at write time. Counter caches do open up the possibility of inconsistent counts. A high volume of `UPDATE` statements to the same counter cache column can also cause database bloat which you'll learn more about later.

Despite these downsides, counter caches are a great performance tactic in many situations because they provide nearly instant counts without `SQL COUNT()` queries. Read more in *Caching counters with ActiveRecord's counter caches*.²⁹

The following Migration for Rideshare added a `trips_count` column to the `users` table to act as a counter cache.

```
sh/add_counter_cache.sh
# create the migration
bin/rails g migration AddTripsCountToUsers trips_count:integer

# run the migration
bin/rails db:migrate
```

The `counter_cache: true` option was added to the `:driver` association in the `Trip` model:

```
class Trip < ApplicationRecord
  belongs_to :driver, counter_cache: true
end
```

When adding a new counter cache to a production system, the counter cache will not be populated for existing rows. You'll need to populate the cached value before you can use it or it will return 0.

To explore how the Counter Cache works for a driver, run the following code from `bin/rails` console.

```
ruby/reset_counters.rb
driver = Trip.first.driver
driver.trips.count
Driver.reset_counters(driver.id, :trips)
driver.trips.size
```

This code gets a `Driver`, and queries for the `Trips` count. Notice it makes a `SQL` query. Next it resets the `Trips Count Counter Cache` for the `Driver`. By

29. <https://blog.appsignal.com/2018/06/19/activerecords-counter-cache.html>

then replacing `.count()` with `.size()` and calling `trips.size` repeatedly, notice how no SQL queries are performed to fetch the value. `driver.trips.size` is equivalent to reading the field `drivers.trips_count` directly, which was populated earlier.

You’ve now seen a variety of ways to use Caches to improve your query performance.

Another type of poor performance can be conducting calculations in Ruby code. PostgreSQL has a rich variety of calculation methods available and may be a better place to conduct this work. These methods are available to use with Active Record.

Performing Aggregations In the Database

One opportunity to improve Active Record performance is to perform calculations in PostgreSQL instead of in Ruby and Active Record code, by using built-in SQL aggregate functions.

Database aggregate operations will generally provide better calculations faster due to fewer round trips between the application and the database.

PostgreSQL aggregate functions and Active Record equivalents are displayed below.

	Active Record	SQL
Count	<code>.count()</code>	<code>COUNT()</code>
Average	<code>.average()</code>	<code>AVG()</code>
Minimum	<code>.minimum()</code>	<code>MIN()</code>
Maximum	<code>.maximum()</code>	<code>MAX()</code>
Sum	<code>.sum()</code>	<code>SUM()</code>

Table 1—Aggregations Comparison

Equivalent Aggregations in Active Record and SQL

Active Record support is provided in the `ActiveRecord::Calculations`³⁰ module. For PostgreSQL, refer to documentation on *Aggregate Functions*.³¹

PostgreSQL has more Aggregate Functions beyond those supported by Active Record. PostgreSQL even allows users to make their own aggregate functions called *User-Defined Aggregates*.³²

30. <https://api.rubyonrails.org/classes/ActiveRecord/Calculations.html>

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

32. <https://www.postgresql.org/docs/current/xaggr.html>

You've seen how to improve your Active Record query code in a variety of ways. Now you'll explore how to directly write SQL within Active Record.

Reduced Object Allocations With SQL In Active Record

While writing Active Record query code typically involves writing Ruby, Active Record also supports SQL. This is a hybrid strategy and is a useful technique where memory or speed is very important.

Active Record has a number of methods that support direct SQL. The first one you'll look at is `.find_by_sql()`. With this method, instead of Ruby Active Record code, you may use SQL as text as an argument.

The results are mapped to instances of the calling class and stored in a Ruby Array. Normally Active Record returns results in a `ActiveRecord::Relation` which enables lazy evaluation and chaining. One trade-off with `.find_by_sql()` is that you won't get lazy evaluation or be able to chain more methods on.

Try running this code from bin/rails console:

```
TripRequest.find_by_sql("SELECT * FROM trips LIMIT 1")
```

Another way to run SQL in Active Record is with the `.execute()` method. With this method the results are mapped to “primitive types” in Ruby like the String or Integer type.

Experiment with this method by running this code from bin/rails console:

```
ActiveRecord::Base.connection.execute("SELECT * FROM trips LIMIT 1")
```

The results are placed into a `PG::Result` class instance. This is a bit lower level than Active Record. Take note of fields like `ntuples=1` and `nfields=7` which indicate one row (tuple) was returned with 7 fields of data.

When primitive types are acceptable or even desired for the response, `.select_all()` and `.select_one()` can be good options.

`.select_one()` works like `.select_all()` but returns only the first item.

With primitive types you won't incur the performance penalty of instantiating Active Record objects because that process does not happen. You'll need to determine whether the results from these methods are compatible with where you wish to use the result data.

To better understand the memory savings opportunity, use Ruby benchmarking techniques as demonstrated in *Optimizing Active Record queries*³³ to

33. <https://dev.to/kattyacuevas/optimizing-active-record-queries-4i84>

compare different ways of fetching and using row data with Active Record. The information below is a performance sample collected using Benchmark.memory, comparing Active Record and SQL equivalents instantiating Trip objects.

Comparison:

```
Using select_one:      4660 allocated
Normal Trip.first:    1988419 allocated - 426.70x more
```

Active Record allocates 426 times more objects compared with the SQL approach. The SQL approaches do not create Trip model instances so the latency involved with that could be avoided.

Wrapping Up

You now have several ways to optimize your Active Record code, starting from improved observability connecting your SQL queries to the Active Record query code that generated it.

You removed unnecessary queries from Active Record helping, conserve limited database resources.

You learned how to identify the N+1 query pattern manually, automatically, and how to fix it using Eager Loading or avoid it using Strict Loading.

You created Subqueries, Common Table Expression (CTE) queries, and Database views. You learned how Views can be Materialized to provide performance advantages. You saw how the lifecycle of versioned database views can be managed over time.

You learned about caches provided by Rails, including the Query Cache and Counter Cache.

With Active Record covered thoroughly, it's time to shift your focus away from the Rails application that generates SQL queries, and into query level observability once those queries have arrived to PostgreSQL.

In the next chapter you'll explore SQL query level performance observability and improvements. Read on to get started.

Improving Query Performance

In this chapter you'll analyze SQL queries running on PostgreSQL, developing a richer understanding of their characteristics as they relate to performance.

To do that you'll learn how to generate and read query execution plans, and use statistics that the database continually collects. Once you understand which portions of a query are most costly, you can begin to use tactics that reduce the costly portions which can improve the overall scalability of your server instance.

Query optimization is a complex subject, with entire books dedicated to covering it. In this chapter you'll get equipped with the basics, learning where to find information and how to interpret it.

Review the following terminology to learn about it for the first time or get refreshed on it. You'll be working with terms like *selectivity*, *cardinality*, a variety of Index types, filtering, sorting, and more in this chapter.

Query Performance Terminology



- Selectivity — How selective a query is
- Cardinality — How many unique values there are
- Sequential Scan — Reading all rows in table
- Index Scan — Fetching values from an Index on a table

To get insights from your query execution, you'll also work with your logs. Before going to PostgreSQL logs, what kind of query information can you get from Active Record logs?

Logging Slow Queries With Active Support Notifications

With Ruby on Rails and without any extra gems, plugins, or database extensions, a wealth of query information can be added to the Rails application log. *Active Support Notifications*¹ can be used to monitor slow queries. (See: *Track slow DB queries without additional gems*²)

To see this in action, read and understand the *Slow Query Subscriber* class added to Rideshare that uses Active Support.

The *Subscriber* listens for events formatted as `sql.active_record` that have query information. The duration of the query can be calculated using the start and finish times associated with the event. When the duration exceeds 1 second, the full SQL statement can be logged using the Rails logger.

Review the class in Rideshare or copy this:

```
ruby/slow_query_subscriber.rb
# Inspiration: https://twitter.com/kukicola/status/1578842934849724416
class SlowQuerySubscriber < ActiveSupport::Subscriber
  SECONDS_THRESHOLD = 1.0

  ActiveSupport::Notifications.subscribe('sql.active_record')
  do |name, start, finish, _, data|
    duration = finish - start

    if duration > SECONDS_THRESHOLD
      sql = data[:sql]
      Rails.logger.debug "[#{name}] #{duration} #{sql}"
    end
  end
end
```

Open `bin/rails` console to test this out. By running the query `SELECT pg_sleep(1);` from Active Record as a query that simulates taking 1 second or more, you'll see the instrumentation get triggered and the following log output.

```
irb(main):004:0> ActiveRecord::Base.connection.execute("SELECT pg_sleep(1)")
[sql.active_record] 1.005035 SELECT pg_sleep(1)
(1005.2ms) SELECT pg_sleep(1)
```

In this technique you've logged a slow query from the client application side. What about when you want to view slow queries from the server side?

1. https://guides.rubyonrails.org/active_support_instrumentation.html
2. <https://twitter.com/kukicola/status/1578842934849724416/photo/1>

Capture Query Statistics In Your Database

The queries in your database consume resources and it's helpful to focus your optimization efforts on the most costly queries. To do that you'll want to get a global view of all queries to see which ones are most costly. You'll use the `pg_stat_statements`³ module which you configured in an earlier chapter, and will be referred to as PGSS.

PGSS takes each query and removes the parameters from it, creating a sort of query group or pattern. Each query then can be put into this group or pattern, and statistics can be collected at the group or pattern level. PGSS then makes the statistics available with a system catalog view.

These groups or patterns are called the *normalized* form of the query. Each normalized query gets a Query Identifier (queryid). The queryid uniquely identifies the original query text before the normalization process. The parameters are removed in the normalization process and replaced with placeholder characters.

Part of the purpose of the normalization where parameters are removed, is so that queries with the same structure can be grouped together. Statistics can then be computed across queries that are similar, sometimes referred to as a "group".

Query statistics are at the group level including the duration and the number of calls, across all instances of a query within the group that may have very different sets of specific parameters.

Enabling the PGSS module requires a database restart. PGSS is generally recommended to add for all databases. Although it adds a minimal amount of latency to collect statistics, the information it provides is worth it. PGSS is supported by cloud providers of PostgreSQL. If you don't already have it enabled in your production database, plan a time to make this parameter change and restart your database during a low activity period.

To configure the module, add `pg_stat_statements` to `shared_preload_libraries` in `postgresql.conf`. Once that's done and PostgreSQL has restarted, you'll enable the extension.

To enable the extension, run the following statement from `psql`. This only needs to be done once per database.

3. <https://www.postgresql.org/docs/current/pgstatstatements.html>

```
sql/create_extension.sql
```

```
CREATE EXTENSION pg_stat_statements;
```

You may also use `ALTER SYSTEM` to modify `shared_preload_libraries` from a `psql` prompt but this isn't recommended. This method generates a value in `postgresql.auto.conf` that overrides the value in `postgresql.conf` (Thanks to Lukas Fittl for this tip!⁴).

This can lead to confusion about which value is active, so consider making all changes only in your PostgreSQL config file and keeping a version controller backup of your single config file.

Rideshare enabled PGSS by enabling the extension via a Rails Migration.

Now that you've enabled PGSS, read on to find out how to populate and review query statistics for Rideshare.

Rideshare Query Statistics

Since Rideshare isn't a running system, you'll need to simulate application activity by populating some query information.

To do that, start up a Rideshare server by running `bin/rails server` from your Terminal.

In another terminal window, run `bin/rails simulate:app_activity` to simulate activity. You'll should see queries being logged to your server. Since PGSS was enabled earlier, when those queries arrived in PostgreSQL, the denormalization and statistics collection process was happening in the background.

PGSS tracks 5000 normalized queries by default. To track more than that, raise the `pg_stat_statements.max` value. The least-executed queries are discarded when the max is reached.

From `psql`, query `SELECT * FROM pg_stat_statements;` to view everything PGSS has collected. A more useful query, though, might be to look at the top few slowest queries.

Once you've done that, PGSS should have captured some statistics.

From your terminal, run `psql --dbname rideshare_development` again and then use the following query to look at some of the queries and fields of data from PGSS. You may want to run `SELECT pg_stat_statements_reset();` to reset the statistics and run the Rideshare simulation again to isolate the results to being only from the simulation.

4. <https://www.postgresql.org/docs/current/config-setting.html>

```
sql/ten_worst_queries.sql
```

```
SELECT
  total_exec_time,
  mean_exec_time,
  calls,
  query
FROM pg_stat_statements
ORDER BY mean_exec_time DESC
LIMIT 10;
```

You should now see some Rideshare queries in the PGSS results. A result row will contain `total_exec_time`, `avg_ms`, `calls`, and the normalized query like in this example:

```
sql/pg_stat_statements_result.sql
```

```
-- -[ RECORD 1 ]-----
-- total_exec_time | 2903.9372489999996
-- avg_ms          | 580.7874497999999
-- calls           | 5
-- query           | SELECT "users".* FROM "users" WHERE "users"."type" = $1
```

This is great. You can now review statistics collected database-wide for Rideshare queries. You're accessing these results from `psql`.

When you work on a team of application developers who might not want to use `psql` to view this information, how can you more easily expose it to them?

Introducing PgHero As a Performance Dashboard

The PgHero tool allows you to review query performance information with your team. PgHero is a PostgreSQL Performance Dashboard. PgHero is an open source Rails Engine that ships as a Ruby gem available on GitHub.

When PGSS is enabled and configured, PgHero captures this information and stores it in tables that are presented in a web application view. This can make the information much more accessible to team members that aren't viewing raw PGSS information from `psql`.

PgHero presents the following data for each query:

- Total number of calls
- Average duration
- The percentage the query represents among the total number of query executions

The PgHero percentage makes it easy to scan for queries that are consuming a lot of resources.

The `pghero` gem was added Rideshare. Run `bin/rails server` from your terminal and visit `/pghero` in your browser to explore it.

Once loaded in your browser, navigate to the *Queries* tab. The queries tab shows information collected from PGSS. The following table is a representation of the data that's presented.

Total Time	Average Time	Calls	
1 min 41%	752 ms	58	postgres
SELECT "users".* FROM "users" WHERE "users"."type" = \$1			

Table 2—PgHero Query Statistics

Normalized query statistics from `pg_stat_statements`

PgHero offers a built-in EXPLAIN (ANALYZE) interface. Queries are safe to execute here because they're placed inside a transaction that's rolled back. A `SET LOCAL` statement timeout of 1 second adds another safeguard.

```
sql/set_local_statement_timeout.sql
SET LOCAL statement_timeout = 1000;
```

The PgHero *Space* section shows the largest tables and indexes. Unused indexes have an *Unused* label on them. This screen is easy to scan and visually identify unused indexes that could be removed. Index maintenance will be covered in an upcoming chapter.

The PgHero *Live Queries* screen shows currently executing queries. This screen uses the `pg_stat_activity` system catalog view you saw in an earlier chapter. Queries may also be terminated or canceled from this screen, which is something you've learned how to do using `psql`.

The *Maintenance* tab shows when tables were last Vacuumed and last Analyzed. We'll do more with the `VACUUM` and `ANALYZE` commands later on.

PgHero is open source; it may be forked and extended with more features (See the [andyatkinson/pghero](https://github.com/andyatkinson/pghero)⁵ fork).

Rideshare uses a forked version of PgHero that makes some small additions. One addition adds a column to the estimated bloat percentage screen. Index bloat is covered in more depth later.

The second addition is a feature that adds visibility into background jobs scheduled using the `pg_cron`⁶ extension. You'll learn more about scheduling jobs directly in PostgreSQL later, too.

5. <https://github.com/andyatkinson/pghero>

6. https://github.com/citusdata/pg_cron

You’ve now seen some ways PgHero can be used as a comprehensive PostgreSQL Performance dashboard for your team. PgHero is so useful that you’ll continue to work with it in upcoming chapters. For now, you’ll focus in on Query Performance.

What other query performance PostgreSQL level visibility tools are available?

Once you have broad query level observability, you’re prepared to identify individual queries to optimize. To optimize queries in PostgreSQL, you’ll need to see how PostgreSQL plans to execute the query. To do that you’ll want the EXPLAIN tool.

Analyzing Query Execution Plans

SQL is a declarative language. When you write a SQL statement you declare the results you want and PostgreSQL determines the best way to get them. PostgreSQL shows you a lot of detail about how it plans and estimates what it will do.

The steps PostgreSQL takes are called the Query Execution Plan. Generating and reading Query Execution Plans is critical to understanding queries, identifying issues, and making improvements. This section will introduce or act as a refresher on how to generate and read query execution plans.

Open up a psql prompt and add EXPLAIN in front of any SQL query. Run this example:

```
EXPLAIN SELECT 1;
               QUERY PLAN
-----
Result  (cost=0.00..0.01 rows=1 width=4)
(1 row)
```

EXPLAIN is a tool with many more arguments like ANALYZE, BUFFERS, and FORMAT that you’ll see how to use. The main purpose is to show some details about the plan it has in mind to fulfill your request.

