

PostgreSQL

for

Jobseekers

Introduction to PostgreSQL administration for modern DBAs



Sonia Valeja

David Gonzalez



PostgreSQL for Jobseekers

*Introduction to PostgreSQL
administration for modern DBAs*

Sonia Valeja
David Gonzalez



www.bpbonline.com

Copyright © 2023 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55514-004

Dedicated to

*My supportive parents – Rupa & Hasanand and
my loving husband – Sunil*

— Sonia Valeja

My loved Sofi, the reason I do all this.

— David Gonzalez

About the Author

- **Sonia Valeja:** Sonia is an Indian DBA who started working in PostgreSQL in the year 2012. Since she has performed multiple migrations from Oracle to PostgreSQL, she learnt Oracle while working with PostgreSQL. She is a SME for beginner-level course on Open Source Database-PostgreSQL in one of her company. She has conducted multiple trainings on PostgreSQL for freshers as well as experienced professionals. Also, she has been an active participant at pgConf and has given lightning talk. She has worked extensively in Taxation as well as manufacturing based projects for Indian as well as African sub-continent.
- **David Gonzalez:** David is a Mexican DBA with 10+ years of hands-on experience with different RDBMS such as PostgreSQL, Oracle, and MySQL. He is good at PostgreSQL Administration and Development and has written multiple blogs on PostgreSQL. He has participated in various projects for banking, IT, and Software companies.

About the Reviewer

Amul Sul is a passionate developer with extensive professional experience in the domain of core database development.

He is an active contributor in PostgreSQL feature development as well as in EDB Postgres Advanced Server propriety feature development.

He is currently working in EnterpriseDB and is part of the database developer team. Prior to EDB, he worked at NTT Data. Amul holds a Master's degree from Mumbai University.

Acknowledgements

I want to express my deepest gratitude to my family and friends for their unwavering support and encouragement throughout this book's writing, especially my parents Rupa & Hasanand and my better half Sunil.

— *Sonia Valeja*

I want to say thank you from the bottom of my heart to my family, my wife Pam, and my beautiful daughter Sofi, for all the support and patience during this book process. They were always there cheering me up and pushing me to reach the goal.

— *David Gonzalez*

We are also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. It was a long journey of revising this book, with valuable participation and collaboration of reviewers, technical experts, and editors.

We would also like to acknowledge the valuable contributions of my colleagues and co-worker during many years working in the tech industry, who have taught me so much and provided valuable feedback on my work.

Finally, we would like to thank all the readers who have taken an interest in our book and for their support in making it a reality. Your encouragement has been invaluable.

Preface

Welcome to the world of PostgreSQL! Whether you are a novice exploring the realm of databases or a seasoned professional looking to expand your knowledge, this book is designed to be your comprehensive guide to everything PostgreSQL offers. We have carefully crafted this book to cover a wide range of topics, from the history and installation of PostgreSQL to its intricate internals, advanced tuning techniques, and securing your environment. We also delve into data replication, backup and restore operations, and the process of contributing to the vibrant PostgreSQL community.

In the first section of this book, we embark on a journey through time to explore the rich history of the PostgreSQL project. From its humble beginnings as a research project at the University of California, Berkeley, to becoming one of the most powerful and popular open-source databases, you will gain insight into the evolution and key milestones that have shaped PostgreSQL into what it is today. Understanding its roots will help you appreciate the philosophy and ethos behind this remarkable database system.

The subsequent chapters focus on the practical aspects of PostgreSQL, starting with the installation process on modern operating systems. We will guide you through each step, providing clear instructions and best practices to ensure a smooth setup. Additionally, we delve into alternative deployment options, such as Docker and Cloud, offering flexibility in how you harness the power of PostgreSQL in your own environment.

Once you have PostgreSQL up and running, we dive deep into its internals. Understanding how PostgreSQL processes, stores, and retrieves data is crucial for effective performance tuning and troubleshooting. We unravel the inner workings of the query optimizer, storage engine, and transaction management system, equipping you with the knowledge and tools to optimize your database for optimal performance. Whether dealing with a sluggish query or facing scalability challenges, this section will be your indispensable resource.

We also explore the essential backup and restore operations practices, enabling you to protect your data against unforeseen disasters. Later, we shed light on implementing data replication, which not only ensures high availability but also opens doors to advanced use cases like distributed architectures and load

balancing. Moreover, we delve into securing the access and integrity of your data, covering topics such as authentication and authorization.

Finally, this book wouldn't be complete without acknowledging the vibrant and collaborative PostgreSQL community. We will learn about some of the most used open-source projects, in the form of extensions and compatible tools that can enhance the native PostgreSQL capabilities. We share valuable insights on how you can contribute to this remarkable community through code contributions, documentation, or engaging in user forums. By becoming an active member, you not only enhance your knowledge and skills but also play a role in shaping the future of PostgreSQL.

Now, fasten your seatbelt and get ready for an exhilarating journey into the world of PostgreSQL. This book is your trusty companion, empowering you to master every facet of this powerful database system. Let's dive in and unlock the true potential of PostgreSQL together!

Chapter 1: Introduction to Opensource Database - PostgreSQL - introduces the open-source paradigm and covers the main general aspects of PostgreSQL like PostgreSQL history, what kind of applications work with PostgreSQL database and much more. Furthermore, the chapter also gives the reader an overview of the PostgreSQL, a brief history of PostgreSQL, PostgreSQL release cycle and the current impact of PostgreSQL in the market.

Chapter 2: Getting PostgreSQL to work - presents a detailed overview two ways of installing PostgreSQL - Source Code Installation and Binary Installation.

Chapter 3: Modern Options to get PostgreSQL - covers some other options to get PostgreSQL such as DBaaS/Cloud vendors and containers/Kubernetes.

Chapter 4: Global Objects in PostgreSQL - allows the reader to learn fundamental concepts of the global objects in a PostgreSQL cluster. These are handled per instance rather than per database. It describes Roles/Users/Groups, Tablespaces and Databases with examples.

Chapter 5: Architecture of PostgreSQL - gives special attention to the main components with respect to Memory Architecture, Background Processes and Physical Architecture.

Chapter 6: PostgreSQL Internals - explains Internals of the PostgreSQL from the perspective of ACID,MVCC, Transaction Isolation levels, Query Processing, and Vacuum.

Chapter 7: Backup and Restore in PostgreSQL - explains the importance of a backup strategy for a PostgreSQL solution, and teaches the different ways to get them. Then it shows how to restore these backups into a working service. Finally, it presents some of the most widely used tools for backup and restore such pgBackRest, WAL-G and so on

Chapter 8: Replicating Data - is dedicated to high availability architecture. It covers the main current replication techniques. It explains their differences, how to configure them, and their advantages and limitations.

Chapter 9: Security and Access Control - explains the importance of access control and security, and how PostgreSQL handles the authentication and authorization. The pg_hba and the GRANT/REVOKE are explored here.

Chapter 10: Most used Extensions/Tools - focus on some of the important extensions used which helps in extending the default behaviour of Postgresql in most frequent utilities. It cover the extensions like pg_stat_statements & pg_repack.

Chapter 11: Basic Database Objects - touches the concepts of database schemas, DDL command, DML Queries and DCL commands that are important from PostgreSQL Developer perspective. The main topics covered are Schema management, Data Definition Language command, Data Manipulation Language commands, control statements and constraints.

Chapter 12: Advance Database Objects - covers the advanced PL/PgSQL concepts like procedures, triggers and rules. It also covers details on custom data types in PostgreSQL.

Chapter 13: Performance Tuning - focuses on index creations along with best practices to be followed in configuration files. Once data size increases, there comes need to tune the database queries using Index and Explain Plan.

Chapter 14: Troubleshooting - gives insights on analyzing log files using database commands as well as OS commands. A DBA should also be aware on basics command of Operating System which are also covered from the DBA perspective.

Chapter 15: Contributing to PostgreSQL Community - discusses how to be part of PostgreSQL Community and build online presence.