The PostgreSQL query execution planner is also called the *optimizer*. This name is based on how it works. The planner will compare multiple query execution options against each other when that’s possible, and choose the optimal plan.

Here are some basics about planning. The planner creates plans broken into nodes and each node has a cost. The costs are totaled up and the lowest cost plan is selected. This planning happens at runtime when you run your query and does add a bit of latency to the process. The planning time itself is added

to the plan output. Run the example above as `EXPLAIN (ANALYZE) SELECT 1;` to see Planning Time added to the output.

Reading query execution plans is a good skill to develop for multiple reasons. You'll also begin to understand how PostgreSQL parses a query and handles different pieces like selecting fields, filtering rows based on column criteria, or ordering results, and this can inform your query design and schema design.

You may have noticed the Planning Time is extremely short. To help make this possible, PostgreSQL relies on pre-calculated statistics about tables, indexes, and estimates. You may not have realized that PostgreSQL continually collects statistics about all the data in your database. You can even trigger manual statistics collection yourself.

Stats can be collected by running the `ANALYZE` command for a table. From `psql`, try running `ANALYZE trips;` to give this a spin.

It would be infeasible to run `ANALYZE` manually all the time. Fortunately PostgreSQL runs it automatically. `Analyze` runs automatically by being part of the Autovacuum process that is enabled by default in PostgreSQL.

When PostgreSQL is running, and Autovacuum is triggered for a table based on its trigger conditions, a `VACUUM (ANALYZE)` process runs for each table in the database. You haven't yet learned about the trigger thresholds and how to tune them, but you will later. In normal operations and for Rideshare, table statistics are updated automatically and the default values you have in your local development machine are fine for now.

Now you know how Stats are collected and you saw the `ANALYZE` argument with `VACUUM`.

`EXPLAIN` also supports an `ANALYZE` argument but it has a different purpose in that context.

`EXPLAIN` only shows a query execution plan but doesn't run it. To both show the execution plan and run the SQL query, use `EXPLAIN ANALYZE`.

With all of that in mind, it's time to start diving into query execution plans.

Query execution plans are packed with information. Besides the tables, indexes, and columns there are many more types of data. Some of the information is presented below. Some interpretation information and "next step" questions are listed above each type of item for you to begin thinking about. For now, focus more on seeing what kind of information is listed over the interpretation of the information.

- Costs
- Timing
- Scan Types. Sequential Scans, *Index Scans*, Index Only Scans
- Filtering. How many rows were filtered from a larger set?
- Indexes. Which indexes are named and were used? Are there indexes not listed that you expected to be there?
- Loops. For plan nodes, are loops greater than 1? This means the plan node was repeated.
- Ordering. Is sorting visible in the plan? Indexes are ordered; was an Index used for the sort operation?

For any Index Scans, compare the list of fields in the query to the fields that are indexed. Can the fields being selected in the query be reduced? If the selected fields match the indexed fields exactly, a more optimal *Index Only Scan* can be used over an Index Scan.

The BUFFERS argument to EXPLAIN provides insight about the amount of disk data that was fetched. The amount of data fetched can negatively impact query execution speed. Is it possible to select less data, such as fewer rows by using a LIMIT clause?

Look for TIMESTAMP and TIMESTAMPTZ columns in the query and whether those fields are indexed for the table. When these timestamp fields are covered by a B-tree index they are stored in a chronological ordering that can be useful for ORDER BY clauses to reduce the sort cost.

As you begin reading PostgreSQL query execution plans you may want to use additional tools to help you understand them. Free and commercial analysis tools are available. A query plan visualizer is built in to the PgAdmin⁷ open source PostgreSQL GUI client. A free tool is also available at explain.depesz.com.⁸

PgMustard is a commercial tool for query plan analysis. Capture query execution plans from your planner and then upload the plan text to the PgMustard web client to get deep insights from the execution plan. While the tool is commercial, PgMustard does provide a free public EXPLAIN Glossary⁹ that helps those learning all of the terms and concepts the query planner uses.

You've now started to read query execution plans and have seen some of the information you can learn. Next you'll dive into some specific query performance problems.

7. <https://www.pgadmin.org>

8. <https://explain.depesz.com>

9. <https://www.pgmustard.com/docs/explain>

One of the most common query performance problems is inefficiently filtering down rows. For efficient filtering you'll want your database table to have an Index that matches the fields in the filter criteria.

Without those indexes, you've got a *Missing Indexes* class of performance problem. How do you find those and fix them?

Finding Missing Indexes

Missing Indexes are a type of performance problem where a query filters down a large set of rows but does that without Indexes.

Since there's no index, PostgreSQL has to scan the entire table, called a *Sequential Scan*. For tables with a row count in the millions, and when filtering just one or a few rows in your result, scanning the whole table is inefficient.

How can you discover these opportunities to add indexes and make sure they improve the query performance? One technique is to use a query like `find_missing_indexes.sql`¹⁰ to analyze PostgreSQL and what it's tracked about past queries.

Find Missing Indexes shows the counts of Sequential Scans and Index Scans for a given table. When a table has a lot more Sequential Scans compared with Index Scans, the table might be missing indexes.

PgHero even has a *Suggested Indexes* capability you can use to help spot missing indexes. This tool uses row estimates for the table, and analyzes the existing table indexes and queries in order to suggest indexes to add.

Another tool that can be used is a command line tool. This tool is from the `rails_best_practices`¹¹ gem and can be used to help identify missing indexes. The gem has been added to Rideshare. From your terminal, run `rails_best_practices .` (the period means you're running it for the current directory) to see what it suggests. Vehicle Reservations have a Foreign Key column to Trip Requests, and Rails Best Practices suggests adding an index to that column.

Although you may wish to proactively add Indexes to tables, consider balancing that with a periodic removal of unused indexes. This process will be covered more later.

10. https://github.com/andyatkinson/pg_scripts/blob/master/find_missing_indexes.sql

11. https://github.com/flyerhzm/rails_best_practices

Besides *PgHero*, there are commercial tools like PgAnalyze that can help you identify missing indexes. PgAnalyze offers a free tool called *PgAnalyze Index Advisor*.¹²

Besides manually adding indexes, Index creation can even be automated using some advanced tools. The HypoPG¹³ extension and dexter¹⁴ tool can automatically create indexes.

You've now seen some ways to build up your query observability tool set.

While PGSS is a critical tool, one limitation is that it shows you a normalized query. Once you've identified a specific poor query group with PGSS, you will want to gather a sample from production. From the sample of a real execution of a query in that group, you can look at the EXPLAIN plan to see whether there's a missing Index or something else to improve.

Later, you'll keep diving further into query analysis. But right now a piece in the query observability tool chain is missing.

From PGSS, you'll want to collect real query samples. Once equipped with those samples, you'll be able to dive into their details. Let's see a couple of methods to get those samples.

Logging Slow Queries

PGSS is a critical tool, but you'll need sample queries within a group to analyze further.

One way is to log slow queries into postgresql.log. You can do this by setting the `log_min_duration_statement` parameter to a value like 1 second, to log any query that takes longer. From psql run `SHOW data_directory;` to find the data directory path. In this directory you'll find your postgresql.conf file and the log directory where your log file is located. Run `SELECT pg_current_logfile();` to see the current log file.

In the config file, set the following parameter and save the file. Reload your config with `pg_ctl reload --pgdata=DATADIR`, replacing DATADIR with the value from `SHOW data_directory;`.

```
log_min_duration = 1000
```

12. <https://pganalyze.com/index-advisor>

13. <https://github.com/HypoPG/hypopg>

14. <https://github.com/ankane/dexter>

After making that change, reloading your config, and tailing your log file, run `SELECT pg_sleep(1);` from psql to test it out. If your config is working, you'll see the logged SQL statement that took 1 second in the postgres log file.

```
pid=90062 query_id=440101247839410938: LOG:  |
duration: 1000.963 ms  plan:
    Query Text: select pg_sleep(1);
    Result  (cost=0.00..0.01 rows=1 width=4)
pid=90062 query_id=440101247839410938: LOG:  |
duration: 1001.715 ms  statement: select pg_sleep(1);
```

Follow this approach for your production instance and start by logging very slow queries. This will be a way to begin capturing full queries with parameters using your logs.

Once you have a full query with parameters, you'll want to capture a query execution plan manually by running your query with `EXPLAIN`.

What if you could automatically capture query execution plans?

Automatically Gathering Execution Plans

The PostgreSQL `auto_explain` module can be used to automatically capture query execution plans. You wouldn't want to capture every execution plan, but fortunately it can be configured to capture query execution plans for slow queries.

To limit the execution plan capture to slow queries, you'll open up your PostgreSQL config file again and set another parameter. Set the `auto_explain.log_min_duration` parameter to 1000 as you did earlier for the `log_min_duration_statement`. This means for queries that take 1 second or more the query execution plan will be logged into the PostgreSQL log file.

For Rideshare, queries run much faster. Try lowering it temporarily to 0 so that all queries have logged execution plans.

If you're tailing the log file with `tail -f` and the path to your `postgresql.log`, you'll immediately see the a multi-line logged query execution plan. You'll also see a queryid there, which became more useful in newer versions of PostgreSQL (See: *Using Query ID in Postgres 14*¹⁵). The logged query execution plan should be the same as the one you generated manually using `EXPLAIN`.

With the Query Identifier or queryid you can now connect some of the dots between PGSS, logged queries, and execution plans.

1. Identify the queryid from a normalized query in PGSS

15. <https://blog.rustprooflabs.com/2021/10/postgres-14-query-id>

2. Enable query execution plan logging for slow queries using `auto_explain`
3. Find matching queryid values in the `postgresql.log`

Make sure `compute_query_id` is set to on in your PostgreSQL config file.

Once you've found a match, you have the query as executed with real parameters that you can analyze using PostgreSQL EXPLAIN.

You now have a game plan to find high frequency and slow queries with PGSS, and then either manually or automatically collect samples from the log, including manual or automatic collection of the query execution plan. You haven't yet seen how to use the query plan, but that will be coming soon.

You're just about ready to dive into execution plans.

Before doing that, there's one more thing to consider before working on query optimization. PostgreSQL can require some database maintenance for tables and indexes that are related to the slow query. Before spending a lot of energy on query optimization, make sure the tables and indexes are optimized.

Performing Maintenance First

Before diving too far into query execution plans make sure the tables and indexes related to the query are optimized.

Perform some of the generic database maintenance steps when you're facing a specific slow query.

1. For each table in the query run a manual VACUUM (ANALYZE) from psql for the table. This makes sure Vacuum has run recently and statistics are updated.
2. For any indexes used by the query, determine whether they have a high percentage of estimated bloat. If so, rebuild the index.
3. If you see Autovacuum or Analyze not running regularly for Indexes with very high bloat across the board, it may be time to schedule supplemental Vacuum, Analyze, and Reindex operations. You'll learn more about that later.

Refer to the *Show database bloat*¹⁶ scripts from the PostgreSQL Wiki. These scripts help you identify indexes with a very high estimated percentage of bloat.

Now that you've made sure the tables and indexes are in good shape, you're ready to dive into query optimization.

16. https://wiki.postgresql.org/wiki/Show_database_bloat

Interpreting Index Scan Execution Plan Info

One of the main use cases for Indexes is efficient lookup and filtering. When a column `user_id` is listed in a `WHERE` clause, make sure there's an index for the table covering `user_id` either as a single column index, or as part of a Multicolumn index where `user_id` is the first column listed.

PostgreSQL won't always choose to use your index until row counts are high, and when a single row or a small set of rows are being requested. PostgreSQL uses the selectivity of the query, cardinality for the column, and compares the costs between scanning the whole table, fetching from the index and table, or using the index alone to determine what's best.

When the index is used, the planner shows an *Index Scan* and lists the name of the index.

The basics of index data structure internals are as follows. Each item in the Index is a reference to a table row. Scanning an Index is typically a faster way to collect a list of row pointers compared with scanning all rows for a large table.

Table access is slower because the data is spread out in random locations. Indexes, on the other hand, are ordered and designed for fast retrieval. Because of this ordered storage characteristic, besides filtering, Indexes can also be used for sorting with `ORDER BY` operations.

The values of a node within the index include a `ctid`. The `ctid` is a pointer to the disk location of the row data. You can add the `ctid` to your query to see the value.

Besides Index Scans, there is an even more efficient scan type related to indexes.

With *Index Only Scans*, PostgreSQL fetches all of the data it needs from the index itself directly without needing to access the heap where the rows are.

Index Only Scans are not supported for all index types. B-tree indexes always support Index Only Scans. Typically B-tree indexes are used for equality comparisons such as comparing the exact value of `user_id = 1`. An index covering all values of `user_id` will be traversed until the value is found.

For now, Indexes can be put into two major categories. One category is a general purpose index and the second category would be a “specialized” index. A general purpose index is useful for many queries and operation types while a specialized index might be designed to be optimal for one query.

Indexes trade off faster retrieval speed for more disk space consumption. When performance is critical it may be worth creating specialized indexes.

As a brief introduction to specialized indexes, “Multicolumn Indexes” and “Covering Indexes” are specialized index types that can be used as part of your query optimization tactics.

The query planner may scan the entire table as a Sequential Scan or may use an Index and perform an Index Scan. Is there a middle ground between table scans and Index scans? There is – Bitmap Scans split the difference.

EXPLAIN Scan Nodes and Bitmap Scans

Let’s see how Bitmap Scans are used. The *Database Page Layout*¹⁷ page is extensive and beyond the scope of the chapter, but some basics will be covered here.

PostgreSQL table data (the *heap*) is represented in a *Page*. The bitmap PostgreSQL builds is one it creates on the fly by visiting every Page, and marking whether a Page contains a row and should be scanned or not.

The scan type is something PostgreSQL chooses but is exposed in the query execution plan.

Bitmap Scans can refer to either the Heap (table) or an Index. A Bitmap Index Scan (See: *EXPLAIN - Bitmap Index Scan*¹⁸) is like a middle ground between a slow scan of the entire table and a fast Index Scan. When would a Bitmap Index Scan be used and not an Index Scan? One scenario would be when an Index would be useful but it doesn’t exist covering the fields being filtered on.

On the other side of things with very large result sets, PostgreSQL may choose to not use an Index Scan for an index that exists because it might be faster to scan the entire table before filtering.

A Bitmap Heap Scan builds a bitmap data structure from a heap scan. A Bitmap Index Scans builds on top of that.

The bitmap is one bit per heap page. (See: *Understanding “bitmap heap scan” and “bitmap index scan”*¹⁹)

17. <https://www.postgresql.org/docs/current/storage-page-layout.html>

18. <https://pganalyze.com/docs/explain/scan-nodes/bitmap-index-scan>

19. <https://dba.stackexchange.com/a/119391>

The planner will show the cost of building the bitmap with a Bitmap Heap Scan. A Bitmap Index Scan will show in the execution plan as a Scan Node when a Bitmap is used.

In this section, you were introduced to Bitmap Heap Scans and Bitmap Index Scans. When you see them in query execution plans, you'll now a little more about how PostgreSQL is trying to efficiently filter a set of rows. From there you may explore terminology in greater depth and whether an even more efficient method for filtering is possible given a specific query and specific table structure and row count.

Besides inefficient filtering operations, another type of performance problem is selecting too many rows. How can you find and address instances of that problem?

Adding Query Boundaries With Filtering and LIMIT

In Ruby on Rails applications operating at scale, you'll want to make sure queries in high frequency use cases filter to the smallest set of rows possible.

Try to add a LIMIT clause to most queries. Besides a LIMIT, try and add more boundaries to queries. Extra indexed fields helps the planner cut down the initial set of rows to scan.

In a multi-tenant application a `tenant_id` or `customer_id` column could be added to tables and indexed, so that a WHERE clause can be scoped just to those rows.

If your users are spread out among geographic regions or sales regions, consider grouping them into a regions table and scoping their records using an indexed `region_id` Foreign Key column.

Perhaps you can filter on Active and Inactive users.

Besides filtering in the WHERE clause, remember to add more conditions to JOIN operations.

For web applications, sometimes you'll need to have upper boundaries set and then introduce Pagination or Searching as user experience workarounds. You'll explore pagination techniques like Keyset Pagination later.

In the next section you'll move on to counting items efficiently.

Performing Fast COUNT() Queries

COUNT(*) queries are known to be slow, as explained in *Faster PostgreSQL Counting*.²⁰ They scan the whole table for accurate results. Counts might be for all rows in a table or for a filtered set of rows.

You can achieve fast counts for both unfiltered and filtered scenarios, but you'll need to use some tricks.

The easiest trick is to use the count estimates that PostgreSQL tracks for each table instead of running a COUNT() query. These estimates will be off a bit but are incredibly quick to fetch.