Coloured Images

Please follow the link to download
the *Coloured Images* of the book:

<https://rebrand.ly/gcsuy63>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/PostgreSQL-for-Jobseekers>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



Table of Contents

1. Introduction to Opensource Database - PostgreSQL.....	1
Introduction.....	1
Structure	1
Objectives.....	2
Open-source introduction	2
<i>The origin of free software.....</i>	2
<i>The open source concept.....</i>	3
<i>The open source nowadays.....</i>	5
A brief history of PostgreSQL.....	6
<i>The POSTGRES project.....</i>	6
Postgres95	7
PostgreSQL	7
<i>PostgreSQL versions key features.....</i>	8
<i>PostgreSQL in stats on a single image</i>	10
PostgreSQL release cycle	10
<i>The current impact of PostgreSQL on the market.....</i>	12
Companies that use PostgreSQL.....	14
<i>Companies that help in enhancing PostgreSQL</i>	14
<i>Advantages of PostgreSQL</i>	16
<i>Distributed architecture with PostgreSQL.....</i>	17
<i>StatefulSet with PostgreSQL</i>	18
Conclusion	18
Bibliography.....	19
2. Getting PostgreSQL to work	21
Introduction.....	21
Structure	21
Objectives.....	22
Source code installation	22
<i>Short version of source code installation</i>	22
<i>Pre-requisites</i>	23
<i>Downloading the source.....</i>	23
Installation procedure.....	25

<i>Verifying directory structure</i>	27
<i>Adding postgres user</i>	27
<i>Creating data directory</i>	28
Initializing PostgreSQL	28
<i>Validating the data directory</i>	30
<i>Start PostgreSQL database</i>	31
<i>Verify postgres process is running</i>	31
Binary installation	33
<i>Create repository configuration</i>	34
<i>Import the repository signing key</i>	35
<i>Update the package list</i>	35
Installing PostgreSQL	36
Conclusion	38
Bibliography.....	39
3. Modern Options to get PostgreSQL	41
Introduction.....	41
Structure	41
Objectives.....	42
Other ways to get PostgreSQL.....	42
<i>On-premise, virtualization, containers, and cloud</i>	42
<i>On-premise</i>	42
<i>Virtualization</i>	43
<i>Containers</i>	44
<i>The Cloud</i>	45
Getting PostgreSQL on modern systems	46
<i>PostgreSQL on Docker</i>	47
<i>PostgreSQL on Kubernetes</i>	48
<i>PostgreSQL on The Cloud</i>	53
Conclusion	58
Bibliography.....	58
4. Global Objects in PostgreSQL	59
Introduction.....	59
Structure	59
Objectives	59

Users/Groups/Roles	60
Tablespaces.....	64
Databases.....	65
<i>CREATE DATABASE command</i>	67
<i>createdb program</i>	68
<i>Using pgAdmin Wizard</i>	69
Conclusion	72
Bibliography.....	72
5. Architecture of PostgreSQL.....	73
Introduction.....	73
Structure	73
Objectives.....	74
Memory architecture	74
<i>Shared memory</i>	74
<i>Shared buffers</i>	75
<i>WAL buffers</i>	75
<i>CLOG buffers</i>	75
<i>Process memory</i>	76
<i>Temporary buffers</i>	76
<i>Work memory</i>	76
<i>Maintenance work memory</i>	77
Background processes	77
<i>Postmaster</i>	77
<i>Checkpointer</i>	78
<i>Writer or background writer</i>	78
<i>Autovacuum</i>	79
<i>Stats collector</i>	79
<i>Logger</i>	79
<i>Archiver</i>	79
<i>WAL writer</i>	80
<i>WAL sender</i>	80
<i>WAL receiver</i>	81
Physical files.....	81
<i>Data files</i>	84
<i>WAL files</i>	84

<i>Temporary files</i>	85
<i>CLOG files</i>	86
<i>Stat files</i>	86
<i>Log files</i>	86
<i>WAL archive files</i>	86
<i>Conclusion</i>	87
Bibliography	89
6. PostgreSQL Internals	91
Introduction.....	91
Structure	91
Objectives	92
ACID	92
<i>Atomicity</i>	92
<i>Consistency</i>	92
<i>Isolation</i>	92
<i>Durability</i>	92
MVCC.....	93
Vacuum.....	94
<i>Autovacuum</i>	94
<i>VACUUM FULL</i>	95
<i>Manual VACUUM</i>	95
<i>pg_repack</i>	96
<i>Preventing transaction ID wraparound failures</i>	97
Transaction isolation levels.....	97
<i>Phenomena</i>	97
<i>Dirty read</i>	97
<i>Non-repeatable read</i>	98
<i>Phantom read</i>	99
<i>Serialization anomaly</i>	99
<i>Isolation levels</i>	100
<i>Read uncommitted/committed</i>	100
<i>Repeatable read</i>	101
<i>Serializable</i>	102
Query processing	102
<i>Parser</i>	103

<i>Rewriter</i>	104
<i>Planner</i>	104
<i>Executor</i>	105
Conclusion	105
Bibliography.....	106
7. Backup and Restore in PostgreSQL	107
Introduction.....	107
Structure	107
Objectives.....	108
Backup	108
<i>Physical backup</i>	109
<i>pg_basebackup</i>	109
<i>Point in time Recovery/Archival</i>	111
<i>Pros and cons of physical backup</i>	111
<i>Logical backup</i>	112
<i>pg_dump</i>	112
<i>pg_dumpall</i>	112
<i>Pros and cons of logical backup</i>	115
Restore	115
<i>psql</i>	115
<i>pg_restore</i>	117
Useful backup and restore tools	118
<i>pgBackRest</i>	119
<i>Barman</i>	125
<i>pg_probackup</i>	130
Conclusion	133
Bibliography.....	134
8. Replicating Data.....	135
Introduction.....	135
Structure	135
Objectives.....	135
Physical replication	136
<i>Hot standby</i>	138
<i>Archive recovery</i>	139

<i>Streaming replication</i>	141
<i>Cascading</i>	142
<i>Delayed replica</i>	144
<i>Configuration</i>	146
Logical replication.....	146
<i>Architecture</i>	147
<i>Publication</i>	147
<i>Subscription</i>	148
<i>Publisher node as well as subscription node</i>	148
Conclusion	150
9. Security and Access Control	151
Introduction.....	151
Structure	151
Objectives.....	152
Authentication.....	152
<i>The pg_hba.conf</i>	152
<i>local</i>	153
<i>host</i>	153
<i>hostssl</i>	154
<i>hostnossl</i>	154
<i>hostgssenc</i>	154
<i>hostnogssenc</i>	154
<i>Database</i>	154
<i>User</i>	155
<i>Address</i>	155
<i>Method</i>	156
<i>[Options]</i>	156
Authentication methods.....	157
<i>The pg_ident.conf</i>	158
<i>Examples</i>	159
Authorization.....	161
<i>Role attributes</i>	161
<i>Object ownership</i>	163
<i>Objects privileges</i>	164
Conclusion	165

Bibliography.....	165
10. Most used Extensions/Tools.....	167
Introduction.....	167
Structure	167
Objectives.....	168
Extensions	168
<i>pg_cron</i>	169
<i>pg_stat_statements</i>	171
<i>pg_repack</i>	174
<i>Tools</i>	177
<i>pgbadger</i>	177
<i>pgbench</i>	179
<i>pgbouncer</i>	181
Conclusion	183
Bibliography.....	183
11. Basic Database Objects	185
Introduction.....	185
Structure	185
Objectives.....	185
Managing schemas.....	186
<i>DB cluster</i>	186
<i>Users/roles</i>	188
<i>Databases</i>	188
<i>Tablespaces</i>	188
<i>Schemas</i>	188
<i>Default - Public Schema</i>	188
<i>SEARCH_PATH in Schema</i>	189
Managing DB Objects using DDL commands	192
<i>Data types</i>	192
<i>Table</i>	193
<i>Create table</i>	193
<i>Alter table</i>	194
<i>Drop table</i>	194
<i>Truncate</i>	194

<i>View</i>	195
<i>Sequences</i>	195
Enforcing data integrity using constraints.....	197
Manipulating data using DML Queries	201
<i>Inserting data</i>	201
<i>Updating data</i>	202
<i>Deleting data</i>	203
<i>Select (Retrieve) data</i>	203
<i>Joins used in data retrieval</i>	204
<i>Inner join</i>	204
<i>Left outer join</i>	206
<i>Right outer join</i>	207
<i>Full outer join</i>	208
<i>Aggregate functions</i>	209
Conclusion	210
Bibliography.....	210
12. Advance Database Objects.....	211
Introduction.....	211
Structure	211
Objectives.....	212
Managing procedures/functions	212
<i>Function</i>	212
<i>Function execution syntax</i>	214
<i>Function execution example</i>	215
<i>Procedure</i>	216
<i>Procedure execution syntax</i>	217
<i>Procedure execution</i>	219
Managing triggers	222
<i>Trigger function</i>	222
<i>Event trigger</i>	227
Managing rules	227
<i>Trigger versus rules</i>	229
Custom data type	230
Conclusion	231
Bibliography.....	231

13. Performance Tuning	233
Introduction.....	233
Structure	233
Objectives.....	234
Indexes.....	234
<i>Reindex.....</i>	235
<i>Index types.....</i>	236
<i>Btree index.....</i>	236
<i>Hash index.....</i>	237
<i>GiST and SP-GiST index</i>	237
<i>Gin index.....</i>	237
<i>Brin index.....</i>	237
<i>Indexes and expressions</i>	237
Statistics	238
<i>Statistics in pg_statistics.....</i>	240
<i>Statistics in pg_statistics_ext_data.....</i>	242
<i>Functional dependencies.....</i>	243
<i>Number of distinct values counts.....</i>	244
<i>Most common values list.....</i>	246
Explain plan	247
Best practices for the postgresql.conf parameters	250
<i>shared_buffers</i>	251
<i>work_mem.....</i>	251
<i>autovacuum.....</i>	251
<i>effective_cache_size.....</i>	252
<i>maintenance_work_mem</i>	252
<i>max_connections.....</i>	252
Summary	253
Conclusion	253
Bibliography.....	253
14. Troubleshooting	255
Introduction.....	255
Structure	255
Objectives.....	256
Debugging using log files	256

<i>Where to log</i>	256
<i>logging_collector</i>	256
<i>log_destination</i>	257
<i>log_directory</i>	257
<i>log_filename</i>	258
<i>log_rotation_age</i>	258
<i>log_rotation_size</i>	258
<i>log_truncate_on_rotation</i>	258
<i>What to log</i>	259
<i>log_line_prefix</i>	259
<i>log_connections/log_disconnections</i>	260
<i>log_min_duration_statement</i>	260
<i>log_lock_waits</i>	260
<i>log_autovacuum_min_duration</i>	260
<i>Parameters summary</i>	261
Debugging using PostgreSQL tools and commands	261
<i>Gather information</i>	262
<i>Check the PostgreSQL version</i>	262
<i>Check database and objects size</i>	262
<i>Check database connections</i>	263
<i>Check slow queries</i>	264
<i>Instruct PostgreSQL</i>	266
<i>Vacuuming and analyzing</i>	266
<i>Terminate queries or user sessions</i>	266
<i>Manage replication</i>	267
Debugging using Operating System tools and commands	268
<i>Service and system-wide tools</i>	268
<i>systemctl</i>	268
<i>free</i>	269
<i>df</i>	270
<i>Processes-oriented tools</i>	270
<i>ps/pgrep</i>	271
<i>Log and events</i>	272
<i>journalctl</i>	272
Conclusion	273

15. Contributing to PostgreSQL Community	275
Introduction.....	275
Structure	275
Objectives.....	276
PostgreSQL community and its members	276
<i>Core members of the PostgreSQL community.....</i>	276
<i>Working pattern of PostgreSQL community</i>	277
<i>Earning of PostgreSQL community</i>	278
Different ways to contribute	279
<i>Code contributor</i>	279
<i>Bug reporter.....</i>	280
<i>Participate in the mailing lists</i>	280
<i>Improving or creating new documentation</i>	281
<i>Participating in events</i>	282
<i>Supporting the community</i>	283
Conclusion	283
Index	285

CHAPTER 1

Introduction to

Opensource Database

- PostgreSQL

Introduction

PostgreSQL, also known as “Postgres,” is a popular open-source database management system. It is known for its strong support for reliability, data integrity, and concurrency. Postgres has a large and active development community and is widely used in businesses, government agencies, and other organizations around the world.

Its path in the industry as open-source software is the result of the initiative and efforts of many people through the years.

Structure

This chapter introduces the open-source concept and covers the main general aspects of PostgreSQL like its history, how it was developed and released and how much it has impacted the current industry scene.

Topics to be covered:

- OpenSource Introduction
- A Brief History of PostgreSQL

- PostgreSQL Release Cycle
- Current impact of PostgreSQL in the market

Objectives

You will learn about the establishment of the open-source software concept and how it became what it is nowadays. We will review the history of PostgreSQL from its initial conception and the way it evolved to “The World’s Most Advanced Open Source Relational Database.”

Then you will learn how the release cycle of PostgreSQL works to deliver new features, fixes, and improvements with every minor and major release. Finally, we will look at the current panorama of PostgreSQL and the main advantages you could get when choosing PostgreSQL.

Open-source introduction

Open source refers to a type of software whose source code is available to the public, meaning that anyone can view, modify, and distribute the code. Open-source software is typically developed by a community of volunteers, who work together to improve the software and share their modifications with others.

The origin of free software

Free software, also known as “libre software,” is software that is distributed with the freedom to use, study, modify, and distribute the software and its source code. The concept of free software has its roots in the early days of computer programming, when programmers would freely share their code and ideas with each other.

By the mid-1970s, companies created the software with rigorous licenses to *protect* it and ensure all the profit for its usage goes to the creator; this is what we know as proprietary software or closed software.

After years, other people found this way of creating and distributing software tedious and frustrating. Some thought they could improve the existing software, but their ideas were not viable since the vendors spread the software without its source code, and multiple laws did not allow modifications. The following are some of the main events in the free software history.

- In the early 1980s Richard M. Stallman announced GNU (from GNU is Not Unix), the first genuine free software initiative. He aimed to release an

operating system everyone could get, use, and distribute. He also started the ideology of free software. In his own words, all the software should be free and “accessible to everyone as freely as possible.” (*Reference: Free Software Foundation*)

- A couple of years later, in the mid-1980s, the **Free Software Foundation (FSF)** was established, with Richard as its president.
- In the late 1980s, the Free Software Foundation published the **GNU General Public License (GPL)**. This license’s essence is to clarify that all the software created under it will be free, so everyone can run the software, study it (get the source code), distribute it, make changes, and share it. This license introduced significant changes to how the software was created and opened the door to various new free software projects.

The ideology of free software had some caveats. It got a philosophical-political solid sense. The concept was beyond that everyone should be able to access the software without paying; it represented the idea that the software is a human right, which should be taken and accepted by other elemental organisms such as the governments.

Also, the “label” free software usually requires to be distinguished between the concept of free (no charge) and free (as in freedom), which caused some confusion in general.

The open source concept

The open-source concept is based on the idea that the source code for a piece of software should be freely available to anyone, and that anyone should be able to use, modify, and distribute the software without restriction. This allows for collaboration and transparency in the development process, as anyone can contribute to the software and see how it works.

Open-source software is typically developed using a decentralized, distributed model, with contributions coming from a wide range of individuals and organizations. This can make it more resilient to changes in leadership or funding, as there is no single entity in control of the software.

By the late 1990s, the people supporting the *free* software initiatives considered introducing some adjustments to the free software concept. Two important events heavily impelled their interest in collaborative and shared software development: 1) the release of the Netscape browser source code and 2) the impact of the Linux kernel.

These happenings led to think the label “free” was not precisely adequate to describe the actual situation of these and any other project created with this development ideology. Then a new *code word* came into the scene: the **open-source** software.

We can look at some details from the two main projects that impulse this change.

Netscape browser:

- The Netscape browser was widely used, they used to release their software as any other proprietary software. But motivated by the idea of having the help of many developers around the world, the company decided to make their code open source.
- Making the browser code open source their vast existing community actively participate in the development process, making the bug identification and fix creation and release a more quick and efficient process.
- Undoubtedly, many other companies and organizations consider open source a viable option.

The Linux Kernel:

- The creator of the Linux kernel is the Finnish software engineer Linus Torvalds.
- He started his work just trying to adapt the “home” version of **UNiplexed Information Computing System** (UNIX) called MINIX (from Mini UNIX) to use on his **Personal Computer** (PC).
- As the project started with the idea of free software, some GNU tools where used, so multiple people related to the existing GNU projects got interested.
- The number of users and collaborators of the Linux project multiplied very quickly.
- People from many different places integrated into the community with diverse backgrounds and interests.
- This apparently “non-organized” way of work proved solid and highly effective.
- All the stages required in the development process, such as tests, bug identification, fixes, releases, and so on., got executed highly faster than the usual time.

The speed of the community growth, the quality of the products, and all the popularity these projects gained set them in a notorious place in the industry and a big part of this success was the usage of the open-source methodology.

These two relevant events made adopting the open-source concept more accessible and transparent. Undoubtedly, the initial efforts from the **Free Software Foundation (FSF)** put things rolling in the direction of getting software differently from the privative model. However, its orientation to have a political-looking side and the way their members built the projects, mainly by small groups or even single dedicated persons, was overpassed by the collaborative communities' notorious engagement. The open-source label got a meaning that many people felt identified with.