Open up psql from your terminal for the Rideshare database. Run the following SQL statements. Since this is counting all tables named users, drop the temp.users table if you still have it, by running DROP TABLE temp.users CASCADE; to get a single count estimate.

```
sql/estimated_count_reltuples.sql
-- Update stats
ANALYZE users;

-- Use reltuples for estimate
SELECT reltuples::NUMERIC FROM pg_class WHERE relname='users';
```

The first statement updates the statistics for the users table and the second statement fetches the estimated row count.

The fast_count²¹ gem has been added to Rideshare, which enables faster counting for Active Record. The gem adds a Migration that adds a database function for fast counts.

For Rideshare, run bin/rails console and try out the User.fast_count method. Compare it with User.count for speed and accuracy.

```
User.fast_count
```

Although you haven't yet worked in detail with Query execution plans, the execution plan text itself can be used as a data source.

This trick uses the query execution plan to get fast estimated filtered counts. Try running bin/rails console in the Rideshare directory and comparing these two Active Record snippets that count filtered sets of Users. If you ran the data generator you should have 10 million users.

```
User.where("last_name ILIKE 'lname123%'").count
```

20. <https://www.citusdata.com/blog/2016/10/12/count-performance/>

21. https://github.com/fatkodima/fast_count

```
User.where("last_name ILIKE 'lname123%']").estimated_count
```

The `estimated_count()` method is defined in the “fast_counts” gem. The estimated count is a bit off from the real count value but is tremendously faster to get.

Finally, if you need to count distinct values, the PostgreSQL wiki documents some tricks on how to do this quickly (See: *Selecting Distinct Values*²²). Fast distinct counts are being worked on as an addition to the fast_count gem at publication time.

Another advanced technique is to use the postgresql-hll extension. The HyperLogLog (HLL) concept is used to estimate the number of distinct values in a large set. The HyperLogLog is described as a *cardinality estimator*.

After installing and enabling the PostgreSQL extension, the HyperLogLog functionality may be used via the active-hll²³ gem. Try configuring it with Rideshare and installing the gem if you’d like to explore it.

If estimates or these tricks won’t work, you may need to manage your own tally of rows at Insert time. The general technique is similar to how the Active Record Counter Cache works.

When reporting multiple COUNT values a CASE statement may be used. The CASE statement is used to set up different integer values and then sum the values.

An alternative to using SUM with CASE and WHEN is the lesser-known FILTER keyword, which can also be used to display multiple counts at once. The post *The FILTER clause in Postgres 9.4*²⁴ shows how to use FILTER for “scoped aggregate functions.”

Try this out for Rideshare. Run this statement from psql:

```
sql/count_with_filter.sql
```

```
SELECT
  COUNT(*) FILTER (WHERE type = 'Rider') AS rider_count,
  COUNT(*) FILTER (WHERE type = 'Driver') AS driver_count
FROM users;
```

The query above displays the counts of Riders and Drivers. Although it scans the table, it does only one scan.

You’ve now seen some ways to improve the performance for count operations. Next you’ll look at some tools that will help you with your database design.

22. https://wiki.postgresql.org/wiki/Loose_indexscan

23. https://github.com/ankane/active_hll

24. <https://medium.com/little-programming-joys/the-filter-clause-in-postgres-9-4-3dd327d3c852>

Using Code and SQL Analysis Tools

Here you'll see how the rails-pg-extras²⁵ Ruby gem and command-line tool can be used to help with schema design and query design. The gem has been added to Rideshare.

In your terminal, run `bin/rails pg_extras:cache_hit` to look at index hit rate and table hit rate ratios. You want to see high hit ratios for both types.

A low index hit ratio could indicate there are too few indexes. PostgreSQL keeps a cache of indexes in memory and a low hit ratio means there are many cache misses. Refer to the gem documentation for more information.

Next run the `bin/rails pg_extras:diagnose` command to get an overall report for Rideshare. Good items are printed in green and improvement areas are printed in red.

Indexes may be skipped for usage in execution plans when they're suffering from very high estimated bloat. As you saw earlier, the solution in that case would be to rebuild the index.

Another analysis tool you saw earlier is `pgbadger`. `PgBadger` can be also be used for historical query analysis.

Queries are presented in P90, P95, and P99 percentile groupings. Focus on the P90 percentile first, before tackling optimization opportunities for less common P95 and P99 percentiles.

Wrapping Up

This chapter focused on helping you build your query observability tool chain. You automatically logged slow queries from the application. You found high impact slow queries using PGSS on the server. You learned how to manually and automatically collect samples and execution plans. You started to identify common performance problems like Missing Indexes and tactics to fix those problems.

You didn't quite yet dive into queries in depth, but you gained comfort modifying parameters and using the PostgreSQL log to identify information. You configured PGSS and `auto_explain` extensions to help you identify and fix problems.

You started to scratch the surface of query execution plans and how Indexing works.

25. <https://github.com/pawurb/rails-pg-extras>

Next you'll cover Indexes offered by PostgreSQL in greater depth. You'll explore general purpose and specialized index types for different types of queries. You'll keep working with query execution plans expanding on what you started with in this chapter.

Optimized Indexes For Fast Retrieval

In the previous chapter you learned how Indexes can be used for filtering operations. Now you're ready to do more with Indexes, going into greater depth on the types of indexes available in PostgreSQL, which fields to cover, and which types are suited best for certain queries. You'll expand on how indexes can be general purpose or have specialized purposes.

You'll create single-column indexes, multicolumn indexes, partial indexes, indexes on expressions, and Covering indexes. As usual, there is a lot of new terminology to learn with Indexing.

Besides speeding up queries for data retrieval from filter operations, indexes are used to enforce UNIQUE, PRIMARY KEY, and EXCLUSION constraints. Indexes store data in sorted order so they can also be used for ORDER BY operations.

Take a look at some of the Indexing terminology you'll encounter.

Indexes Terminology



- Index Definition — How the Index was created
- Multicolumn Indexes — Index definitions that cover multiple columns
- Partial Indexes — Index definitions with an expression that reduces the covered rows
- Covering Indexes — Index definitions that cover all columns for a query

You'll work with some less common Index types, but first up is a refresher on the default Index type, which is the most common type.

The default and most common Index type used is a B-tree Index.¹ A B-tree index is a secondary data structure represented as a tree optimized for retrieval speed.

B-tree indexes are used for equality comparisons. Each node points to a row. A query using the Index may be searching for rows with matching column values to filter down a result set.

When an Index was scanned and used to identify the row, as you saw earlier the query execution plan will name the index and show that an Index Scan was performed. The index contains a reference to the row identifier. Using the row identifier called the ctid, the table or heap is then scanned to retrieve the row with the columns specified in the query.

With that refresher in place, it's time to work with a less common Index that transforms data before being stored.

Generating Data for Experiments

One of the best ways to learn about the behavior of PostgreSQL is to create experiments by generating test data and performing queries and running commands on large sets of data. To work with Indexes locally on large sized data, open up psql from your Terminal and run these SQL statements:

```
sql/insert_users_generate_series.sql
CREATE SCHEMA IF NOT EXISTS temp;

DROP TABLE IF EXISTS temp.users;

CREATE TABLE IF NOT EXISTS temp.users AS
SELECT
  'fname' || seq AS first_name,
  'lname' || seq AS last_name,
  'user_' || seq || '@' || (
    CASE (RANDOM() * 2)::INT
      WHEN 0 THEN 'gmail'
      WHEN 1 THEN 'hotmail'
      WHEN 2 THEN 'yahoo'
    END
  ) || '.com' AS email,
  CASE (seq % 2)
    WHEN 0 THEN 'Driver'
    ELSE 'Rider'
  END AS type,
  NOW() AS created_at,
  NOW() AS updated_at
FROM GENERATE_SERIES(1, 10000000) seq;
```

1. <https://www.postgresql.org/docs/current/btree-implementation.html>

These statements create a temp.users table on the fly from a SELECT statement inside the temp schema you already have or will create. The temp.users table is similar to the Rideshare users table but with fewer columns. You'll add indexes covering some of these columns.

To start, there will be no indexes on the temp.users table. This makes it faster to insert rows because no Index maintenance will be performed as rows are inserted, updated, or deleted.

Once you have the table created and populated with 10 million rows, you're ready to begin creating experiments.

When filtering rows for tables with low row counts, PostgreSQL may decide to scan the whole table instead of using an Index you've created when it needs to find a few rows for a query. This can be surprising behavior but makes sense as you learn more about Indexing.

You may already know that Index Scans will outperform a Sequential Scan (table scan) when filtering for a row or two. You may not know that when items are found in an index and the PostgreSQL query plan shows an "Index Scan", that PostgreSQL doesn't usually get all of the information needed solely from the Index. PostgreSQL still needs to access the "Heap"² (which is the table row data) after the Index most of the time. That means an Index Scan might be two scans, a Index Scan and a Heap Scan. The combination of those two scans might be slower than scanning the table directly. This is dependent on many factors.

The most optimal scan type is an Index Only Scan. This happens when all of the fields needed either from a SELECT clause or a filtering option are contained in the Index itself. In that case a Heap scan is not necessary, which saves a considerable amount of time.

Enabling Index Only Scans is one of the reasons you saw earlier to limit the fields you're requesting in your Active Record query code.

With those explanations in place, you're now ready to begin experimenting with the temp.users table you've just created.

From psql run this query:

```
SELECT * FROM temp.users
WHERE last_name IN ('lname10000', 'lname100000', 'lname1000000');
```

2. <https://www.postgresql.org/docs/current/glossary.html>

Without an index this query might take a long time. From psql, run the query again with EXPLAIN (ANALYZE, BUFFERS) in front of it. On PostgreSQL 15 this uses a Parallel Sequential Scan for the temp.users table, which is quite fast but takes more than 500ms and reads over 100,000 buffers. In a web application context where this query was running at a high volume, this time might be unacceptably slow and cause a lot of unnecessary disk access.

You'll fix this by adding an index to the temp.users table. Since you're filtering on the last_name field, add an index to that field.

Since you're adding the index to a non-live system, skip the CONCURRENTLY option – it's not needed and would slow down the creation unnecessarily. To see how long it takes to add, toggle on timing by running the \timing command. With timing enabled, run this statement:

```
CREATE INDEX users_last_name_idx ON temp.users (last_name);
```

After adding the index, analyze the table using the ANALYZE temp.users statement from psql.

Run the query again using EXPLAIN (ANALYZE, VERBOSE). The query now runs in less than 5 milliseconds and there is only one Buffer that's read. The index you just added made this query very efficient with a very low cost and low disk usage. You'll see the name of the index "users_last_name_idx" and a Index Scan as the scan type.

As a Rails developer, you use Active Record Migrations to create indexes, keeping all of the databases the application works with in sync.

Create an Active Record Migration for this Index Creation. This way the index will be created on your local development machine, in any lower environments like CI or staging, and in your production environment.

An example Migration is listed below. This Migration uses the if_not_exists: true option which adds the index only if it doesn't exist. This way you're able to add the Index manually if necessary to production, but backfill a Migration to keep everything in sync. The Migration will run on a live system so the CONCURRENTLY option is used in the conventional way for Active Record.

```
ruby/add_index_users_last_name.rb
```

```
class AddIndexUsersLastName < ActiveRecord::Migration[7.0]
  disable_ddl_transaction!

  def change
    add_index :users, :last_name, algorithm: :concurrently,
      if_not_exists: true
  end
end
```

If you're newer to Active Record or haven't worked with it as much, you might be wondering how Ruby generates SQL. Read on to learn more.

Transforming Ruby to SQL

As a Rails developer you may have only created indexes using Active Record Migrations and be less familiar with the SQL statements they generate.

In the last section you saw a CREATE INDEX statement in SQL by creating it with SQL first and then an example of how it's created with Ruby.

You may wish to explore the definitions as creation statements for all of the Indexes when removing and adding back Indexes. PostgreSQL tracks the index creation statements internally in system catalogs. Query the pg_indexes catalog from psql as follows to take a look.

```
SELECT indexdef FROM pg_indexes WHERE tablename = 'users';
```

The statement above shows the index definitions as creation statements for all indexes on the public.users table.

Displayed this way, the creation statements do not contain the CONCURRENTLY keyword even though they may have been created using it.

On your non-live system Rideshare database, practice with more SQL Index modifications. Try dropping an index. Remember that your local development database is a great place to practice. You can always reset the Rideshare schema completely to the original configuration.

From psql, run the DROP INDEX statement below to drop an index named “index_users_on_last_name”.

```
DROP INDEX index_users_on_last_name;
```

Dropping Indexes in a live system is also an operation that should use the CONCURRENTLY keyword:

```
DROP INDEX CONCURRENTLY index_users_on_email;
```

If there are other transactions using the index, CONCURRENTLY avoids introducing long-running locks that would block those transactions.

Now that the indexes are dropped, try manually creating them using the SQL statement and the CONCURRENTLY keyword. Make sure that \timing is enabled.

Why Are My Indexes Not Being Used?

As you saw earlier, even when you've added an Index you want PostgreSQL to use, it may choose to not use it. You saw that this can depend on many factors. Let's look at some of those factors in more depth.

Queries should be as selective as possible, meaning they select as few rows as possible, for your database server to scale better and achieve a higher volume of queries. Most of the Active Record queries you're writing or running are in an online transaction processing (OLTP) Ruby on Rails web application context where short durations are important.

Besides very selective queries, consider the cardinality of the columns you're working with. The number of distinct values in a column is the "cardinality." High cardinality means there are a lot of distinct values.

Indexes are ideal for high cardinality columns and highly selectivity queries. When there is a table index for a column with low cardinality, or in other words, the column has very few distinct values such as two or three total possible values, PostgreSQL may prefer a Sequential Scan over an Index Scan. In that case the Index Scan and Heap Scan may have a higher total cost compared with a Heap Scan only.

In the next section you'll begin to consider the definitions of your indexes. Indexes can cover single columns and multiple columns. Read on to learn more.

Using Single Column And Multiple Column Indexes

*Multicolumn Indexes*³ are indexes in PostgreSQL that include multiple columns in their definition. Terminology like "composite" or "compound" indexes from other relational databases are synonyms for Multicolumn Indexes in PostgreSQL.

Open the Rideshare database with psql. Run `VACUUM ANALYZE` without listing any tables to run it for all tables. Use the Index Maintenance query from the PostgreSQL wiki https://wiki.postgresql.org/wiki/Index_Maintenance to display information about all indexes for Rideshare grouped by tables.

Multicolumn Indexes are considered a specialized index. Most Indexes in Rideshare are single column indexes.

3. <https://www.postgresql.org/docs/current/indexes-multicolumn.html>

A database with a large amount of Multicolumn indexes could indicate excessive or redundant indexing, where single column indexes would serve the same purpose and consume less space. You'll want to look for overlapping indexes for possible optimization. We'll cover that later.

On the other hand, Multicolumn Indexes can provide some of the best performance when each of the columns covered in the definition are used by a query. Columns are used from left to right. The leftmost column in a Multicolumn Index definition can be used by queries that only need that single column. Soon you'll learn how to make sure all columns of Multicolumn Indexes are being used.

On the same PostgreSQL wiki page, use the “Index size/usage statistics” query for the Rideshare database. This query will show usage information.

To simulate some usage of Rideshare, from your Terminal run `bin/rails server` from the Rideshare directory to start the Rails Server. From another Terminal window, run `bin/rails simulate:app_activity` which will send some requests to the Rideshare server.

With that completed, you will now have some Index usage information from queries that were executing from that simulation. The number of scans shows a count of how many Index Scans have been performed using this index. The tuples fetched and tuples read show information on the number of tuples provided by this index. Indexes with no scans and no tuples fetched may be unused or infrequently used indexes.

Creating good Indexes for your database is challenging because it requires knowledge of multiple factors including the data statistics, the query patterns, and the Index types available.

Is the order of how columns appear in the Index significant?

Understanding Index Column Ordering

The order of columns covered by a Multicolumn Index is significant.

Consider a Multicolumn index that covers columns “a” and “b”, ordered (a, b) (a then b) from the left to the right. Which types of queries can use this Index? Queries with column “a” and “b” can use this index, or queries with only column “a” can use this index. Queries with only column “b” cannot use this index. Only the first column of a Multicolumn index can be used.

This makes Multicolumn indexes a type of specialized index. They are useful for specific queries that work with multiple fields, when those fields are in

the correct order. Multicolumn indexes are also useful in other scenarios, but only the first column for queries that work with that column.

For a query with column “b” that above Index won’t be used. What if your application needs an index to efficiently filter on column “b”? PostgreSQL recommends creating separate indexes for each of column “a” and “b”.

The documentation page *Combining Multiple Indexes*⁴ demonstrates how multiple Indexes can be scanned for a query, or even multiple scans of the same Index can be used to combine Index data for AND and OR conditions.

For the example above, having single column indexes on “a” and “b” and a query that filters on A and B, would allow PostgreSQL to combine the results of Index scans on each of the single column indexes.

Since Indexes add some latency to all DML (Inserts, Updates, and Deletes) operations, it’s worth avoiding redundant indexes. This implies that having all three indexes mentioned so far, a single column index covering “a”, a single column Index covering “b”, and a Multicolumn index covering “a” and “b” is redundant.

On the other hand, a Multicolumn index that covers two columns where the first column is for filtering and the second column is for ordering, may provide the best query performance. You may wish to have specific redundancy like this when your Multicolumn index is specialized for certain queries, and single columns indexes are both used for others.

What types of Indexes are best for columns with boolean values?

Indexing Boolean Columns

Indexing columns that store boolean values has some pitfalls. The column may have three possible values when nulls are allowed, otherwise it will have only two possible values.

Imagine a column called `is_deleted` that’s false by default, does not allow nulls, and is set to TRUE when the row is soft deleted. The majority of the rows will have the column value of false.

Adding an index to the `is_deleted` column may not be helpful depending on the queries. When querying for false values, which are the majority of the rows, PostgreSQL will likely scan the whole table with a Sequential Scan to filter out rows with true values that are Soft Deleted rows.

4. <https://www.postgresql.org/docs/current/indexes-bitmap-scans.html>

Indexes work best when they are highly selective, meaning they select a small proportion of the total number of rows. If only 10% of the rows have a value of true, then a query that filters on that boolean column for True values will likely produce an Index Scan and perform well.

Now you're ready to work with some of the less common Index types. First up is the Expression Index.

Transform Values with an Expression Index

An Index that covers a column may transform the column value before being stored. When this type of transformation occurs the Index is called an Index on an Expression. This may also be called a Functional Index because a database function is being called, like LOWER().

Imagine that want to search users by email address in a case insensitive way. One solution would be to transform the email address using the LOWER() function on the column before it's stored. Use the temp.schema in Rideshare that you set up earlier for the example you'll find next. If you need to recreate this schema and table, refer to the code example called "insert_users_generate_series.sql" from an earlier chapter. When you've done this, you'll have the temp schema and temp.users table with 10 million rows populated.

Run this SQL statement from psql to insert more rows into temp.users:

```
INSERT INTO temp.users (
  first_name, last_name, email, type, created_at, updated_at
)
VALUES
  ('Jane', 'Example1', 'Jane@example.com', 'Driver', NOW(), NOW()),
  ('jane', 'example2', 'jane@example.com', 'Driver', NOW(), NOW());
```

Email addresses are case insensitive, so you've created duplicate email addresses now in your system. You want know which Jane should get an email, whether it should be jane% or 'Jane%'. This creates a problem and you can use a Unique Index on an Expression to solve this.

First, remove the users you just created by running this statement:

```
DELETE FROM temp.users
WHERE email IN ('Jane@example.com', 'jane@example.com');
```

After improving your data model by validating that you only accept unique case insensitive email addresses, you'll try and add the users back.

To improve your data model, add the following Unique Index that has an expression, calling the LOWER() function on the email column. Run \timing to toggle timing if you wish to see how long it takes to add this index.

```
CREATE UNIQUE INDEX index_temp_users_lower_email_unique
ON temp.users (LOWER(email));
```

By lowercasing the input value for the email column, you'll be able to compare emails in a case insensitive fashion.

Now we can move on from Indexes on Expressions. You'll work with the GIN index type and see how it can be used.

Using GIN Indexes with JSON

GIN stands for “Generalized Inverted Index.” In the B-tree index type, each item points to a specific row. The ctid row identifier points to a physical location on disk.⁵

GIN indexes are different. For a GIN index, each index entry points to all occurrences for a value where each occurrence is in a different location. A GIN index is more like the index found at the back of a book that lists a term from the book, and all the pages where it appears.

When is a GIN index better than a B-tree index? A GIN index can provide faster retrieval when a given column contains multiple values within it.

In a json or jsonb field, many values can be contained within a single column. Data structures like key-value or even nested key-value structures can be stored as JSON compatible strings in a single column.

Thus, a GIN index can be used to index values within the JSON string in ways that don't work with B-tree indexes. A B-tree index on a JSON column only supports equality matching.

Another type of nested data are arrays placed in a column. A GIN index makes it possible to efficiently query items in the array.

Here's an example using Rideshare. Imagine that the business wishes to collect additional metrics on rides. They want to know how many bags each driver is picking up. This might help with dispatching the properly sized vehicle. They might want to know whether Drivers have the music on or not, and whether a water bottle is offered to the Rider. These data points may be represented as strings, booleans, or numbers.

5. <https://www.postgresql.org/docs/8.2/ddl-system-columns.html>

Since it's unclear whether these additional data points are needed for the long term, and since they're unlikely to ever be joined on, you decide to store this data as JSON compatible text in a denormalized document style representation. Another benefit of this approach is that you'll be able to append more attributes to this list over time. On the downside, you're wondering whether you'll be able to efficiently query this data or not when the row counts in this table grow.

To store JSON compatible strings you can choose between column types of `json` or `jsonb`. JSONB supports indexes on the data so you decide to use JSONB.

Additional details about the attributes are:

- `bags_in_trunk` (a number)
- `music_on` (true or false)
- `water_offered` (true or false)

In the snippet below, `\gset` is used to create a *dollar quoted literal* (See: *Defining multi-line strings in psql*⁶).

This is like a Heredoc or multi-line string in Ruby and is used here to better format the JSON string for printing purposes.

After making the variable `json_string` in `psql`, the variable is referenced in the statement `SELECT jsonb_pretty(:json_string);`.

Open `psql` and run this command:

```
SELECT
$$'{
  "ride_details": {
    "bags_in_trunk": 1,
    "music_on": true,
    "water_offered": true
  }
}'$$ AS json_string \gset
```

You've now set the `json_string` variable. Referencing `:json_string` inside the `JSONB_PRETTY()` function below will format it nicely. Besides formatting, this also ensures the JSON is valid.

Run this statement in `psql`:

```
SELECT jsonb_pretty(:json_string);
      jsonb_pretty
```

6. <https://stackoverflow.com/a/31457183>


```

{
    "ride_details": {
        "music_on": true,
        "bags_in_trunk": 1,
        "water_offered": true
    }
}
(1 row)

```

The following SQL file performs a few actions. First you'll create three JSON variables as JSON formatting strings that represent the data source for the additional attributes.

To store the data, you'll modify the Rideshare trips table and add the data column as a new nullable jsonb column. The SQL statement does assume that trips with primary keys 1, 2, and 3 exist. The dollar quoted literal technique and line breaks are used for legibility.

Run these statements from psql:

```

sql/jsonb_trip_details_multi_insert.sql
-- Populate json_string1, json_string2, and json_string3
SELECT $$ '{"ride_details":
{"bags_in_trunk": 1, "music_on": true, "water_offered": false}
}'::jsonb$$
AS json_string1 \gset

SELECT $$ '{"ride_details":
{"bags_in_trunk": 2, "music_on": true, "water_offered": true}
}'::jsonb$$
AS json_string2 \gset

SELECT $$ '{"ride_details":
{"bags_in_trunk": 3, "music_on": false, "water_offered": true}
}'::jsonb$$
AS json_string3 \gset

-- Add the `data` column to the trips table
ALTER TABLE trips ADD COLUMN data jsonb;

-- Insert 3 json blobs from above into the `trips`
-- table, for ids 1, 2, 3
--
-- use UPDATE ... FROM for multi-row update
-- https://stackoverflow.com/a/18799497
UPDATE trips AS t
SET data = c.json_string
from (VALUES
    (:json_string1, 1),
    (:json_string2, 2),
    (:json_string3, 3)
) AS c(json_string, trip_id)

```

```
WHERE c.trip_id = t.id;
```

Query the following Trips rows by id and confirm their data column is populated.

```
SELECT * FROM trips WHERE id IN (1,2,3);
```

With the JSON string data populated, you're now able to query it. Select rows where ride_details is present in data by running the following statement in psql. Running the query with EXPLAIN shows that a Sequential Scan of the trips table is being used.

```
EXPLAIN SELECT * FROM trips WHERE data ? 'ride_details';
```

With the data populated, you're now ready to Index it and take advantage of the GIN index type. Create the following GIN index by running the statement in psql and then Analyzing the table.

```
CREATE INDEX ON trips USING gin(data);
```

```
ANALYZE trips;
```

Run the query again with EXPLAIN and notice that a Bitmap Index Scan on trips_data_idx is now used instead of a Sequential Scan.

Now try performing an aggregation. Get the average number of bags in the trunk on trips. This might help inform you whether to send cars with bigger trunks at certain times of the day.

```
sql/avg_bags_in_trunk.sql
```

```
SELECT AVG((data->'ride_details'>'bags_in_trunk')::integer)
FROM trips WHERE data ? 'ride_details';
```

Next you'll look at two more indexing options as alternatives to the GIN index you've seen so far.

PgAnalyze describes a containment query (See: *Understanding Postgres GIN Indexes: The Good and the Bad*⁷) example that can be adapted to Rideshare. First add the index, specifying the jsonb_path_ops class in the index creation statement.

```
CREATE INDEX trips_data_path_ops
ON trips USING gin(data jsonb_path_ops);
```

Now, you can answer questions about trips using a containment query like “Which drivers offered a water bottle to Riders?” or “Which trips had two or more bags in the trunk?”

7. <https://pganalyze.com/blog/gin-index>

```
sql/query_json_operators.sql
-- water was offered
EXPLAIN SELECT * FROM trips
WHERE data @> '{"ride_details":{"water_offered": true}}';

-- 2 bags in the trunk
EXPLAIN SELECT * FROM trips
WHERE data @> '{"ride_details":{"bags_in_trunk": 2}}';
```

You'll continue to expand your usage of these new analytics about trips.

Correlate these ride analytics with particular drivers and locations to help answer Rideshare operational questions. To do that you'll explore B-tree indexes with JSON data. B-tree Indexes can be used with JSON data by creating an expression as you did earlier.

Run the following statement in psql to create a B-Tree Index on the data column for expressions on specific attributes. In the case below, the index narrows down all trips where two or more bags in the trunk were recorded.

```
CREATE INDEX ON trips USING btree(data)
WHERE (data->'ride_details'->'bags_in_trunk')::int4 >= 2;
```

With that index in place, the following query can be run efficiently. Confirm the index you just added was used by prepending EXPLAIN. Run \d trips to confirm the index names correlate to the Indexes you expect.

```
EXPLAIN SELECT * FROM trips
WHERE (data->'ride_details'->'bags_in_trunk')::int4 >= 2;
```

When deciding on JSONB with Ruby on Rails for row storage, consider using the store_accessor functionality for the Active Record model described in *Hidden Gems: ActiveRecord Store storage*.⁸

One trade-off to be aware of with Store Accessor is that it's limited to a single level of depth. The above structure where bags_in_trunk is nested under ride_details wouldn't work because of more than one level of depth. For that, use the JSONB capabilities directly from Active Record.

You've now seen how you can use GIN indexes for use cases like efficiently looking up values in JSONB columns.

What else can GIN indexes be used for?

Using GIN Indexes for Full Text Search

You can use a GIN index to provide full text search capabilities to Rideshare.

8. <https://www.honeybadger.io/blog/rails-activerecord-store/>

You'll work with LIKE queries and develop a similarity search solution using GIN indexes and trigrams (See: *Postgres GIN Index in Rails*⁹). Trigrams can be captured and a GIN index can be used to accept a user's search input and find similar text. Trigrams will help create matches on similarity. The query planner will use a bitmap index scan with this type of GIN index.

Run the following query from psql that performs an ILIKE query for some rows in the users table. The table should be populated from the data generators and have at least 20 thousand rows in it. If you wish to reset the Rideshare database, run `bin/rails db:reset` and then `bin/rails data_generators:generate_all` from your Terminal.

```
sql/query_ilike.sql
```

```
SELECT DISTINCT first_name FROM users WHERE first_name ILIKE 'henr%';
```

ILIKE is being used for case insensitive similarity searching. Lowercase "henr" is used as the partial matching text. Passing the text "henr" with case insensitivity matches on Henrietta, Henriette, and Henry.

You should get those three results when you run the query above. The query performed a Sequential Scan of the whole table. You can speed that up using a GIN index.

You'll enable an extension and add the index using a Active Record Migration.

The following Migration has not been added to Rideshare. To add it, generate a migration file by running this command from your terminal:

```
bin/rails g migration AddTempUsersGinTrgmOpsIndex
```

In the generated Migration file, copy and paste the implementation body from the example below. The Migration will enable the extension and then add the index.

```
ruby/migration_similarity_search.rb
```

```
class AddFirstNameSimilaritySearch < ActiveRecord::Migration[7.0]
  disable_ddl_transaction!

  def change
    enable_extension('pg_trgm')

    add_index(:users,
              :first_name,
              name: 'users_first_name_gin_trgm_ops_idx',
              using: 'gin',
              opclass: :gin_trgm_ops,
              algorithm: :concurrently)
```

9. <https://blog.kipprosh.com/postgres-gin-index-in-rails/>

```
end
end
```

Run the following command in your Terminal to apply the Migration.

```
bin/rails db:migrate
```

After adding the index, open `psql` for Rideshare and run `\d users` to confirm that you now see a new GIN index covering the `users.first_name` field.

After adding the index and a bit of warm-up, the same `ILIKE` query from above should be much faster! Confirm the index is used by adding `EXPLAIN ANALYZE` in front of your query in `psql`.

Next you'll see another new concept. PostgreSQL helps you explore the similarity and distance between words. Run the query below to see how similar “henr” is to each of the names. The distances between “henr” and the name can also be printed. The `%` operator¹⁰ is being used in the `WHERE` clause to filter the results.

```
sql/query_similarity_distance.sql
```

```
SELECT
  distinct first_name,
  similarity(first_name, 'henr') AS similarity,
  first_name <-> 'henr' AS distance
FROM users
WHERE first_name % 'henr';

-- first_name | similarity | distance
-- -+-----+-----
-- Henrietta  | 0.36363637 | 0.6363636
-- Henriette  | 0.36363637 | 0.6363636
-- Henry      | 0.5714286  | 0.4285714
```

If you hadn't guessed it, the query above is also using the new GIN index you created.

Next you'll change things up a bit.

In the previous query a fragment of a word was matched exactly. What about similar words that don't match exactly? For example, a search query from a user may have a typo. For typo errors where the letters are ordered incorrectly, the search would produce no results which is not a good user experience. We all make typos!

As an example, searching “Henrieta” with one `t` instead of two will not produce any results.

10. <https://www.postgresql.org/docs/9.1/pgtrgm.html>

You can do better! Using the GIN index and `gin_trgm_ops` class you’ve already created covering `first_name`, it’s possible to search and filter using the similarity information you just learned about and find matches even for typos.

Try running the following statement in psql. Note that you’re now using the `SIMILARITY()` function as your filter criteria.

```
sql/query_filter_similarity.sql
SELECT DISTINCT
  first_name
FROM
  users
WHERE
  similarity(first_name, 'henrieta') > 0.4;

-- first_name
-- -----
-- Henrietta
-- Henriette
```

Although this query is much slower, it does find names that are similar. This can be improved further using Full Text Search in Rails with PostgreSQL, documented in Rails Guides.¹¹ By storing a list of all possible words, matches can be made using different operators to match queries against word content that is similar.

That covers it for GIN indexes. Next you’ll continue with B-Tree Indexes but see how to use a Partial Index.

Using Partial Indexes

Partial indexes are B-tree indexes that include an expression in the index definition that limits the rows covered. Reducing the rows in the index means the index will be smaller which consumes less space, but also reduces index maintenance overhead at write time. In other words, as data is being written in, if it doesn’t fit into the expression for a partial index, it won’t be included in the index.

A partial index is a specialized index because it’s optimized for certain values within a column, at the expense of not being useful for others.

Here’s an example to help clarify this. Imagine that the Rideshare application uses “soft deletes” when rows are deleted in the app. As a refresher, “Soft Deletes” means rows aren’t actually deleted but are instead updated to make them hidden from the application.

11. https://guides.rubyonrails.org/active_record_postgresql.html#full-text-search

To make the rows hidden, tables get a `deleted_at` timestamp column that's `NULL` by default and then is set when a row is deleted. Any rows with a `deleted_at` value set are hidden.

Imagine that Users can deactivate their accounts and this action populates the `deleted_at` column value. The Marketing department wants to find those users and offer them incentives to come back to the platform.

Create the index again like you did earlier, except this time exclude the deleted users from the index.

Rideshare does not currently have Soft Deletes enabled, so you'll need to add the `deleted_at` column.

Run the following statements in `psql` for Rideshare to add the column, and then add a Partial Index. After that you'll delete about 10 users, and then analyze the table.

```
sql/index_users_deleted_email_partial.sql
ALTER TABLE users ADD COLUMN deleted_at timestampz;

-- Index on deleted users
CREATE INDEX IF NOT EXISTS index_users_deleted_email_partial
ON users
USING btree (email)
WHERE deleted_at IS NOT NULL;

-- delete some users
UPDATE users
SET deleted_at = NOW()
WHERE id <= 10;

ANALYZE users;
```

Now try running this query:

```
SELECT email FROM users
WHERE deleted_at IS NOT NULL;
```

Prepend `EXPLAIN ANALYZE` to the query and confirm the Index is being used.