The open-source software will keep its path regarding making the software available for everyone. It also promotes a collaborative and distributed way of development. It is not making the idea that software is a human right its primary purpose, but that anyone can participate and share efforts to build it. Please find the high-level overview of Open-source timeline in the *Figure 1.1*:

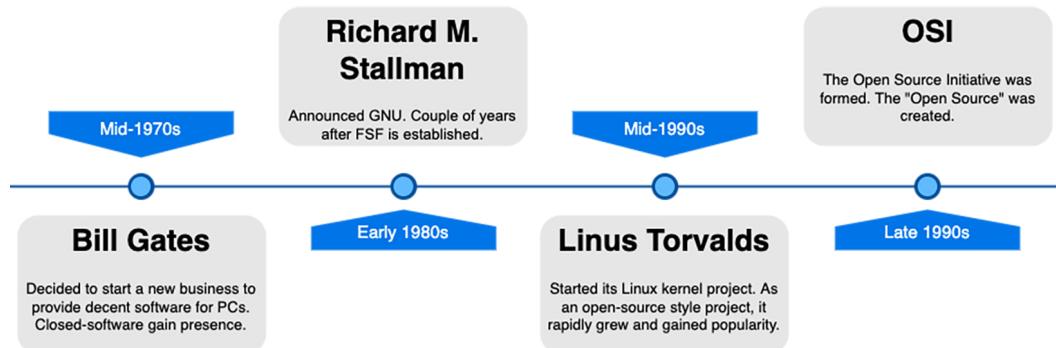


Figure 1.1: Open Source timeline

The open source nowadays

The Open Source has grown in adoption and usage. Multiple big projects are created under its ideology and support the concept that everyone can use and improve the software. The communities of the open-source projects usually are very active, and the results of their collaborative efforts are remarkable.

The way these communities work lets them provide early access releases and make the beta, or even delta, versions available, unlike the closed software (A.K.A privative software). Which usually only releases “complete” final versions of their products and fixes the bugs through their internal process. The open-source model leverages the capacity of the community of users, often collaborators, to find and fix the bugs and issues from the early access releases. So, following the release cycle, these will be addressed and solved by the time the alpha or stable release is made public.

All these characteristics have made open source reliable and sufficient for large enterprise solutions. Modern software takes the best from the collaborative model and the dynamism of the user's communities and software engineers to build compelling, secure, and resilient solutions. We can get coverage for all our system layers, backend, middleware, frontend, security, high availability, and so on., from the current open-source options.

And you know what? Our loved PostgreSQL is one of these open-source options. Now that you know what Open Source is, let's dive into PostgreSQL.

A brief history of PostgreSQL

PostgreSQL, also known as "Postgres," is a powerful and feature-rich open-source database management system. It was first developed at the University of California, Berkeley in the mid-1980s as a research project by a team led by computer science professor Michael Stonebraker.

The POSTGRES project

The POSTGRES project was a research project at the University of California, Berkeley that developed the first version of the PostgreSQL database management system. The project was led by computer science professor Michael Stonebraker and a team of researchers, and it was funded by the **Defense Advanced Research Projects Agency (DARPA)** and the **National Science Foundation (NSF)**.

The goal of the POSTGRES project was to develop a database management system that could handle complex data types and support user-defined types and functions.

By 1993, the number of users from the external community was twice. The participation of these users and their findings to improve the product made project maintenance more difficult. Some members of the original project thought the time they were using to address the code issues and support could be used for development and research. So finally, they decided to terminate the Berkeley POSTGRES project that year; the final release was version 4.2.

The original POSTGRES project found application in several different projects. The financial market, the aerospace industry, medical, and several geographic projects used the software for their purposes. It was commercialized by Illustra, merged into Informix, and finally bought by IBM.

Postgres95

The next part is the beginning of the history of our current PostgreSQL as an open-source database.

By the mid-1990s, after the termination of the initial POSTGRES projects. Andrew Yu and Jolly Chen, two Ph.D. students from Stonebraker's laboratory, took the original code and made many changes and improvements, the more relevant one was the addition of an SQL interpreter, and then they released it to the web. This new project, inspired by the POSTGRES project and continuing its legacy, started its history as an open-source database.

PostgreSQL

After a couple of years, the new name would not stand the pass of time. So it was re-launched in 1996 as PostgreSQL version 6.0. The name was chosen to keep the relationship with the original POSTGRES project and added the SQL suffix to make clear the support of the SQL language; the original project used a query language called PostQUEL. Also, the version number was set to keep continuity from the releases of the previous project. Even now, the nickname *Postgres* is still used for easy pronunciation and keeps remembering its origins.

This just released system rapidly got the attention of multiple hackers and experts from the databases world around the globe. All they got compromised to the improvement and development of new features. The contributions of this community added to PostgreSQL a great code consistency, fixed an innumerable number of bugs, created a mail list to report bugs and usability issues, tested the quality of the software extensively, and filled the documentation gaps for both developers and users.

The PostgreSQL community behind all this work is known as The PostgreSQL Global Development Group. The code comes from contributions from proprietary vendors, support companies, and open-source programmers. Its presence on the web at the www.postgresql.org site started on October 22, 1996, and since then, this portal has gained an essential place in the open-source world.

PostgreSQL versions are released under the **Free and Open-source Software (FOSS)** license. This means anyone can get, execute, study, change and distribute the software (the free part). Also, the source code is openly shared, and people are encouraged to participate voluntarily in improving the software (the open-source part).

PostgreSQL versions key features

PostgreSQL has undergone many versions and updates since its inception, and each version has introduced new features and improvements. Here are some key features that have been introduced in various versions of PostgreSQL.(Reference: *About PostgreSQL*)

Versions 6.0 - 8.0

- **Multiversion Concurrency Control (MVCC).** Table-level locking was replaced with a sophisticated multiversion concurrency control system, which allows readers to continue reading consistent data during writer activity and enables online (hot) backups while the database is running.
- Important SQL features. Many SQL enhancements were made, including subselects, defaults, constraints, primary keys, foreign keys, quoted identifiers, literal string type coercion, type casting, and binary and hexadecimal integer input.
- Improved built-in types. New native types were added.
- Speed. Major speed and performance increases of 20-40% were made, and backend startup time was decreased by 80%. (Reference: *PostgreSQL Community History*)
- **Write-Ahead Log (WAL)** is an efficient mechanism to log all the data changes into files that can be used to replay on top of data files after a checkpoint. This brings the possibility of performing recovery after a system crash and Point-In-Time Recovery (PITR).
- SQL schemas. Logical containers to organize the tables and other relational objects in a database.
- TOAST. Additional data files to handle those records can surpass the size of the data pages, usually 8KB.
AUTOVACUUM. An automated background process to trigger the VACUUM operations over the database tables.

Versions 8.1 - 9.6:

- **Window functions:** Powerful SQL functions to perform calculations over a set of rows related to the current one. Used mainly for analytics and specialized reports.
- **Background Checkpointer:** Background process in charge of triggering the checkpoint in a database. This will throttle the write of data changes from

memory (cache) to the physical data files (disk). So the IO impact on the system is lower as possible.

- **Parallel query:** A feature that enabled the capacity to create a coordinator process and some workers to split the reads operation over large tables, speeding up the query performance.
- **Unlogged tables:** A special table type that does not log its data changes to the WAL. This is especially useful to boost the performance in certain write workloads that can be temporary or recreated and don't need crash recovery protection.
- **Foreign table inheritance:** A table relation technique that enables the possibility of creating a "master" or template table so that others can inherit from it. This also made possible the first table partitioning solution.
- **Streaming Replication:** A versatile and robust data replication mechanism. Built on top of the WAL recovery feature, this replication mode lets a standby get the data changes from the primary WAL directly without needing to copy the physical file.
- **Hot Standby.** When a replica or standby is configured with this option, the instance becomes available for reading operations. This permits design clusters to scale out the reads without impact in the primary instance.
- **Extension Installation.** This enabled the possibility of extending the core features of PostgreSQL by adding extensions, which are pieces of code that the PostgreSQL database can execute once installed.

Versions 10 – 14:

- **B-tree deduplication.** A new enhancement for the general-purpose index structure B-tree avoids adding duplicated values, for example, when creating an index in a column that allows NULL values.
- **REINDEX CONCURRENTLY.** An addition that allows rebuilding indexes in stages, avoiding a prolonged lock in the table so other operations can continue working.
- **Declarative table partitioning.** When this was added, PostgreSQL could create partitioned tables with native syntax, an improvement from the previous inherit approach.
- **Logical replication.** Even when the support for logical decoding was added in the 9.4 version, the native syntax was added until version 10. Now is possible the establish the replication technique without needing external extensions.

- **Logical replication for partitioned tables.** An improvement to the previous one can now be logically replicated from and to partitioned tables. The tables can be partitioned or not in the source or target.

The cumulative work from the community has provided PostgreSQL with a robust, secure, and performant core. Which has turned it into one of the favorite open-source options for modern application development projects. Also, extending the core functions with the extensions or running custom code from different programming languages, such as Perl, Python, or TCL, opens the door to many new options to adjust PostgreSQL to various projects and solutions.

PostgreSQL in stats on a single image

The history of PostgreSQL as an open-source project has been impressive. The number of individual stories from its community might be outstanding, from the most experienced collaborators to the newest users getting started with PostgreSQL; everyone has thoughts and feelings about how their path working with Postgres has been. And one can bet, one way or another, there should be satisfaction and rewards on it. Please find the PG Statistics in *Figure 1.2*:

What's a database project without statistics?

30+ Years Development	650+ Contributors	52,000+ Commits	55+ Local User Groups
1,500,000+ Lines of C	650+ Events	Millions of Happy Users	∞ Data Stored

Figure 1.2: PG statistics.

PostgreSQL release cycle

As shown in the graph (*Figure 1.3*), the first version of PostgreSQL was released in 1996, and in 2022, PostgreSQL 15 version will be released.:(*Reference: Versioning*)

PostgreSQL Major Version Releases

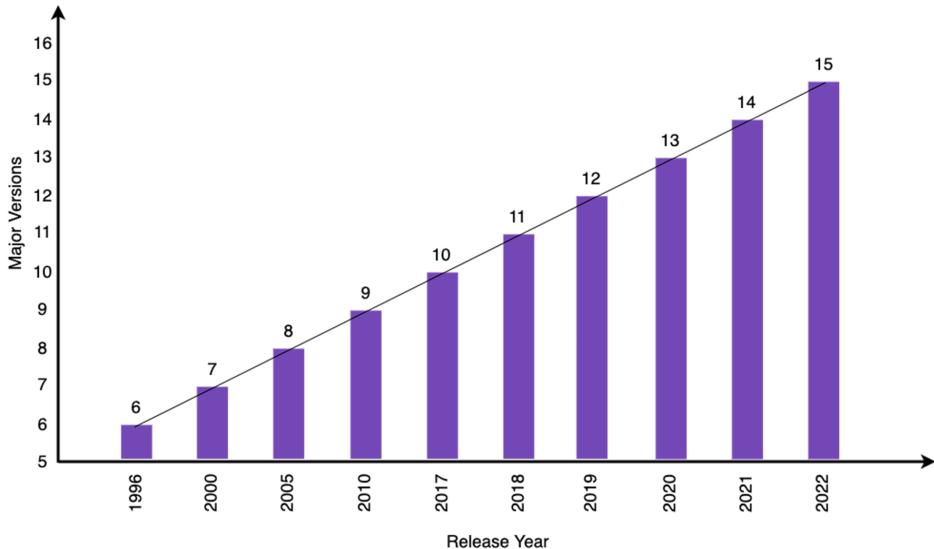


Figure 1.3: PostgreSQL Major Version Release Timeline

PostgreSQL Global Development Group (aka PostgreSQL Community) is responsible for releasing different versions of PostgreSQL. Normally major versions are released once a year - towards the end of Q3 or the beginning of Q4. The minor version containing bug fixes and security patches is released every three months unless some version needs immediate important updates that cannot wait for the regular release cycle.

Before Postgres version 10, version number contained three parts: 9.6.1, 9.6.2, and so on. And after version 10, all versions are released with 2 parts in the numbers: 10.1, 10.2, and so on.

This vibrant and global community of PostgreSQL supports any version till 5 years of its release. So it is recommended to upgrade the Postgres after checking proper release notes. At least the minor version of PostgreSQL should be the latest as per recommendation from Community. The Major Versions of PostgreSQL generally release new developments, major functionalities, and features.

Please find the *Table 1.1* containing release dates for each major version:

Major Version Release	Latest Minor Version	First Release	Latest Release	End of Life
6	6.3.2	01-Mar-98	07-Apr-98	01-Mar-03
7	7.0.3	08-May-00	11-Nov-00	08-May-05
8	8.0.26	19-Jan-05	04-Oct-10	01-Oct-10
9	9.0.23	20-Sep-10	08-Oct-15	08-Oct-15
10	10.22	05-Oct-17	11-Aug-22	10-Nov-22
11	11.17	18-Oct-18	11-Aug-22	09-Nov-23
12	12.12	03-Oct-19	11-Aug-22	14-Nov-24
13	13.8	24-Sep-20	11-Aug-22	13-Nov-25
14	14.5	30-Sep-21	11-Aug-22	12-Nov-26
15	15 Beta	-	11-Aug-22	-

Table 1.1: PostgreSQL Major Version Release Data

The current impact of PostgreSQL on the market

PostgreSQL is a popular and widely used open-source database management system. It is known for its robust feature set, performance, and reliability, and it is used in a variety of applications, including data warehousing, web development, and data analysis.

PostgreSQL has been around since 1986, and it keeps getting matured day by day with its strong community at its base. Since it is open source and among the TOP stable databases with extensive support of vast features, many startups / companies are officially supporting and developing their own version of PostgreSQL or working towards enhancing its functionalities.

According to recent surveys and reports, PostgreSQL is currently one of the most popular open-source databases in the market, with a strong user base and a large and active development community. It is often cited as a top choice for organizations that are looking for a powerful, reliable, and cost-effective database solution.

PostgreSQL is used by a wide range of businesses, government agencies, and other organizations around the world, and it has a strong reputation in the market. It is often seen as a viable alternative to proprietary databases, such as Oracle and

Microsoft SQL Server, and it is widely supported by a range of tools, libraries, and frameworks. Survey results are shown in *Figure 1.4*: (Reference: StackOverflow)

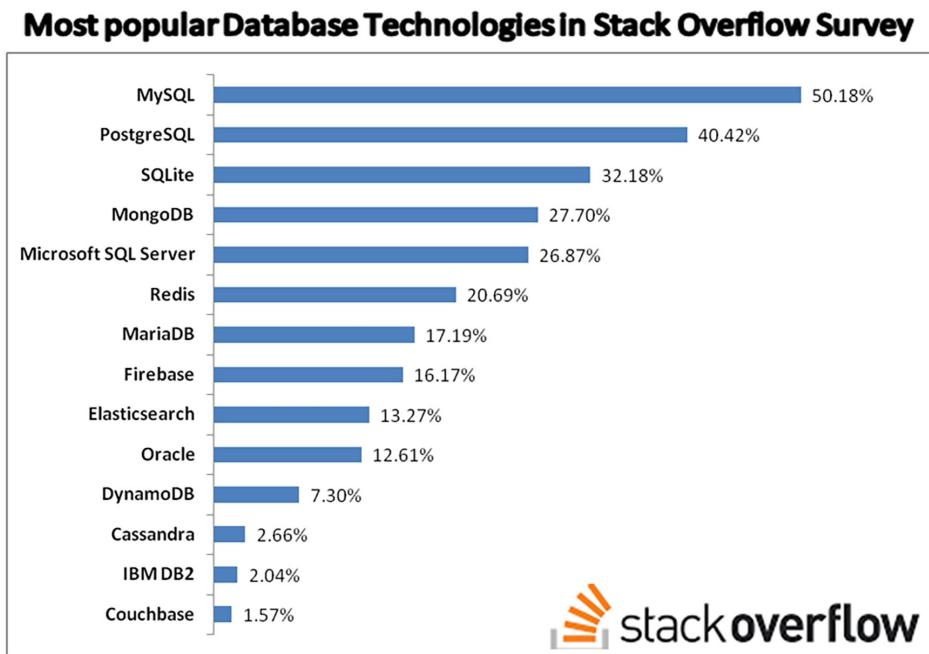


Figure 1.4: Popular DB Technologies

Over the years, PostgreSQL Community and many individual contributions have extensively worked on improving the user experience easier and better. This makes developers consume more of PostgreSQL DB and love the same. (Reference: Info World)

PostgreSQL helps programmers create new applications and admins better protect data integrity, and developers build resilient and secure environments. PostgreSQL allows its users to manage data, regardless of how large the database is. One can say that the sky is the limit. It supports all the most popular operating systems like Windows, all flavors of Unix & Linux, and whatnot. It provides extensive customization support like user-defined data types and user-defined functions. One can also override and inherit the built-in functionalities to create custom functionalities - and too totally free of cost. Also, one can use any programming language like C, C++, Python, or Perl to create customization. (Reference: LearnSQL)

Companies that use PostgreSQL

Since it is a great Database System, well-known market leaders are using the same. It is perfect for analytical tasks. It is good for financial applications because of OLTP and ACID Compliant. Many Web Applications are developed keeping PostgreSQL in their database layer. It is not only good for BI & Data Science applications but also for Government applications involving GeoSpatial Data which needs to be super secured.