An alternative would be to use a Multicolumn index. Add an index covering the `deleted_at` column to filter down to only deleted rows, and then include the email column as the second column in the definition, so that the email can be fetched from the index without needing a Heap Scan.

Run this in `psql`:

```
CREATE INDEX IF NOT EXISTS users_deleted_email
ON users USING btree(deleted_at, email);
```

Now run `\d users` to confirm you have indexes “`index_users_deleted_email_partial`” that you created earlier, and “`users_deleted_email`” you created just now.

Run the query from earlier again, and confirm the Multicolumn index is used and an Index Only Scan is performed.

Review the trade-offs between these two approaches. Both the Partial index and Multicolumn index made the query much faster. They are both special purpose indexes for certain queries. One trade-off is that they consume different amounts of space and have different maintenance overhead.

Consider the size of the Multicolumn index. Run these `SELECT` statements from `psql` to compare the sizes:

```
SELECT
pg_size_pretty(pg_total_relation_size('users_deleted_email'));
pg_size_pretty
-----
1360 kB
```

Now check the size of the partial index:

```
SELECT
pg_size_pretty(pg_total_relation_size('index_users_deleted_email_partial'));
pg_size_pretty
-----
16 kB
```

The partial index is much smaller at 16 kB, but it may be used by fewer queries. You will have to test both of these in your database, carefully analyzing the query performance and the space consumption for each index type.

You’ve now seen how Partial Indexes can be used as specialized indexes for specific queries, with a small file size.

Next you’ll work with a different Index type, the BRIN index, to see how that Index can be used for large result sets while trying to minimize space consumption.

Using BRIN Indexes

You’ll use BRIN indexes to help improve query performance on a large data set. BRIN stands for Block Range Index.

What is a block range index entry?

A block range index entry points to a page (the atomic unit of how PostgreSQL stores data) and stores two values: the page's minimum value and the maximum value of the item to be indexed.¹²

BRIN indexes can be a good fit for analytical queries or queries that use time ranges where the data has a strong correlation between the physical ordering of the rows and how the data is queried.

Since the BRIN index stores page positions and since all rows are stored in pages, the most benefit from a BRIN index for query performance is when entire pages can be skipped.

A BRIN index only stores page values so it consumes little space. Since BRIN indexes are small, they may be worth having in addition to B-tree indexes to help for certain queries.

You saw what a block range index entry is, but what about a block range?

A block range is a group of pages that are physically adjacent in the table

In the post *When does BRIN win?*,¹³ the author shows examples of queries where the data is highly correlated, and when it has low correlation. The query uses an `AVG()` aggregate across a large data set and compares the performance with a BRIN index and a B-tree index including query times and index sizes.

In *Large Data Performance With Minimal Storage*,¹⁴ the author again compares `AVG()` performance on large sets of data using an B-Tree Index and a BRIN index. The BRIN index not only outperforms the B-Tree Index but is massively smaller.

“for BRIN to be effective, you need a table where the physical layout and the ordering of the column of interest are strongly correlated.”

To adapt this example to Rideshare, imagine a Trip Positions model that captures GPS points from Drivers and Riders on trips. This is a high volume insert-only table where rows are not updated or deleted but might be queried.

For this example, you'll insert Trip Position records with a generated series ticking every one second over several months. This ticking second value will be used as the value for the `created_at` column, and will move chronologically through time from the beginning of the range to the end of the range.

12. <https://www.crunchydata.com/blog/postgresql-brin-indexes-big-data-performance-with-minimal-storage>

13. <https://www.crunchydata.com/blog/postgres-indexing-when-does-brin-win>

14. <https://medium.com/geekculture/postgres-brin-index-large-data-performance-with-minimal-storage-4db6b9f64ca4>

This creates a lot of data, over 5 million rows to cover that time period in seconds. The rows are considered to have a “high correlation” because the values for “created_at” are clustered together with how they’re stored on disk and they’re similar in time. Low correlation would be values that are similar in time but scattered all over.

This high correlation meets the guideline above for a BRIN index to be effective.

From psql, run this SQL statement:

```
sql/trip_positions_populate_correlation.sql
INSERT INTO trip_positions (
    position, trip_id, created_at, updated_at)
SELECT POINT(' (37.769233' ||
    FLOOR(RANDOM() * 10 + 1)::TEXT ||
    ', -122.3890705)'),
(SELECT MIN(id) FROM trips),
seq,
seq
FROM generate_series(
    '2023-08-01 0:00'::timestampz,
    '2023-10-01 0:00'::timestampz,
    '1 second'::interval)
AS t(seq);
```

With that data created, you’re ready to query it. Run the following query in psql to get counts by day, grouping hundreds of thousands of records by day from the first 5 days of the month.

```
sql/trip_positions_by_day.sql
SELECT
    date_part('day', created_at) AS date,
    COUNT(*)
FROM trip_positions
WHERE created_at >= '2023-08-01' AND created_at < '2023-08-06'
GROUP BY 1
ORDER BY 1;
```

Prepend EXPLAIN ANALYZE and confirm that the query uses a Sequential Scan of the trip_positions table.

The WHERE clause above can use an index on the created_at column. To improve the speed of WHERE clause filtering, use a B-Tree or BRIN index. As you saw in the earlier posts, the B-Tree and BRIN indexes may offer comparable performance but the BRIN index will be much smaller.

To create the BRIN index, use the USING brin clause. Run this statement from psql to add the index and then run ANALYZE trip_positions:

```
sql/trip_positions_create_index_btree.sql
```

```
CREATE INDEX trip_positions_created_at_idx
ON trip_positions (created_at);

ANALYZE trip_positions;
```

With the B-Tree Index in place, the query runs much faster. The query execution plan shows an Index Only Scan using the “trip_positions_created_at_idx” index. Check the size of the index.

Running `SELECT pg_size_pretty(pg_relation_size('trip_positions_created_at_idx'));` in `psql` shows a size over 200 MB. Drop the Index and try creating a BRIN index instead.

```
sql/trip_positions_create_index_brin.sql
```

```
CREATE INDEX trip_positions_created_at_idx_brin
ON trip_positions USING brin (created_at);

ANALYZE trip_positions;
```

Run the same query again. You should see a Bitmap Index Scan using the “trip_positions_created_at_idx_brin” and very good query performance comparable to the B-Tree Index. Check the size of the BRIN index. The size is just around 40 kB. Much smaller!

You’ve seen how to use Indexes for filtering operations. How else can they be used?

Using Indexes In Less Common Ways

Besides speeding up queries, Indexes can be used for sort operations where `ORDER BY` clauses appear in a query. Indexes store sorted data as opposed to how the data might be ordered in pages.

B-tree indexes are sorted in ascending order by default.¹⁵

In this section you’ll put this concept into practice. For Rideshare the trips table has a `completed_at` column that’s set when a trip is completed. The trip’s `completed_at` is `NULL` when a trip is in progress.

Run the following query from `psql` to order the Trips by their completion time. If you’ve run the data generators you should have at least 1000 trip records.

```
EXPLAIN (ANALYZE) SELECT * FROM trips
WHERE completed_at IS NULL
ORDER BY completed_at;
```

The query uses a Sequential Scan of the trips table.

15. <https://devcenter.heroku.com/articles/postgresql-indexes#b-trees-and-sorting>

Taking inspiration from the Heroku “B-trees and Sorting” post from earlier, create an index on `trips.completed_at` that will be used by the `ORDER BY` clause from the query above. This Index orders the `NULL` `completed_at` records at the end. Run this statement in `psql` to create the index:

```
CREATE INDEX trips_completed_at_index
ON trips(completed_at DESC NULLS LAST);

ANALYZE trips;
```

Run the query again. The query is now much faster and the execution plan lists the index “`trips_completed_at_index`”. An Index Scan was performed for the Sort Operation with the Sort Key: `completed_at`.

You’ve now seen a less common way to use an Index. Indexes usage might be more common, including to cover multiple columns as a Covering Index. How does that work? Read on to learn more.

Using Covering Indexes

When the fields being requested match exactly with the fields covered by an index, this type of Index is called a “Covering Index.”

From PostgreSQL version 11 the `INCLUDE`¹⁶ keyword can be used for Covering Indexes to enable Index Only Scans. The `INCLUDE` keyword is a little different from a Multicolumn index but they are two ways to create a Covering Index.

With the `INCLUDE` keyword, columns can be added to a special payload area. Payload columns are not filtered on in a `WHERE` clause but are available if chosen in the `SELECT` part of a query.

In Active Record it’s less common to explicitly select columns. Consider places in your codebase that don’t select specific columns. Look at the minimum set of columns needed, and what columns are already Indexed. You may want to add missing Covering Indexes for performance sensitive areas that select a few columns.

When columns match a Covering Index, PostgreSQL can perform the very efficient Index Only Scan type and get all of the data needed from the Index.

It’s time to create examples of these scenarios to better understand them.

Create a Multicolumn index on `first_name` and `last_name` and then analyze the table, by running these statements from `psql`:

```
CREATE INDEX users_fname_lname_multi_idx ON users (first_name, last_name);
```

16. <https://www.postgresql.org/docs/11/indexes-index-only-scans.html>

```
ANALYZE users;
```

Querying only for the first_name “Elroy” uses the index. Run the following query in psql.

```
EXPLAIN (ANALYZE) SELECT first_name, last_name
FROM users WHERE first_name = 'Elroy';
```

This query uses an Index Only Scan using the users_fname_lname_idx Multicolumn Index.

An alternative to the Multicolumn index would be to use the INCLUDE keyword and set up a payload column. Drop the Multicolumn Index and create the following index from psql. Notice how this is a single column index on the first_name, but the INCLUDE keyword adds the last_name as a payload column.

```
CREATE INDEX users_fname_include_lname_idx
ON users (first_name)
INCLUDE (last_name);
ANALYZE users;
```

With this index, an equivalent Index Only Scan is used with index “users_fname_include_lname_idx”.

If you checked the sizes of these two Indexes you’d see the sizes are about the same. If the sizes are the same, what’s the benefit of one approach over the other? When enforcing uniqueness, the newer Covering Index style with the INCLUDE keyword has some benefits over the Multicolumn Covering index style.

When uniqueness is defined on the first_name column without the INCLUDE payload column, you’d need to create two indexes for the users table. One index would be a UNIQUE constraint Index and the other would be a Multicolumn Covering Index.

With the INCLUDE style, a single Index be used both for adding a UNIQUE constraint and to act as a Covering Index. Try creating the Index INCLUDE style one from earlier and make it a Unique Index.

Now you’ve learned about a lot of the types of Indexes available in PostgreSQL and how to use them for certain types of queries. You saw earlier how Indexes are useful to help speed up queries, but also consume space and thus shouldn’t be created unless they’re needed.

Part of operating an efficient database is identifying Unused indexes and removing them. Managing unused indexes is part of the overall database

maintenance operations that you'll learn how to perform. It's time to dive into common database maintenance challenges and see how to solve them.

High Impact Database Maintenance

You’ve seen how row modifications are made up of tuples, how there can be live and dead tuples, and how this is part of the Multiversion Concurrency Control (MVCC¹) system.

With this knowledge, you’re ready to learn about database maintenance in greater depth and see how automatic maintenance is performed and where you can help out by manually conducting maintenance.

For databases with hundreds of millions of rows and a high volume of queries you’ll need to tune your system and possibly conduct manual maintenance operations.

You’ll learn about the “VAR” aspects of database maintenance, which are Vacuum, Analyze, and Reindex.

Maintenance Terminology



- Visibility Map — Tracks which row versions (tuples) are visible to transactions
- Vacuum — Mark space as reusable, reclaim space (Vacuum Full), update Visibility Map
- Analyze — Update statistics about table data
- Reindex — Rebuild an index

First you’ll get an introduction or refresher on Autovacuum.

1. <https://www.postgresql.org/docs/current/mvcc.html>

Basics of Autovacuum

Automatic Vacuum, or *Autovacuum*, is a background process that runs by default when PostgreSQL starts. Autovacuum can be tuned using a variety of parameters. On modern production servers the default Autovacuum parameter values are considered to be very conservative. For this reason, Autovacuum tuning guides often recommend changing the values to make Autovacuum run more frequently.

If you're completely new to Autovacuum, it can be thought of as the “garbage collection process”² for PostgreSQL. In a roughly similar way to how garbage collection runs periodically in garbage-collected programming languages when objects are deallocated, Autovacuum runs periodically for dead row versions in a database that are no longer needed for any transactions.

Visible Rows are made up of row versions behind the scenes. Row versions can be live, reflecting what's currently visible to all live and running transactions, or dead. When a row is updated or deleted in PostgreSQL it's not actually updated or removed physically, but instead a new row version called a tuple reflects the changes from the update or delete.

Run the following query from psql. This query shows the ctid column, which is a hidden column. The two values displayed are the page number and tuple number.³

```
SELECT ctid,id FROM users WHERE id = 1;
```

Update the same User by running `UPDATE users SET first_name = 'Jane' WHERE id = 1;`

Now run the same query showing the ctid from above. The tuple version will change. The page number may be the same unless a lot of updates have happened, and in that case PostgreSQL may place the row version in a new page.

When a former row version is no longer referenced by any live transaction, it becomes dead. At that point the dead row version may be removed. Until it is removed, dead row versions are commonly referred to as *bloat*.

The way Autovacuum works in depth and the dozens of tunable parameters are beyond our scope. The basic way it works and a couple of tuning parameters will help you get started.

2. <https://www.postgresql.org/docs/current/routine-vacuuming.html>

3. <https://nidhig631.medium.com/ctid-field-in-postgresql-d26977de7b58>

Tuning Autovacuum Parameters

Autovacuum runs Vacuum workers per table when threshold values are met. These threshold values are configurable and there are a lot of them for different purposes.

Generally speaking, you may tune Autovacuum so that it runs jobs more frequently, for longer periods of time, and more of them in parallel. The tradeoff is that by running more workers for longer, you'll use more server system resources. These are the same server resources available to your application query workload, so you'll need to carefully make adjustments and monitor the impact.

As your database gets busier with higher rates of UPDATE and DELETE operations, dead row versions accumulate faster. Autovacuum may fall behind and not clean up dead tuples quickly enough before going to sleep again.

When Autovacuum is not keeping up you'll see a symptom of long VACUUM durations where it might run for hours or even days. VACUUM run times are logged in the `postgresql.log`.

The threshold to trigger an Autovacuum worker for a table is controlled by the parameter `autovacuum_vacuum_threshold`. This can be adjusted by tuning the `autovacuum_scale_factor`.

The default value for the scale factor is 0.2. This means that when 20% of the table size is made up of dead tuples VACUUM is triggered.

Many resources recommend reducing `autovacuum_scale_factor` so that VACUUM is triggered when there are fewer dead tuples. A value of 0.01 (1%) or even less will cause VACUUM to run much more frequently, which may be necessary based on the rate that dead tuples accumulate. As you make these changes, first make the change to specific tables that have a high volume of Updates and Deletes before making the change globally for all tables.

To get more familiar with this, you'll begin to tune Autovacuum for the Rideshare database.

The parameter `autovacuum_vacuum_cost_limit`⁴ specifies a “cost” of work that Autovacuum completes before it stops running. Raising this integer value to specify a higher cost causes Autovacuum to run longer, completing more work.

4. https://postgresqlco.nf/doc/en/param/autovacuum_vacuum_cost_limit/

In PostgreSQL 15 the default value is -1 and a separate parameter `vacuum_cost_limit` is used instead. The `vacuum_cost_limit` default value is 200, and this parameter is in effect when `VACUUM` is run manually from `psql`. Try running `VACUUM VERBOSE` trips; from `psql`. The `VERBOSE` option is optional but shows more information about what's happening.

Override the default by setting `autovacuum_vacuum_cost_limit` to a higher value like 2000. With this value, Autovacuum vacuum jobs will complete ten times the work before stopping. This is a good change to try if your Autovacuum Vacuum jobs are not cleaning up dead tuples at a fast enough rate. This might be worth checking mainly for tables with a very high volume of Updates and Deletes. This parameter value can be changed without restarting PostgreSQL so you can adjust it up and down to observe the effect.

Change the value globally in your PostgreSQL config and reload it using `pg_ctl reload` from your terminal.

```
sh/postgresql_conf_parameter_change.sh
pg_ctl reload -D /path/to/postgresql.conf
server signaled
LOG:  received SIGHUP, reloading configuration files
LOG:  parameter "autovacuum_vacuum_cost_limit" changed to "2000"
```

Although it can be changed globally, a better method to start might be to change it only for tables with heavy Updates and Deletes. To do that, as an example run `ALTER TABLE vehicles SET (autovacuum_vacuum_cost_limit = 2000);` from `psql` for the Rideshare database, to adjust the value for the vehicles table.