Examples of major players using PostgreSQL is shown in *Figure 1.5*:



Apple: Tech Giant Apple has replaced MySQL and started using PostgreSQL since Oracle has licensed MySQL. Also, PostgreSQL is the default database in MacOS Servers and is also available on the Apple Store.



IMDb: The world's largest database of entertainment businesses is stored online by IMDB. Multiple parts of this database are handled in PostgreSQL.



Instagram: Many people in the world cannot imagine their life without Instagram. It is not only a social media app but also one of the major sources of income for many from 1 Billion+ insta user. They are also using RDBMS, and PostgreSQL is used to handle one of the main tasks.



Skype: VoIP-based videotelegrapgy App used majorly for calling, as chat messenger, and for file transfer uses PostgreSQL's pgBouncer tool for load balancing purposes. It also uses features like Stored Procedures, pgQ, and many more functionalities.



International Space Station (aka NASA): Yes, PostgreSQL is reached to space station too. NASA has implemented Nagios on the Space Station, using PostgreSQL to store the data in orbit and then replicate that database on the ground.

Figure 1.5: Major Players Using PostgreSQL

Apart from the above big Giants - companies like GroupON, MasterCard, Delivery Hero, Fujitsu, and so many others are using PostgreSQL as one of the important RDBMS databases in their organization.

Companies that help in enhancing PostgreSQL

PostgreSQL is an open-source database management system, which means that it is developed and maintained by a global community of volunteers. There are many companies and organizations that contribute to the development and improvement

of PostgreSQL, either through direct contributions to the codebase or through the development of tools and services that support the database.

Some examples of companies that have made significant contributions to PostgreSQL include:

- **EnterpriseDB**: A company that provides commercial support and services for PostgreSQL, as well as tools and products that enhance the database. EnterpriseDB is a major contributor to the PostgreSQL project and has been instrumental in the development of several features and improvements in the database.
- **2ndQuadrant**: A company that provides consulting, training, and support services for PostgreSQL, as well as tools and products that enhance the database. 2ndQuadrant is a major contributor to the PostgreSQL project and has been involved in the development of many features and improvements in the database.
- **Pivotal**: A company that provides tools and services for data management, including support for PostgreSQL. Pivotal is a major contributor to the PostgreSQL project and has been involved in the development of many features and improvements in the database.

Along with these organizations, there are multiple big names associated with PostgreSQL that are working towards enhancing its capabilities along with the PostgreSQL community. These companies are either contributing to the community by developing more open-source tools/utilities or working towards creating chargeable tools/utilities that make PostgreSQL better daily.

- Google is developing its own PostgreSQL by customizing the community PostgreSQL.
- AWS supports PostgreSQL by AWS RDS or Aurora DB in AWS Cloud.
- Timescale Inc. is another organization that has extended PostgreSQL, which can help work with Time Series Data easily.
- EnterpriseDB Corporation has created multiple products on top of PostgreSQL which are handy for multiple functionalities like BART (Backup and Recovery Tool), Migration Toolkit, and many more.
- Percona has developed a monitoring tool called PMM, which helps in monitoring PostgreSQL Database.
- Apart from these, MigOps, Cybertack, Crunchy Data, and many other companies are working towards extending features of PostgreSQL or supporting PostgreSQL.

Advantages of PostgreSQL

PostgreSQL is a powerful and feature-rich open-source database management system that is widely used in businesses, government agencies, and other organizations around the world. It has a strong reputation for reliability, performance, and data integrity, and it offers many advantages over other database systems. Some key advantages of PostgreSQL include:

Vibrant community:

PostgreSQL Community is a group of individuals across the globe with vivid knowledge and experience working for 30+ years to make PostgreSQL better with each passing day. PostgreSQL community is one of the most active communities in the world in the open source technology space. They try their lever best to fix the bugs and answer the queries.

Portable:

PostgreSQL is written in ANSI C, and it supports multiple operating systems like Windows, MacOS, and major flavors of Unix/Linux.

Reliable:

It is an **Atomicity, Consistency, Isolation, and Durability (ACID)** compliant open source database. It is largely compliant with SQL. It supports Transactions using BEGIN blocks and SAVEPOINTS. It uses **Write-Ahead Logging (WAL)**.

Scalable:

It uses **Multiversion Concurrency Control(MVCC)** Architecture. It supports Table Partitioning and Tablespaces. It supports writing database functions in SQL, Perl, Python, C/C++, and many more languages. One can create own custom complex data types and user-defined functions. Also, it supports Object Oriented Relational Database Management System (ORDBMS), where users can overloading and inheritance multiple objects.

Secure:

It employs Host-based Access Control and provides Object level permissions using GRANT and REVOKE. It supports extensive logging by enabling various Log related parameters in its configuration files. One can also use SSL certificates to enhance security.

Replication and High Availability:

It supports replication using various utilities like Streaming Replication, Slony, Cascading, and so on. It also supports High Availability. Patroni is one of the advanced tools for replication & high availability.

Advance features:

- It also supports multiple advanced features like full-text search, Window functions, triggers, rules, and many more keep upgrading with each new version.
- It has good support for JSON and XML.
- PostgreSQL with the [PostGIS extension](#) supports geospatial databases for **Geographic Information Systems (GIS)**.
- It supports advanced indexing techniques like GIST Indexing, GIN Index along with default B-Tree Index.
- One can customize the stored procedures using PL/pgSQL, PL/Perl, PL/Python, PL/PHP, and multiple other languages.
- Since it is a global community, many tools have been developed using Python, PHP, and so on.
- It supports Hot-Backup, and **Point-In-Time-Recovery (PITR)**.
- There are no restrictions on DB Size. One can create a Database of unlimited size or equivalent size of the disk space.

Distributed architecture with PostgreSQL

Distributed architectures involve the use of multiple database servers, which can be located on the same network or on different networks, to store and manage data.

In a distributed architecture with PostgreSQL, multiple database servers can be used to store and manage data in different ways, depending on the needs of the application. Some common ways to use PostgreSQL in a distributed architecture include:

- **Horizontal scaling:** This involves using multiple database servers to handle increased workload or to provide additional capacity. Each database server can be used to store and manage a portion of the data, and the workload can be distributed among the servers.

- **Data partitioning:** This involves dividing the data into smaller chunks and storing each chunk on a different database server. This can be useful for managing large amounts of data or for improving performance by distributing the workload across multiple servers.
- **Replication:** This involves creating copies of the data on multiple database servers, which can be used to provide redundancy and failover in the event of a server failure. PostgreSQL supports various types of replications, including synchronous and asynchronous replication.

Using PostgreSQL in a distributed architecture can provide many benefits, including increased scalability, reliability, and performance. It is important to carefully plan and design the distributed architecture to ensure that it meets the needs of the application and the organization.

StatefulSet with PostgreSQL

Stateless protocol does not store the past information while serving the request to the application.

On the other hand, stateful protocol stores past transaction data while serving the request.

PostgreSQL uses statefulset protocol as it stores data in the tables in the storage space. If this data is stored in RAM, these changes will disappear after a restart. Also, PostgreSQL can be scaled to multiple nodes, which can help make PostgreSQL applications use stateful protocols. Since multiple standby read-only servers are created using the replication and high availability architecture, data is stored consistently across all the nodes.

Conclusion

In this chapter we have learnt about – what is OpenSource Database, how PostgreSQL originated, how release cycle of PostgreSQL works and what is impact of PostgreSQL at the moment in the market.

In the next chapter we are going to learn about the multiple ways to install PostgreSQL in detail.

Bibliography

- Free Software Foundation: <https://www.fsf.org/history/>
- Info World: <https://www.infoworld.com/article/3605138/the-shifting-market-for-postgresql.html>
- LearnSQL: <https://learnsql.com/blog/companies-that-use-postgresql-in-business/>
- PostgreSQL Community History: <https://web.archive.org/web/20170326020245/https://www.postgresql.org/about/history/>
- Versioning: <https://www.postgresql.org/support/versioning>
- About PostgreSQL: <https://www.postgresql.org/about/>
- StackOverflow: <https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-databases>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 2

Getting PostgreSQL to work

Introduction

When the time comes to start “playing” with PostgreSQL, the first thing you need is to get it ready for use. There are a couple of options to install PostgreSQL in our favorite Linux distribution, which are largely used. These are the Source Code and the Binary Installation methods. Let us check them out.

Structure

This chapter presents and describes the main methods to install PostgreSQL in modern Operating Systems. We will cover the generic and most frequent steps used.

The topics to be covered:

- Source Code Installation
- Installation Procedure
- Initialize PostgreSQL
- Binary Installation

Objectives

You will learn the principal methods to install PostgreSQL on modern Linux Operating Systems. You will get actual command examples to accomplish a functional installation. In the end, you will be clear about the options you can choose to get PostgreSQL up and running and ready to work.

Source code installation

One of the most “artisanal” ways of getting PostgreSQL is the Source Code installation. This might remind us of a time when many open-source projects started. When a project started as open-source or turned into one, the first thing was to make the code available, so everyone could get it, compile it and use it.

The installation of PostgreSQL from its source code is the same in almost major Operating Systems; however, some specific installation steps depend upon each operating system. You can find the instructions on the community PostgreSQL website.

Short version of source code installation

As mentioned in the Postgres community website, the source code mentioned in the following code block is the short version of PostgreSQL installation. (*Reference: Source Code Install*)

```
./configure  
make  
su  
make install  
adduser postgres  
mkdir /usr/local/pgsql/data  
chown postgres /usr/local/pgsql/data  
su - postgres  
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data  
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile start
```

```
/usr/local/pgsql/bin/createdb test
```

```
/usr/local/pgsql/bin/psql test
```

The process of creating an executable from the code is called compilation. It is defined as 5 stages: preprocessing, parsing, translation, assembling, and linking. In modern Linux systems, these steps are executed with the combination of the **make** command and the input the **./configure** command provides. These automated the process and turned it into a quicker and easier operation.

In the above steps, we can also see a few directories we can adjust depending on the setup we want. The shown commands work but are intended to illustrate the process.

Pre-requisites

In this example, we will install community PostgreSQL 14.5 on Ubuntu.

Before installing PostgreSQL, please update the system's repository and reboot the same as per command given below:

```
sudo apt update && apt upgrade -y
reboot
```

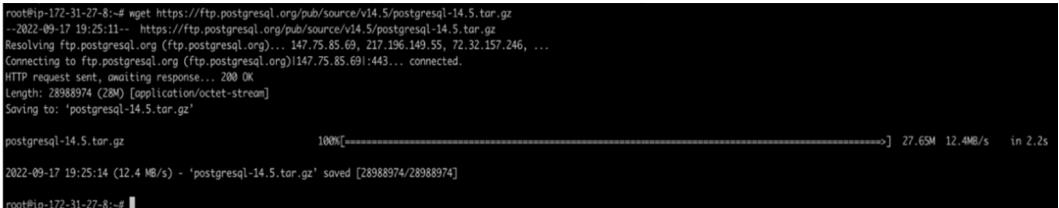
Also, install the following packages as it gives issues while installing PostgreSQL.

```
sudo apt install build-essential zlib1g-dev libreadline-dev -y
```

Downloading the source

Download the source code from **PostgreSQL FTP Server** using **wget** as shown in *Figure 2.1* with the command mentioned in the below code block. (*Reference: PostgreSQL on Ubuntu*)

```
wget https://ftp.postgresql.org/pub/source/v14.5/postgresql-14.5.tar.gz
```



A terminal window showing the execution of a wget command to download the PostgreSQL 14.5 source code. The command is `wget https://ftp.postgresql.org/pub/source/v14.5/postgresql-14.5.tar.gz`. The output shows the progress of the download, including the URL, connection details, file length (28988974 bytes), and download speed (27.65M 12.4MB/s). The download completes successfully in 2.2 seconds.

```
root@ip-172-31-27-8:~# wget https://ftp.postgresql.org/pub/source/v14.5/postgresql-14.5.tar.gz
--2022-09-17 19:25:11-- https://ftp.postgresql.org/pub/source/v14.5/postgresql-14.5.tar.gz
Resolving ftp.postgresql.org (ftp.postgresql.org)... 147.75.85.69, 217.196.149.55, 72.32.157.246, ...
Connecting to ftp.postgresql.org (ftp.postgresql.org)|147.75.85.69|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 28988974 (28M) [application/octet-stream]
Saving to: 'postgresql-14.5.tar.gz'

postgresql-14.5.tar.gz          100%[=====] 27.65M 12.4MB/s    in 2.2s

2022-09-17 19:25:14 (12.4 MB/s) - 'postgresql-14.5.tar.gz' saved [28988974/28988974]
root@ip-172-31-27-8:~#
```

Figure 2.1: Downloading PostgreSQL Source

PostgreSQL FTP Server looks as shown in *Figure 2.2*. Take the name of the exact file name (mentioned towards the end of `wget` command) depending upon the PostgreSQL version which you want to install.

The screenshot shows a web-based file browser for the PostgreSQL FTP server at `postgresql.org/ftp/source/v14.5/`. The page title is "8th September 2022: PostgreSQL 15 Beta 4 Released!". On the left, there's a "Quick Links" sidebar with links to Downloads (Packages, Source), Software Catalogue, and File Browser. The main area is titled "File Browser" with a "Top → source → v14.5" breadcrumb. It has sections for "Directories" (with a "Parent Directory" link) and "Files". Under "Files", there's a list of tarball files with their sizes and last modified dates:

<input type="checkbox"/>	postgresql-14.5.tar.bz2	2022-08-08 20:59:36
<input type="checkbox"/>	postgresql-14.5.tar.bz2.md5	58 bytes
<input type="checkbox"/>	postgresql-14.5.tar.bz2.sha256	90 bytes
<input type="checkbox"/>	postgresql-14.5.tar.gz	27.6 MB
<input type="checkbox"/>	postgresql-14.5.tar.gz.md5	57 bytes
<input type="checkbox"/>	postgresql-14.5.tar.gz.sha256	89 bytes

At the bottom of the page are social media sharing icons for Twitter and LinkedIn, and links to Policies, Code of Conduct, About PostgreSQL, and Contact.

Figure 2.2: PostgreSQL FTP Server

Next is to decompress the `.gz` file using the below command:

```
tar xvzf postgresql-14.5.tar.gz
```

Change the directory to decompressed file and check its contents, as shown in *Figure 2.3*. It should contain an executable file named `configure`, which we will use in the next step.

```
root@ip-172-31-27-8:~# cd postgresql-14.5/
root@ip-172-31-27-8:~/postgresql-14.5#
root@ip-172-31-27-8:~/postgresql-14.5# ls -la
total 808
drwxrwxrwx 6 1107 1107 4096 Aug  8 20:59 .
drwx----- 5 root root 4096 Sep 17 19:28 ..
-rw-r--r-- 1 1107 1107 730 Aug  8 20:44 .dir-locals.el
-rw-r--r-- 1 1107 1107 183 Aug  8 20:44 .editorconfig
-rw-r--r-- 1 1107 1107 9307 Aug  8 20:44 .git-blame-ignore-revs
-rw-r--r-- 1 1107 1107 1579 Aug  8 20:44 .gitattributes
-rw-r--r-- 1 1107 1107 504 Aug  8 20:44 .gitignore
-rw-r--r-- 1 1107 1107 41 Aug  8 20:57 .gitrevision
-rw-r--r-- 1 1107 1107 1192 Aug  8 20:44 COPYRIGHT
-rw-r--r-- 1 1107 1107 4259 Aug  8 20:44 GNUmakefile.in
-rw-r--r-- 1 1107 1107 277 Aug  8 20:44 HISTORY
-rw-r--r-- 1 1107 1107 63944 Aug  8 20:59 INSTALL
-rw-r--r-- 1 1107 1107 1665 Aug  8 20:44 Makefile
-rw-r--r-- 1 1107 1107 1213 Aug  8 20:44 README
-rw-r--r-- 1 1107 1107 445 Aug  8 20:44 aclocal.m4
drwxrwxrwx 2 1107 1107 4096 Aug  8 20:58 config
-rwxr-xr-x 1 1107 1107 588441 Aug  8 20:44 configure
-rw-r--r-- 1 1107 1107 85890 Aug  8 20:44 configure.ac
drwxrwxrwx 58 1107 1107 4096 Aug  8 20:57 contrib
drwxrwxrwx 3 1107 1107 4096 Aug  8 20:58 doc
drwxrwxrwx 16 1107 1107 4096 Aug  8 20:59 src
root@ip-172-31-27-8:~/postgresql-14.5#
```

Figure 2.3: PostgreSQL source code directory

Installation procedure

Once the source code has been downloaded, we need to proceed with the configuration, making the installation and installing the process with the steps mentioned in this section.