Another parameter to tune for PostgreSQL versions earlier than 15 is the `autovacuum_vacuum_cost_delay`. This parameter controls the pause duration. A lower value means Autovacuum is paused for shorter periods of time. In the table below the value is reduced from 20ms to 2ms. In PostgreSQL 15 the default value for this parameter became 2ms, so this parameter change isn't necessary there.

Parameter Name	Original Value	Tuned Value
<code>autovacuum_vacuum_cost_limit</code>	200	2000
<code>autovacuum_vacuum_cost_delay</code>	20ms	2ms

Table 3—Autovacuum Database Parameters

Tuning parameter values

These Autovacuum parameter changes were proposed by PostgreSQL Major Contributor David Rowley⁵ after analyzing a PostgreSQL 10 database with

5. <https://www.postgresql.org/community/contributors/>

significant bloat problems. The database had tables with very high amounts of Updates causing dead tuples to accumulate quickly. These changes helped Autovacuum run more effectively.

Another example from Sentry.io in the post *Transaction ID Wraparound in Postgres*,⁶ they describe lowering `autovacuum_vacuum_cost_delay` down to 0ms because Vacuum was not running frequently enough.

You've now seen how to adjust Autovacuum. Shift your focus to the Indexes on your tables to make sure they're maintained well.

Rebuilding Indexes Without Downtime

Periodically rebuilding an index in PostgreSQL ensures it only has pointers to live tuples. As a refresher, Indexes are secondary data stores that are owned by tables.

To rebuild Indexes in PostgreSQL you use the `REINDEX` statement. Dropping and creating the index again achieves the same outcome as it builds the index based on live tuples. But on a live system, this is unsafe because it means the index won't be available for active queries that use it.

On older versions of PostgreSQL, rebuilding indexes while the server was running was not possible. An index could be safely rebuilt (Reindexed) if the database was taken down without disrupting any queries.

Fortunately PostgreSQL has made it possible to rebuild indexes while the server is running. The index remains available to live queries that are using it.

If you're running PostgreSQL version 12 or older you won't yet have access to this capability. Upgrading to PostgreSQL 13 or newer is the best option because there are many improvements besides online index rebuilds. If upgrading is not an option, you've got alternatives. Use the extension and command line program `pg_repack`⁷ to perform online index rebuilds.

The post *Using 'pg_repack' to Rebuild Indexes*⁸ provides an overview of how to use `pg_repack` to replace Indexes online. Briefly, `pg_repack` can be used to rebuild an index. The technique it uses is to rebuild a replacement index and then swap in the replacement while dropping the old index in a transaction. `pg_repack` works well for that purpose but it is a separate client program from

6. <https://blog.sentry.io/2015/07/23/transaction-id-wraparound-in-postgres/>

7. https://reorg.github.io/pg_repack/

8. <https://andyatkinson.com/blog/2021/09/28/pg-repack>

PostgreSQL. What if there was a built-in way to rebuild indexes without disrupting any queries that are using the Indexes?

Fortunately from PostgreSQL 13 onward, you can use the `CONCURRENTLY` keyword with the `REINDEX` command to accomplish this, rebuilding Indexes online without requiring a third party tool like `pg_repack`.

Try putting this into action. From `psql`, run the following command to reindex the `index_trips_on_driver_id` index. The `Verbose` is used below and it's optional but shows more information about what's happening.

```
REINDEX (VERBOSE) INDEX CONCURRENTLY index_trips_on_driver_id;
```

Instead of Reindex with an Index, try specifying a Table to reindex all the indexes for the table. Run this statement from `psql`:

```
REINDEX (VERBOSE) TABLE CONCURRENTLY trips;
```

Running Manual Vacuums

PostgreSQL lays out data on disk in pages that are 8kb in size. The pages are not completely filled by default; space is left available to store new tuples from row modifications. Dead tuples take up space in those pages as well. When Autovacuum runs a `VACUUM` worker for each table in PostgreSQL, one of the jobs it performs is to reclaim the space used by dead tuples. Once completed, the space that was used by those dead tuples is marked as available again for new row modifications.

As a refresher, performing a manual `VACUUM` for a table like `users` is done by running this command from `psql`:

```
sql/vacuum_table.sql
VACUUM users;
```

The `ANALYZE` option can be added, and this updates table statistics for the table. The statistics include estimates that are used by the query planner as you saw earlier.

```
sql/vacuum_analyze_table.sql
VACUUM ANALYZE users;
```

New versions of PostgreSQL continue to add capabilities to Vacuum and Analyze. Version 11 made it possible to specify multiple tables at once. Version 12 added `SKIP_LOCKED` support which skips locked tables (See: *Postgres 12*

highlight - *SKIP_LOCKED* for *VACUUM* and *ANALYZE*⁹). Try running this statement from psql:

```
sql/vacuum_multiple_skip_locked.sql
```

```
VACUUM (SKIP_LOCKED) trip_requests, trips;
```

PostgreSQL keeps track of when Vacuum and Analyze are run both automatically and manually.

The following query shows a last_analyzed timestamp for the users table, which is when it had been last vacuumed manually. The timestamp last_auto_analyzed_at shows when the table was analyzed last by Autovacuum.

```
sql/last_analyzed_pg_stat_all_tables.sql
```

```
SELECT
    schemaname,
    relname,
    last_autoanalyze,
    last_analyze
FROM pg_stat_all_tables
WHERE relname = 'vehicles';
```

Try running it for the vehicles table. The value for last_analyze may be empty. Run ANALYZE vehicles; and then check the last_analyze value again.

Since PostgreSQL 13, Parallel Vacuum Workers can be configured (See: *Parallel Vacuum in Upcoming PostgreSQL 13*¹⁰).

The default value for MAX_PARALLEL_MAINTENANCE_WORKERS is 2. Try changing it to 4. Specify the PARALLEL option with a value of 4, and set the VERBOSE flag to print more information when the command runs. Up to 4 workers will be started.

```
sql/set_max_parallel_maintenance_workers.sql
```

```
SET MAX_PARALLEL_MAINTENANCE_WORKERS=4;

VACUUM (PARALLEL 4, VERBOSE) users;
```

Let's explore Bloat in greater depth.

Simulating Bloat and Understanding Impact

As you saw earlier, Bloat refers to the amount of dead tuples that are present in your system. Dead tuples consume space that could be made available for live tuples. When bloat is very high it can contribute to worse performance.

9. <https://paquier.xyz/postgresql-2/postgres-12-vacuum-skip-locked/>

10. <https://www.highgo.ca/2020/02/28/parallel-vacuum-in-upcoming-postgresql-13/>

Very high bloat is not normally observed in local development databases. Dead row versions might accumulate in a high volume production system with a very high amount of Updates and Deletes and with Autovacuum not tuned and using default conservative values. This problem has lessened in newer versions of PostgreSQL and may continue to lessen in versions in the future.

Bloat management is nonetheless a common challenge, so you'll work with it a bit more in a hands on fashion.

To better understand bloat in your local development PostgreSQL server, you'll disable Autovacuum and create a very high amount of updates and deletes. By measuring the estimated bloat percentage before and after these updates, you can start to see how bloat accumulates.

To get started, collect a bloat estimate of the users table. Use the `pgsql-bloat-estimation/table/table_bloat.sql`¹¹ script from GitHub by running the statement from `psql`. Add the line `WHERE schemaname = 'public' AND tblname = 'users'` just before the `ORDER BY`, to limit the rows to the `public.users` table. The `bloat_pct` shows the current estimated bloat percentage. Take note of the current value.

Next, simulate bloat by creating a lot of Updates to the table. Run the following statement from `psql` to create 10 million updates to the users table with unique values for each update.

```
sql/bloat_simulate_user_updates.sql
UPDATE users
SET first_name =
  CASE (seq % 2)
    WHEN 0 THEN 'Bill' || FLOOR(RANDOM() * 10)
    ELSE 'Jane' || FLOOR(RANDOM() * 10)
  END
FROM GENERATE_SERIES(1,1000000) seq
WHERE id = seq;
```

This statement generates one update per row, matching the Primary Key `id` column to a sequence value. If you've got around 20 thousand rows, you'll see around 20 thousand updates. Each row gets a unique string that's either Bill or Jane with a number added on. Run the statement a few times to generate more updates.

Periodically check the `bloat_pct` column by running the query from earlier. Run some more updates. You'll see the `bloat_pct` growing. With these heavy amounts of updates, you're causing bloat in the table and in the Indexes.

11. <https://github.com/loquix/pgsql-bloat-estimation>

To reclaim the space in the table, you'll need to run `VACUUM FULL` users; from `psql`. Running `VACUUM FULL` on a live system is not recommended because it will lock the table while it runs.

Indexes for the users table will also become bloated and should be rebuilt. This keeps them optimized for queries.

As you saw earlier, you may run `REINDEX` for a single Index, or for all Indexes on a table. Run this statement from `psql` to rebuild one index:

```
sql/reindex_concurrently.sql
REINDEX INDEX CONCURRENTLY index_users_on_email;
```

You've now seen how to use Vacuum, Analyze, and Reindex to help with your database maintenance. What other types of Index maintenance are there?

Removing Unused Indexes

The PostgreSQL Wiki has a dedicated page for *Index Maintenance*¹² worth reviewing.

Indexes are added to tables to be used by queries. PostgreSQL tracks their usage when a query execution plan performs an Index Scan.

To make sure Index Scans are tracked, confirm that the following values are set to on (they are on by default) by running these statements in `psql`.

```
SHOW track_activities;
SHOW track_counts;
```

Regularly search for indexes that aren't used by queries. These are called Unused Indexes and they should be removed. Indexes can consume a lot of space and they also add some latency to write operations as they're maintained.

- Unused Indexes may prevent Heap Only Tuple (HOT) Updates
- Unused Indexes unnecessarily increase the `VACUUM` run time for a table
- Unused Indexes unnecessarily increase query planning time
- Unused Indexes unnecessarily slow down backup and restore operations

Indexes can grow very large, even larger in size than the tables themselves. This can be very surprising at first. These very large Unused Indexes weigh down your system needlessly.

To remove an index safely on a live system, use the `CONCURRENTLY` keyword as you've done many times in earlier examples.

12. https://wiki.postgresql.org/wiki/Index_Maintenance

Before dropping Unused Indexes, spend some time to double check that they aren't used.

- Run the unused indexes query, identifying indexes with no Index Scans
- As a rollback plan precaution, copy the CREATE INDEX definition prior to removal so that it's available to recreate if needed.
- Perform the index removal using the CONCURRENTLY option

If you're removing an Index via an Active Record Migration, perform the index removal conditionally in case the index doesn't exist for all databases where the migration runs. This can be the case if you have many deployments of your application each with their own database, and the database definition has drifted.

To collect the Index definition as a CREATE INDEX statement, run the following query from psql with an example Rideshare table index.

```
sql/index_create_statements.sql
SELECT indexdef FROM pg_indexes
WHERE indexname = 'index_users_on_email';
```

An Active Record Migration that conditionally removes the index might look like this:

```
ruby/remove_index_concurrently.rb
class RemoveIndexUsersEmail < ActiveRecord::Migration[7.1]
  disable_ddl_transaction!

  def change
    if index_exists?(:users, :email)
      remove_index(:users, :email, algorithm: :concurrently)
    end
  end
end
```

With Active Record, if the index cannot be found using the table and field name, the name option can be used to explicitly name the index.

Besides Unused Indexes, another form of Index Maintenance is to prune Duplicate or Overlapping Indexes. What do those look like?

Removing Duplicate And Overlapping Indexes

Indexes cover one or more columns, and PostgreSQL does not prevent creating multiple indexes covering the same columns as long as the indexes have unique names. Try this out in psql by running these statements:

```
CREATE INDEX index_users_on_email2 ON users (email);
CREATE INDEX index_users_on_email3 ON users (email);
```


If you encounter duplicate indexes, one of them can be removed.

Watch out for Index Duplicates

PostgreSQL Does Not Prevent Duplicate Indexes



- Duplicate Indexes can be created to cover the same fields as long as the indexes have different names.
 - Remove duplicate Indexes
-

Exact duplicates are straightforward and not common. What is less straightforward and more common is overlapping or redundant Indexes.

In this scenario two or more indexes cover the same column and only one of them is necessary. The redundant index may have a different definition, but never be used by a query.

For example, when a query filters on the `first_name` column of the `users` table, a single column index covering `first_name` could be used. A Multicolumn index added later covering `first_name` and `last_name` could be used for filtering on `first_name` as well. These two indexes overlap, both covering the `first_name` field. If the Multicolumn index was added for a query later, it's possible the single column index is no longer needed.

You'll want to see if there are Index Scans for each of these Indexes. To help find these, you can use the PgHero tool. PgHero detects and displays overlapping indexes. The source code has a method `duplicate_indexes` that will compare columns being covered by indexes for a table and check for overlapping definitions.

This information is presented in a section on the main screen called *Duplicate Indexes*. For example, PgHero will explain the following.

These indexes exist, but aren't needed. Remove them with a migration for faster writes.

```
users_first_name_idx (first_name) is covered by users_first_name_email_idx (first_name, email)
```

In the example above, the second Multicolumn index can be used by queries for the `first_name` column. PgHero is suggesting that the first single column index on `first_name` is unnecessary. An Active Record Migration is even generated to perform the removal.

Bloat accumulates in tables with high amounts of Updates and Deletes. What about tables that only receive Inserts?

Removing Indexes On Insert Only Tables

For tables where rows are only inserted and not queried (SELECT), updated, or deleted, these kinds of tables may not have a need for indexes.

For Rideshare, `trip_positions` is one example of a table where rows are only inserted. To confirm this, run this query from `psql`:

```
sql/check_table_dml_behavior.sql
-- Check number of inserts, updates, deletes
-- for `trip_positions` table
SELECT
    relname,
    n_tup_ins,
    n_tup_upd,
    n_tup_del
FROM pg_stat_user_tables
WHERE relname = 'trip_positions';
```

If you find Indexes on *Insert Only Tables* and they aren't enforcing a constraint, they may be unneeded and can be removed.

Indexes for PRIMARY KEY and UNIQUE constraints are needed.

For Insert Only tables, since they did not accumulate dead tuples from UPDATE and DELETE operations, in the past PostgreSQL did not VACUUM these tables. Because VACUUM didn't run, table statistics weren't being updated which then could cause incorrect query planning estimates.

That oversight was fixed in PostgreSQL 13 which now performs statistics collection from Autovacuum even for Insert Only Tables.¹³ This was mentioned in the *Row Estimates* episode of the Postgres.fm podcast.¹⁴

You've now seen how Automatic Vacuum jobs should run. What if you want to run manual vacuum jobs or schedule them; how might you do that?

Scheduling Jobs Using `pg_cron`

`pg_cron`¹⁵ is a third party extension that has wide support on cloud hosted PostgreSQL databases. You'll work with it in your local PostgreSQL server.

Note that this is the first time you're working with a different database. As a refresher, connect to PostgreSQL using "psql" from your Terminal and run `\l` to list all the databases. From `psql`, connect to `postgres` by running `\c postgres`.

13. <https://www.cybertec-postgresql.com/en/postgresql-autovacuum-insert-only-tables/>

14. <https://postgres.fm/episodes/row-estimates>

15. https://github.com/citusdata/pg_cron

With that in mind, you're ready to add `pg_cron`. Add `pg_cron` to the extensions list in `shared_preload_libraries` in your PostgreSQL config file. The extension version will need to be compatible with your version of PostgreSQL.

Configure `pg_cron` to use the `postgres` database in the config file. In `shared_preload_libraries`, add `cron.database_name = 'postgres'` to do that.

To schedule a job, the `cron.schedule` function is provided from the extension. This function schedules jobs only in the database where `pg_cron` is configured. At this writing, `pg_cron` can be configured in only one database. Instead of that function, use the function `cron.schedule_in_database` to work around that limitation. This function is more complex but allows scheduling of jobs in other databases besides where `pg_cron` runs. You'll want to use this other function so that you can schedule jobs for the Rideshare database.

For a very high `UPDATE` rate table, you've decided to schedule a periodic manual `VACUUM`. This way `VACUUM` will run more frequently compared with relying solely on `Autovacuum`. Note that you can adjust `Autovacuum` settings per table and that's a good place to start. For this example, assume that you've done that already and you'd still like to run a manual `Vacuum`.

To schedule `VACUUM` for trips in the `rideshare_development` database for 10:00 UTC, run this statement from `psql`:

```
SELECT cron.schedule_in_database(
    job_name:='rideshare trips vacuum',
    schedule:='0 10 * * *',
    command:='VACUUM ANALYZE trips',
    database:='rideshare_development',
    username:='postgres'
);
```

You've now scheduled a job from the `postgres` database where `pg_cron` is running, to run for a table in the `rideshare_development` database, from the `postgres` database.

To view the job that was just scheduled, run this query from `psql`:

```
SELECT * FROM cron.job ORDER BY jobid;
```

Since it's scheduled but has not yet run, there will be no results from this query.

To break down the parts, `cron` is the schema where the function is defined, `job` is the table, and this table is in the `postgres` database.

Besides creating jobs, jobs can be altered like tables. To alter a scheduled job, use the `cron.alter_job` function.

Now that you've created a scheduled job, you'll want to monitor it. Read on to find out how to do that.

Monitoring pg_cron Scheduled Jobs

To see past runs of scheduled jobs, run this query from psql:

```
SELECT * FROM cron.job_run_details;
```

Jobs can be altered. Alter the job schedule to make it run every minute. After running it every minute, there should now be past run details to view in cron.job_run_details.

Run the following query to find the jobid for the *Rideshare trips vacuum* named job scheduled earlier in this section.

```
SELECT jobid, jobname, schedule, command FROM cron.job ORDER BY jobid;
```

If jobid is 1, run the following statement to modify the schedule attribute for jobid 1, setting the cron expression to mean “every 1 minute.”

```
SELECT cron.alter_job(job_id:=1,schedule:='* * * * *');
```

Once this is set, you should see log entries like these:

```
2022-12-28 14:33:00.003 CST [91464] LOG: \
cron job 1 starting: VACUUM ANALYZE trips
2022-12-28 14:33:00.049 CST [91464] LOG: \
cron job 1 COMMAND completed: VACUUM
```

Run the query `SELECT * FROM cron.job_run_details` from psql again in the postgres database to confirm that the status of those runs shows as succeeded.

To verify this further, you should see that PostgreSQL has tracked that the trips table has been Vacuumed.

Check timestamps for the last manual ANALYZE and VACUUM on trips by running this query:

```
SELECT schemaname, relname, last_analyze, last_vacuum
FROM pg_stat_all_tables
WHERE relname = 'trips';
```

You've now configured a VACUUM ANALYZE job to run as a cron job on a schedule.

What if you were able to monitor pg_cron Scheduled Jobs via PgHero?

In a Scheduled Jobs PR,¹⁶ a new *Scheduled Jobs* page was added that adds visibility to `pg_cron` jobs. Scheduled Jobs that are scheduled and jobs that have run are displayed.

One final topic for maintenance is to learn about some helpful maintenance tooling.

Conducting Maintenance Tune-Ups

Performing database maintenance to keep operations optimal is part of your job as a database administrator.

The most important operations you've conducted are `VACUUM`, `ANALYZE`, and `REINDEX` (See: *Maintaining a PostgreSQL database health with ANALYZE, REINDEX, and VACUUM commands*¹⁷).

You also learned why and how to remove unused Indexes and duplicate or redundant indexes.

You saw the `active_record_doctor`¹⁸ gem and `database_consistency`¹⁹ gem; these can also be used to help with database maintenance. Database Consistency has various Checkers including a `RedundantIndexChecker` that helps you find redundant indexes to remove.

You also saw the `rails-pg-extras`²⁰ gem; it can be used for maintenance as well. This gem checks for Duplicate Indexes, Unused Indexes, and Bloat. The gem also looks for what it calls Null Indexes, which are indexes with a high proportion of NULL values. These may be good opportunities to replace a regular B-Tree index with a smaller Partial Index.

Performing maintenance regularly and monitoring the automatic maintenance will help you keep your database running smoothly.

Now you're ready to look at handling higher levels of concurrent activity.

16. <https://github.com/andyatkinson/pghero/pull/3>

17. <https://andreigrigoriev.com/blog/2016-04-01-analyze-reindex-vacuum-in-postgresql/>

18. https://github.com/gregnavis/active_record_doctor

19. https://github.com/djezzl/database_consistency

20. <https://github.com/pawurb/rails-pg-extras>

Handling Errors From Increased Concurrency

In this chapter you'll see some common challenges faced with higher levels of concurrent use of the database. More concurrency comes with higher query volume. Your database will begin to have many more client connections and this will make it more critical that queries run quickly. You will also likely to see more errors due to lock contention or queries that are canceled, including cancellations due to safeguards you add.

Managing database connections is an important part of managing your server resources. Database connections are a limited resources and as the query volume increases you may begin to use all of the available connections and even need more connections than what's available.

You'll see how to do that in this chapter. You'll add safeguards that help prevent bad queries from consuming too many resources. You'll see how to better use the limited set of database connections that are available.

Although you may not have these issues today, by developing awareness and practicing how to deal with them in advance, you'll be well positioned to solve them if they arise.

Concurrency Terminology



- Connection Pooler — Middle layer software between Rails and PostgreSQL, to more efficiently use database connections
 - Server Connections — Database connections
 - Client Connections — Active Record pool client connections
-

Monitoring Database Connections

Your Ruby on Rails application is a client application from the perspective of a PostgreSQL server instance. When you connect to it with `psql`, that's another client application.

SQL queries that are generated from Active Record code are sent to PostgreSQL using a database connection. Your application uses TCP over a network to communicate with the server. These are the basics, and a deep dive into networking out of our scope.

The total number of connections that can be served concurrently is limited based on the CPU and server resources of your PostgreSQL instance. Remember that PostgreSQL runs as a server process on a single server (computer) and it will use as much CPU, Memory, and Disk resources as it can based on how you've configured it.

In this section you'll take a high level look at how Active Record Ruby code is transformed into SQL statements and sent to the server.

1. You write Active Record Ruby code, chain methods together, there may be lazy evaluation, and eventually one or more SQL queries are produced as query text
2. The “driver” code ensures that the generated SQL query is PostgreSQL compatible
3. In Ruby on Rails the `pg` Ruby gem is a common adapter for connecting to PostgreSQL
4. Active Record has an application side Connection Pool, which is a pool of database connections. A connection is checked out from the pool in order to send a query to the server. The pool size expresses the maximum possible amount of connections that could be in use, although fewer could be in use at any given time.
5. Finally, the SQL queries are sent over the network using a connection. PostgreSQL computes a result, and in a normal scenario it's returned using the same connection. Other possibilities are that a statement does not complete successfully or is canceled.

Connections are opened on demand. The Active Record connection pool is a client and is responsible for closing the connection. If the connection is not closed it will be held open in an idle state indefinitely.

An idle connection can be in two states. It can be performing a transaction (a query happens inside an explicit or implicit transaction), or it can be idle and not performing a transaction.

Most of the time, the state of connections observed will be idle or active. In PostgreSQL you may see idle in transaction,¹ which means the transaction has been opened with BEGIN but is not completing work.

Take a look at the Active Record Connection Pool. In the example below the pool is configured with pool: 5, meaning up to 5 connections can be used.

```
default: &default
  adapter: postgresql
  pool: 5
```

How else can you see what's happening? Query pg_stat_activity to view connection status information. The results include unrelated PostgreSQL background processes, so add a filter so those aren't displayed.

Filter on the application_name. An example application name is /bin/rails, and backend_type could be client backend.

Run the following query in psql. You may want to run the Rideshare server by running bin/rails server in a Terminal window, and then simulate activity by running bin/rails simulate:app_activity in another terminal window. While the simulation is running, this query will display results:

```
sql/list_connections_and_queries.sql
SELECT
  pid,
  datname,
  username,
  application_name,
  client_hostname,
  client_port,
  backend_start,
  query_start,
  query,
  state
FROM pg_stat_activity
WHERE pid != PG_BACKEND_PID() -- exclude this query
AND datname = 'rideshare_development' -- specify app DB
AND state IS NOT NULL; -- exclude PG background processes
```

In your production systems, look for a connection state of idle. The query field has the SQL statement itself. Compare backend_start and query_start. In *Monitoring Stats*² the backend_start is set when the backend process was started, and query_start is when the query started.

1. <https://www.postgresql.org/docs/current/monitoring-ps.html>
 2. <https://www.postgresql.org/docs/current/monitoring-stats.html>

When the query is executing, the state is “active”. When the query is no longer executing, a `state_change` timestamp is recorded. For example, when a connection is no longer active and is now idle, `state_change` should be set at the point where that transition occurred.

An idle connection is available for re-use by the client once it is no longer performing a transaction.

If connections are idle, how can those be managed?

Managing Idle Connections

The `idle_timeout` timeout in the Active Record Connection Pool can be set to determine a maximum allowable time that idle connections can be checked out from the pool. When they are idle and checked out for longer than the max allowed value, those connections will be forcibly returned to the pool. By default the value is 300 seconds (5 minutes).

A similarly named PostgreSQL server parameter exists called `idle_session_timeout`³ which is new in Version 14. The parameter has this description:

Terminate any session that has been idle (that is, waiting for a client query), but not within an open transaction, for longer than the specified amount of time.

This parameter can complement the `idle_timeout` parameter in the Active Record Connection Pool. The server parameter provides an additional layer of protection within PostgreSQL from connections that are idle and not completing any work. The reason those are bad is because they lock up one of the limited database connections.

Idle connections that are open for a long time may be terminated. For example, for pid 87581 with a idle connection and a query that's not actively running, cancel it using a query like this one:

```
sql/pg_cancel_backend.sql
-- cancel pid 87581
SELECT pg_cancel_backend(87581);
```

In *Waiting for PostgreSQL 14 – Add `idle_session_timeout`*, experiments are conducted with `idle_session_timeout` to help demonstrate how it works.

A query is sent to the background and then the `postgresql.log` shows the terminated connection with error FATAL: terminating connection due to idle-session timeout after the maximum time was exceeded. In the second example, despite

3. https://postgresqlco.nf/doc/en/param/idle_session_timeout/

exceeding the maximum `idle_session_timeout` value, a transaction opened with an `BEGIN` but idle was not terminated.

Per *How to Monitor PostgreSQL Connections* from EnterpriseDB, clients are responsible for closing connections.

It is worth mentioning that if the connection is not explicitly closed by the application, it will remain available, thereby consuming resources — even when the client has disconnected.⁴

Setting Active Record Pool Size

The pool size should be set with care to not exceed the maximum number of connections PostgreSQL can handle. The `max_connections` value should reflect a safe maximum possible value. Active Record will use *up to* the pool size of connections. Connections will be held open and idle for up to 5 minutes by default.

The article *Concurrency and Database Connections in Ruby with ActiveRecord*⁵ from Heroku shows how to use an environment variable to control the value for pool to make it easier to change on the fly.

From *Finding and killing long running queries on PostgreSQL*,⁶ run the following query to list queries that have been running for more than 5 minutes. View the duration column to see how long they've been running. Run the following query from `psql`. There will need to be active queries to display any results.

```
sql/long_running_queries_duration.sql
SELECT
  pid,
  NOW() - pg_stat_activity.query_start AS duration,
  query,
  state
FROM pg_stat_activity
WHERE (
  NOW() - pg_stat_activity.query_start
) > INTERVAL '5 minutes';
```

Using the Active Record Connection Pool

Active Record provides an application-level Database Connection Pooling (See: *Database Pooling*⁷) capability.

4. <https://www.enterprisedb.com/postgres-tutorials/how-monitor-postgresql-connections>

5. <https://devcenter.heroku.com/articles/concurrency-and-database-connections>

6. <https://medium.com/little-programming-joys/finding-and-killing-long-running-queries-on-postgres-7c4f0449e86d>

7. <https://guides.rubyonrails.org/configuring.html#database-pooling>

Database connections can be *physical* or *server* connections; server connections are used by the client Ruby on Rails application with the Active Record Connection Pool when it opens connections.

When a proxy server is between Ruby on Rails and PostgreSQL, connections from the client application to the proxy server are called *client* connections. Client connections between Rails and the proxy are re-used. Multiple client connections may use a single physical connection. Server connections between the proxy and PostgreSQL can be used more efficiently by client applications. You haven't yet worked with a proxy server but you'll see more about that soon.

Active Record lazily allocates physical database connections, meaning they're opened as needed and then kept open in an idle state when a query completes.

As a brief refresher on the backend pids and PostgreSQL processes, for new database connections the master PostgreSQL process is forked. Process forking is fast, but does add latency to the client-server round trip time. To eliminate this latency, the application layer connection pooler will keep connections open but idling. This way the process forking latency is not incurred for every new query. This is kind of like leaving your car running when you go into a store for just a minute!

In the next section, you'll look at dedicated connection pooling software that enhances the capabilities beyond what's possible with the Active Record Connection. These capabilities help your application reach higher levels of concurrency beyond what would otherwise be possible.

Running Out Of Connections

Ruby on Rails application instances can be scaled horizontally by running more instances of the (stateless) application. Commonly, background job processing with Sidekiq⁸ is run with multiple processes as well, and these background workers also require database connections.

For puma or Sidekiq, more processes are added with multiple threads of execution, and each thread requires a database connection.

Client queries may overrun the maximum number of database connections. This is called *exhausting* the connections. When this happens, no new queries can run because no connections are available. The server will return an error to the client like FATAL: sorry, too many clients already.

8. <https://github.com/mperham/sidekiq>

The PostgreSQL parameter `max_connections`⁹ can be used to control the maximum number of connections in use. Changing this value requires restarting PostgreSQL.

A default value of 100 reflects a smaller PostgreSQL database server. A modern production PostgreSQL server might have a higher value like 500 or more.

The steps to simulate connection exhaustion are:

- Set the `max_connections` to a low value of 3
- Open 3 psql connections to the server, and run a long query
- Open a 4th connection, observing that the connection fails to open because all connections are in use

Run this statement in psql:

```
sql/show_max_connections.sql
-- default: 100
SHOW max_connections;
```

Now change it. Edit `postgresql.conf`, setting `max_connections` to 3. Edit the value of `superuser_reserved_connections` to be 1 less than the value of `max_connections`, which is a requirement from PostgreSQL for this exercise. Restart PostgreSQL:

```
sh/postgresql_conf_edit_max_connections.sh
# use `vim` or another text editor
vim /path/to/postgresql.conf

# set to low value
max_connections = 3
superuser_reserved_connections = 2

# restart PostgreSQL, required when changing `max_connections`
pg_ctl restart -D /path/to/data_directory/
```

Now that you've edited `max_connections` to be 3, when you attempt to run a query using a 4th connection you'll see an error. To simulate this run the following script from your Terminal by copying and pasting it into a file and then running it with "sh exhaust_database_connection.sh".

```
sh/exhaust_database_connections.sh
#!/bin/bash

# set max_connections to 3
query="SELECT pg_sleep(30)"

for run in {1..4}; do
  echo "running query ${query}. times: ${run}"
  psql |
```

9. https://postgresqlco.nf/doc/en/param/max_connections/

```
--dbname rideshare_development \
-c "$query" & # separate processes
done
```

If you've configured PostgreSQL as indicated above, the 4th query requires a database connection and requesting one produces an error because no connections are available. You'll see output like the following.

```
sh/exhaust_database_connections_output.sh
```

```
running query SELECT pg_sleep(30). times: 1
running query SELECT pg_sleep(30). times: 2
running query SELECT pg_sleep(30). times: 3
running query SELECT pg_sleep(30). times: 4
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432" failed \
FATAL: sorry, too many clients already
```

Running out of database connections is a severe problem.

- In an application with increased use, more database connections will be needed
- Your organization may not wish to scale up (vertically) the database server, which does add CPUs and makes it possible to serve more connections, but costs more money
- Your application may keep a large amount of idle connections open, using up connections
- Idle connections from a long running query use up connections

For a Ruby on Rails application, on the conservative side, queries should be finishing in less than 1 second. For an application in a scaling mode under heavy concurrency, queries should run in the tens of milliseconds.

Run these queries to explore the active connections:

```
sql/idle_transactions.sql
```

```
-- e.g.
-- count,state
-- 7,active
-- 3510,idle
SELECT count(*), state
FROM pg_stat_activity
GROUP BY 2;

---https://dba.stackexchange.com/a/39758
SELECT * FROM pg_stat_activity
WHERE (state = 'idle in transaction')
AND xact_start IS NOT NULL;
```

How many connections do you need? Conduct some capacity planning and resource utilization investigation.

- How many Ruby on Rails application instances are running? How many threads are configured per process? Each thread acquires a database connection to complete work.
- What is the connection pool size per process? Multiply the number of processes by the configured pool size.
- How many Sidekiq processes are running? How many threads are running per process? Sum the total number of threads to get the total concurrent number of transactions that are possible.

Complete this type of capacity planning analysis, totaling up web processes, background processes, and looking at their pool sizes, to gain a possible maximum theoretical number of concurrent connections the application would use.

Compare that figure with how you've configured `max_connections`. Under provision, leaving some connections available for administrative work like a `psql` session or other PostgreSQL CLI programs like `vacuumdb`. This also assumes there are not other client applications connected to the same PostgreSQL database. As a best practice, give each client application a unique database.

When you're running out of connections, you'll need more help. Fortunately there is additional software you can run for such a scenario.

Working With PgBouncer

When the concurrency of your application grows, the number of physical database connections used by the Active Record Connection Pool alone may not be enough.

PgBouncer, free open source software that you can run in between Ruby on Rails and PostgreSQL, allows you to scale to higher levels of concurrent database activity.

- Client connections may exceed `max_connections`
- Connections can be held open even if the database server is temporarily unreachable, delaying their processing but not refusing the connections
- The pooler re-uses connections more efficiently, achieving higher levels of concurrency on the same compute resources
- Connection Poolers may keep connections open while backends are modified, for example when modifying the server instance

In this section you'll install, configure, and begin using PgBouncer with Rideshare.

Earlier you learned that establishing a database connection has some costs, and adds some latency to the query. Some of the costs are:

- Negotiating SSL
- Authentication

Without reuse of a connection from a pool, these costs are paid for every new query. With a connection pooler, these costs are not paid for each new query or even each new connection. PgBouncer holds connections open to PostgreSQL so that clients like Active Record can reuse them.

On Mac OS, install PgBouncer using Homebrew.¹⁰ Run this command from your Terminal:

```
sh/pgbouncer_install.sh
brew install pgbouncer
```

After PgBouncer is installed, run the following commands to inspect the pgbouncer process and restart it.

```
sh/pgbouncer_commands.sh
brew services info pgbouncer
brew services restart pgbouncer
```

PgBouncer runs on port 6432 which is exactly 1000 higher than the default PostgreSQL port of 5432. Exciting trivia for your next PostgreSQL cocktail party!

Next, set up a PostgreSQL user that will act as the application user to connect with. Run these statements from psql:

```
sql/create_test_user.sql
DROP ROLE IF EXISTS app_user;
CREATE USER app_user WITH PASSWORD 'insecurepwd';
```

Configure the application user. Add the `admin_users = app_user` section below to make `app_user` an admin user. Edit the `.ini` configuration file by opening it in your text editor. Find the file path by running `brew info pgbouncer`. After changing the config file, run `pgbouncer -R /usr/local/etc/pgbouncer.ini` from your Terminal to restart PgBouncer.

```
sh/pgbouncer_ini_config.sh
[pgbouncer]
listen_port = 6432
listen_addr = localhost
auth_type = md5
auth_file = /usr/local/etc/userlist.txt
```

10. <https://brew.sh>

```
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = app_user
```

With PgBouncer configured, you're ready to connect it to the Rideshare database.

Try connecting to PostgreSQL via pgbouncer by running the following `psql` command with a connection string from your Terminal.

```
sh/pgbouncer_psql_connection_string.sh
# pgbouncer user, password, created earlier
# port=6432
psql "postgresql://app_user:insecurepwd@localhost:6432/rideshare_development"
```

PgBouncer acts transparently as a proxy and is compatible with `psql`. View some information about PgBouncer by running these commands:

```
sh/pgbouncer_connect_explore.sh
# Special administration "pgbouncer" database
psql -p 6432 -U app_user pgbouncer

# List commands available from pgbouncer prompt
SHOW help;
```

This brings up a command line interface like `psql` where you can run various commands. PgBouncer can be restarted using a rolling restart. Refer to the documentation for more information.

```
sh/restart_pgbouncer.sh
# Note that -R/--reboot is deprecated. Refer to documentation
# for a rolling restart.
pgbouncer --reboot /usr/local/etc/pgbouncer.ini --daemon
```

PgBouncer supports SQL formatted commands. Try running some of the following commands while connected to the special pgbouncer database.

```
sql/pgbouncer_commands.sql
SHOW help;
SHOW clients;
SHOW databases; -- Review 'pool_size', 'pool_mode', etc.
```

If you ran `show databases;`, one of the options you'll see is the Pool Mode. What's that all about?

Choosing A PgBouncer Pooling Mode

PgBouncer supports various *Pool Modes*. The Pool Mode specifies how a server connection may be reused by clients.

Pool Modes are listed below, ordered from the most aggressive level to the most conservative level in how connections are reused. A more aggressive Pool Mode will reuse connections earlier, potentially achieving higher levels of concurrency, at the cost of having to disable certain client side features.

Some features like *Prepared Statements* and *Advisory Locks* are not compatible with all pool modes. These are covered later.

The descriptions below are directly from PgBouncer.

Mode	Description
session	The default mode. The Server is released back to pool after client disconnects.
transaction	Server is released back to pool after transaction finishes.
statement	Server is released back to pool after query finishes. Transactions spanning multiple statements are disallowed in this mode.

Table 4—Pool Modes

Selecting a Pool Mode for maximum reuse

The *Transaction* Pooling Mode cannot be used with Prepared Statements, which are enabled by default in Active Record.

The statement Pool Mode is not commonly used. This was discussed in the *Connection Poolers*¹¹ episode of Postgres.fm.

The Transaction mode is most commonly used. To use the *Transaction* Pooling Mode, though, Prepared Statements must be disabled.

If disabling Prepared Statements is not a viable path, then the *Session* mode from PgBouncer is the next best option.

Now that you’ve installed and configured a connection pooler, and have an idea of how it can be used, shift your focus to errors related to database connection.

Identifying Connection Errors and Problems

In this section you’ll use psql to analyze connections. Connections may be viewed in PgHero¹² which you began to use earlier.

Some connection errors to avoid are:

- Exhausting connections

11. <https://postgres.fm/episodes/connection-poolers>

12. <https://github.com/ankane/pghero>

- Having too many idle connections
- *Transaction ID wraparound*
- Runaway queries, or zombie queries, that never return results to the client

Some of the safeguards you'll add are:

- Setting a `statement_timeout` to set a max allowed amount of time for a statement to run
- Setting a `lock_timeout` to prevent waiting too long to acquire a lock
- Setting `idle_in_transaction_session_timeout`¹³ to protect against transactions that are idle and not performing any work

You've already seen `statement_timeout` and `lock_timeout` in an earlier chapter.

Focus now on some of the idle timeouts. The following table presents these parameters and some recommended values in milliseconds.

Parameter Name	Default	Tuned Value	Restart
<code>idle_session_timeout (14+)</code>	0	30000	false
<code>idle_in_transaction_session_timeout</code>	0	3000	false
<code>statement_timeout</code>	0	10000	false
<code>lock_timeout</code>	0	3000	false

Table 5—Database Concurrency Parameters

Tuning parameter values

When considering `idle_session_timeout`¹⁴ and `idle_in_transaction_session_timeout`, start by adjusting the latter.

Recommended parameter values are from the *Tightly Coupled* post *GOTO Postgres Configuration for RDS and Self-managed Postgres*¹⁵ and *My GOTO Postgres Configuration for Web Services*.¹⁶

To collect the best values for your particular database, conduct experiments and gradually introduce changes in small increments. Develop a good understanding of how you'll observe the effect and make sure it provides the benefit you're looking for.

Note that Cloud providers will tune some parameters like memory allocation dynamically based on the server instance. Client parameters like timeouts may be left up to customers to enable and adjust.

13. <https://www.postgresql.org/docs/current/runtime-config-client.html>

14. https://postgresqlco.nf/doc/en/param/idle_session_timeout/

15. <https://tightlycoupled.io/goto-postgres-configuration-for-rds-and-self-managed-postgres/>

16. <https://tightlycoupled.io/my-goto-postgres-configuration-for-web-services/>

More Lock Monitoring With `pg_locks`

The `pg_locks`¹⁷ system view provides current information about held locks. In MVCC, all transactions use either a shared or exclusive lock of database resources like a table or rows of data.

You observed lock activity focused on table level locks and concurrent modifications to table structures. Now you'll focus on errors with concurrency from DML operations.

Besides DDL changes like `ALTER TABLE` that use an exclusive lock, Autovacuum uses a `SHARE UPDATE EXCLUSIVE` lock mode. This lock type means that it is both an exclusive lock and a shared lock. Access for reads is Shared, but is locked for updates.

When an `UPDATE` occurs to a row, the row is locked exclusively during the update. Concurrent reads of the row are possible.

`SELECT ... FOR UPDATE` adds additional protection when selecting rows being updated (See: *Selecting for Share and Update in PostgreSQL*¹⁸).

`ROW SHARE LOCK` locks are taken on those rows. Although rows could be selected and updated within a transaction, a concurrent transaction could modify the same rows being selected.

The solution is to open a transaction and select the rows, but add the `FOR UPDATE` keyword to indicate that rows will be exclusively locked for the duration of the `SELECT` statement.

Taking an exclusive lock on rows will block other row `SELECT` operations and modifications. Any concurrent readers or modifications will be in a waiting state for the `SELECT ... FOR UPDATE` transaction block to `COMMIT` or `ROLLBACK`.

The `NOWAIT` keyword can be added to the `SELECT ... FOR UPDATE` to prevent excessive waiting to acquire the lock on the rows to update.

`FOR UPDATE` is considered to create an explicit lock, similar to how the `LOCK` command may be used to explicitly lock a table.

Monitoring Row Locks

Set the `log_min_error_statement`¹⁹ to log queries to `postgres.log` that may time out due to being blocked on lock acquisition.

17. <https://www.postgresql.org/docs/current/view-pg-locks.html>

18. <https://shroyasha.io/selecting-for-share-and-update-in-postgresql.html>

19. <https://www.postgresql.org/docs/current/runtime-config-logging.html>

SKIP LOCKED limits the selection to rows that are not currently locked.

For a query running repeatedly, skipping locked rows would eventually select all rows, assuming that lock durations are small and the transactions holding them complete successfully.

SKIP LOCKED is a good concurrency control mechanism, when using a database as a queue of items to process.

As mentioned in *Selecting for Share and Update in PostgreSQL*,²⁰ rows on a referenced table by a Foreign Key column will also be locked when SELECT ... FOR UPDATE is used. This is necessary to maintain referential integrity.

Another type of weaker lock is SELECT ... FOR SHARE. This creates a shared lock on the rows being selected. A shared lock allows another concurrent reader to select the same row.

A shared lock still prevents Updates and Deletes of the row, but as a weaker lock type, allows for greater read concurrency.

How do you find lock conflicts?

Finding Lock Conflicts

In the Citus article *PostgreSQL rocks, except when it blocks: Understanding locks*,²¹ SQL operations are put into a table along with lock types and conflicts. When operations conflict with each other, a red X is placed on the intersection. When they do not conflict, a checkmark is placed in the intersecting table cell.

For example, a SELECT will never conflict with a SELECT, so that intersecting table cell gets a green checkmark.

A SELECT *will* conflict with an ALTER TABLE operation, because the ALTER TABLE requires a table level exclusive lock to run, which blocks concurrent reads. Reads are blocked until the ALTER TABLE lock expires which is when the table modification completes.

Each statement acquires a lock type, usually implicitly.

Next, look at conflicts from CREATE INDEX. Scan the CREATE INDEX column, identifying SQL statements that conflict. Any DML operation will require a lock type that conflicts with the lock type a CREATE INDEX takes. This is why creating

20. <https://shiroyasha.io/selecting-for-share-and-update-in-postgresql.html>

21. <https://www.citusdata.com/blog/2018/02/15/when-postgresql-blocks/>

indexes using the `CONCURRENTLY` keyword is recommended for any production database where DML operations are occurring concurrently.

How else can you analyze lock behavior?

Using PgBadger For Lock Analysis

You can again use the PgBadger²² log analysis tool you saw earlier, except this time using the reports it creates for lock analysis.

Using PgBadger is straightforward. Call the executable, passing a single argument which is the path to your `postgresql.log` file.

Run this command from your Terminal:

```
sh/pgbadger_log_analysis.sh
pgbadger /path/to/postgresql.log
```

The program crunches the data and produces an HTML output file that can be opened in your browser for analysis, named `out.html` by default. Run `man pgbadger` to review the large list of optional arguments.

The data presented by PgBadger is historical data from logged activity. Since it relies on log data, it cannot show you what is currently happening.

One of the features of PgBadger is Lock Analysis. Navigate to the top navigation “Locks” tab and explore the information there.

You may want to set a value for `log_lock_waits`²³ and observe the effect. PostgreSQL runs a deadlock checking routine after 1 second.

You may want to monitor for row level locking problems when using `SELECT ... FOR UPDATE`.

What about locking in Active Record?

Active Record Optimistic Locking

In a concurrency context, locks are acquired either optimistically or pessimistically. Pessimistic locks are what PostgreSQL uses, meaning the locks are taken upfront. PostgreSQL supports high levels of concurrent reads and modifications, even with a pessimistic locking strategy.

22. <https://github.com/darold/pgbadger>

23. https://postgresqlco.nf/doc/en/param/log_lock_waits/

When using PostgreSQL with Active Record, an optimistic locking technique can be used. This technique can be used when multiple transactions are updating the same row.

With optimistic locking, a lock is taken “late.” Each process with an optimistic lock can check whether the object being locked has already been modified. When the second modification has lost the “race” to modify the locked object, an `ActiveRecord::StaleObjectError` exception is raised.

To use this functionality, add a `lock_version` column to tables used by Active Record objects that you wish to lock. The column is automatically updated by Active Record when objects are instantiated.

The Engine Yard post *A Guide to Optimistic Locking*²⁴ shows how to handle concurrent modifications. When locking is required for the application, and the locking experience involves user interface level interaction, consider rescuing this error, and rendering the edit page with the latest version of the object instantiated. Add a helpful Rails *Flash Message* for the user, to describe that their edit was not allowed.

When locking is behind the scenes and not user facing, lock management at the row level may best be done using the PostgreSQL capabilities of `SELECT` with either `FOR UPDATE` or `FOR SHARE`. Use `FOR UPDATE` when the goal is to select some rows and prevent concurrent updates.

What about lighter weight locks?

Using Advisory Locks

PostgreSQL provides a weaker lock type called an *Advisory Lock*. Advisory Locks in Active Record use the Ruby standard library `Mutex`.

Ruby on Rails uses Advisory Locks. One example is when Active Record Migrations are run, an Advisory Lock is taken to prevent two threads of execution from applying the same migration. If this happened, an `ActiveRecord::ConcurrentMigrationError` error is raised.

Advisory Locks are useful, but come with other trade-offs. The Transaction Pooling Mode for PgBouncer cannot be used with Advisory Locks. To use Transaction Pooling Mode with PgBouncer, from Rails 6, Advisory Locks can be disabled²⁵ by setting `advisory_locks: false` in `config/database.yml`. As you saw earlier, you’d also need to disable Prepared Statements.

24. <https://www.engineyard.com/blog/a-guide-to-optimistic-locking/>

25. <https://blog.saeloun.com/2019/09/09/rails-6-disable-advisory-locks.html>

Wrapping Up

In this chapter you've started to see how to handle a common challenge in scaling up a PostgreSQL instance which is managing limited database connections. You saw how to identify the number of connections being used and how to scale them up even beyond what PostgreSQL is capable of itself by using a connection pooler.

The upcoming chapters will focus more on Optimizing and Scaling topics.

In the next chapter you'll consider the Ruby code design and Schema design for some common application features, from a scalability perspective, applying what you've learned so far in your evaluation.

Part IV

Optimize and Scale

Scalability of Common Features

Content to be supplied later.

Working With Bulk Data

Content to be supplied later.

Scaling With Replication And Sharding

Content to be supplied later.

Boosting Performance With Partitioning

Content to be supplied later.

Part V

Advanced Uses

Advanced Usage and What's Next

Content to be supplied later.

Installation Guides

Content to be supplied later.

APPENDIX 2

psql Client

Content to be supplied later.

Getting Help

Content to be supplied later.

Thank you!

We hope you enjoyed this book and that you're already thinking about what you want to learn next. To help make that decision easier, we're offering you this gift.

Head over to <https://pragprog.com> right now and use the coupon code BUYANOTHER2022 to save 30% on your next ebook. Void where prohibited or restricted. This offer does not apply to any edition of the *The Pragmatic Programmer* ebook.

And if you'd like to share your own expertise with the world, why not propose a writing idea to us? After all, many of our best authors started off as our readers, just like you. With a 50% royalty, world-class editorial services, and a name you trust, there's nothing to lose. Visit <https://pragprog.com/become-an-author/> today to learn more and to get started.

We thank you for your continued support, and we hope to hear from you again soon!

The Pragmatic Bookshelf



The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by professional developers for professional developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page

<https://pragprog.com/book/aapsql>

Source code from this book, errata, and other resources. Come give us feedback, too!

Keep Up-to-Date

<https://pragprog.com>

Join our announcement mailing list (low volume) or follow us on Twitter @pragprog for new titles, sales, coupons, hot tips, and more.

New and Noteworthy

<https://pragprog.com/news>

Check out the latest Pragmatic developments, new titles, and other offerings.

Buy the Book

If you liked this ebook, perhaps you'd like to have a paper copy of the book. Paperbacks are available from your local independent bookstore and wherever fine books are sold.

Contact Us

Online Orders: <https://pragprog.com/catalog>

Customer Service: support@pragprog.com

International Rights: translations@pragprog.com

Academic Use: academic@pragprog.com

Write for Us: <http://write-for-us.pragprog.com>

Or Call: +1 800-699-7764