Perform the configuration of the system using the `configure` command as follows:

```
./configure
```

```
root@ip-172-31-27-8:~/postgresql-14.5# ./configure
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking which template to use... linux
.
.
src/backend/port/pg_shmem.c
config.status: linking src/include/port/linux.h to src/include/pg_
config_os.h
config.status: linking src/makefiles/Makefile.linux to src/Makefile.port
root@ip-172-31-27-8:~/postgresql-14.5#
```

Once the `./configure` command is done, you will get the `Makefile` input as shown in the code block below, which is required in the next step; build the system using the `make` command:

```
make
```

```
make[2]: Leaving directory '/root/postgresql-14.5/src/makefiles'
make -C test/regress all
make[2]: Entering directory '/root/postgresql-14.5/src/test/regress'
make -C ../../src/port all
make[3]: Entering directory '/root/postgresql-14.5/src/port'
.
```

```
.  
make[2]: Leaving directory '/root/postgresql-14.5/src/test/isolation'  
make -C test/perl all  
make[2]: Entering directory '/root/postgresql-14.5/src/test/perl'  
make[2]: Nothing to be done for 'all'.  
make[2]: Leaving directory '/root/postgresql-14.5/src/test/perl'  
make[1]: Leaving directory '/root/postgresql-14.5/src'  
make -C config all  
make[1]: Entering directory '/root/postgresql-14.5/config'  
make[1]: Nothing to be done for 'all'.  
make[1]: Leaving directory '/root/postgresql-14.5/config'
```

Finally, install PostgreSQL using the **make install** command as per shown in the following code block

```
make install  
  
make[2]: Entering directory '/root/postgresql-14.5/src/makefiles'  
/usr/bin/mkdir -p '/usr/local/pgsql/lib/pgxs/src/makefiles'  
/usr/bin/install -c -m 644 ./pgxs.mk '/usr/local/pgsql/lib/pgxs/src/  
makefiles/'  
make[2]: Leaving directory '/root/postgresql-14.5/src/makefiles'  
make -C test/regress install  
make[2]: Entering directory '/root/postgresql-14.5/src/test/regress'  
. .  
/usr/bin/mkdir -p '/usr/local/pgsql/lib/pgxs/config'  
/usr/bin/install -c -m 755 ./install-sh '/usr/local/pgsql/lib/pgxs/  
config/install-sh'
```

```
/usr/bin/install -c -m 755 ./missing '/usr/local/pgsql/lib/pgxs/config/  
missing'  
make[1]: Leaving directory '/root/postgresql-14.5/config'
```

Verifying directory structure

At this stage, check the directories and files which have been created at the below path shown in the following code block:- (Reference: PostgreSQL on Linux)

```
root@ip-172-31-27-8:/# cd /usr/Local/pgsql/  
root@ip-172-31-27-8:/usr/local/pgsql# ls -la  
total 24  
drwxr-xr-x  6 root root 4096 Sep 17 20:01 .  
drwxr-xr-x 11 root root 4096 Sep 17 20:01 ..  
drwxr-xr-x  2 root root 4096 Sep 17 20:01 bin  
drwxr-xr-x  6 root root 4096 Sep 17 20:01 include  
drwxr-xr-x  4 root root 4096 Sep 17 20:01 lib  
drwxr-xr-x  6 root root 4096 Sep 17 20:01 share
```

Adding postgres user

The PostgreSQL initialization and service execution should be done with a user different from the root user, which is the super admin of the operating system. This is both a best practice recommendation and a security limitation. The postgres operating system user is usually used. Add the postgres user as shown in the following code block:

```
root@ip-172-31-27-8:/usr/local/pgsql# adduser postgres  
Adding user `postgres' ...  
Adding new group `postgres' (1001) ...  
Adding new user `postgres' (1001) with group `postgres' ...  
Creating home directory `/home/postgres' ...  
Copying files from `/etc/skel' ...
```

```
New password:  
Retype new password:  
passwd: password updated successfully  
Changing the user information for postgres  
Enter the new value, or press ENTER for the default  
  Full Name []: postgres  
  Room Number []:  
  Work Phone []:  
  Home Phone []:  
  Other []:  
Is the information correct? [Y/n] Y  
root@ip-172-31-27-8:/usr/local/pgsql#
```

Creating data directory

Once we have the `postgres` user created and the PostgreSQL is installed, we can create a dedicated directory that will contain the data for our database. We know this as the Data Directory. The privileges and ownership should be set to the `postgres` user using the command mentioned in the following code block:

```
root@ip-172-31-27-8:/usr/local/pgsql# mkdir /usr/Local/pgsql/data  
root@ip-172-31-27-8:/usr/local/pgsql# chown postgres:postgres /usr/  
Local/pgsql/data  
root@ip-172-31-27-8:/usr/local/pgsql# ls -ld /usr/Local/pgsql/data  
drwxr-xr-x 2 postgres postgres 4096 Sep 18 15:51 /usr/local/pgsql/data  
root@ip-172-31-27-8:/usr/local/pgsql#
```

Initializing PostgreSQL

At this step, we are ready to initialize the database cluster system. This will populate the just-created data directory with all the required files and folders to start the database system. Please go through the following:

```
root@ip-172-31-27-8:/usr/local/pgsql# su - postgres  
postgres@ip-172-31-27-8:~$ /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

The files belonging to this database system will be owned by user “postgres”.

This user must also own the server process.

The database cluster will be initialized with locale “C.UTF-8”.

The default database encoding has accordingly been set to “UTF8”.

The default text search configuration will be set to “english”.

Data page checksums are disabled.

```
fixing permissions on existing directory /usr/local/pgsql/data ... ok  
creating subdirectories ... ok  
selecting dynamic shared memory implementation ... posix  
selecting default max_connections ... 100  
selecting default shared_buffers ... 128MB  
selecting default time zone ... Etc/UTC  
creating configuration files ... ok  
running bootstrap script ... ok  
performing post-bootstrap initialization ... ok  
syncing data to disk ... ok
```

```
initdb: warning: enabling “trust” authentication for local connections  
You can change this by editing pg_hba.conf or using the option -A, or  
--auth-local and --auth-host, the next time you run initdb.
```

Success. You can now start the database server using:

```
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile  
start
```

```
postgres@ip-172-31-27-8:~$
```

Validating the data directory

Now we can verify the content of the data directory if the **initdb** command was executed successfully, we will see following list of folders as shown in the code block:

```
postgres@ip-172-31-27-8:~$ ls -lrth /usr/local/pgsql/data  
total 120K  
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_twophase  
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_tblspc  
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_stat_tmp  
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_stat  
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_snapshots  
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_serial  
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_replslot  
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_notify  
drwx----- 4 postgres postgres 4.0K Sep 18 15:53 pg_multixact  
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_dynshmem  
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_commit_ts  
-rw----- 1 postgres postgres     3 Sep 18 15:53 PG_VERSION  
-rw----- 1 postgres postgres  29K Sep 18 15:53 postgresql.conf  
-rw----- 1 postgres postgres   88 Sep 18 15:53 postgresql.auto.conf  
-rw----- 1 postgres postgres 1.6K Sep 18 15:53 pg_ident.conf  
-rw----- 1 postgres postgres 4.7K Sep 18 15:53 pg_hba.conf
```

```
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_xact
drwx----- 3 postgres postgres 4.0K Sep 18 15:53 pg_wal
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 pg_subtrans
drwx----- 2 postgres postgres 4.0K Sep 18 15:53 global
drwx----- 5 postgres postgres 4.0K Sep 18 15:53 base
drwx----- 4 postgres postgres 4.0K Sep 18 15:53 pg_logical
postgres@ip-172-31-27-8:~$
```

Start PostgreSQL database

We should be able to start our database system. The next command shown in the below code block will start the new database cluster, allocating all the required memory pools and creating the background processes:

```
postgres@ip-172-31-27-8:~$ /usr/local/pgsql/bin/pg_ctl -D /usr/local/
pgsql/data -l logfile start
waiting for server to start.... done
server started
```

Verify postgres process is running

We can ensure PostgreSQL is up and running searching for the background process. If everything goes well, we will see a similar list of processes running as the postgres operating system user as shown in the following code block :

```
postgres@ip-172-31-27-8:~$ ps -ef | grep postgres
root      14625  14550  0 15:53 pts/1    00:00:00 su - postgres
postgres  14626  14625  0 15:53 pts/1    00:00:00 -bash
postgres  14657      1  0 15:57 ?        00:00:00 /usr/local/pgsql/
bin/postgres -D /usr/local/pgsql/data
postgres  14659  14657  0 15:57 ?        00:00:00 postgres:
checkpointer
postgres  14660  14657  0 15:57 ?        00:00:00 postgres: background
writer
```

```
postgres  14661  14657  0 15:57 ?          00:00:00 postgres: walwriter
postgres  14662  14657  0 15:57 ?          00:00:00 postgres: autovacuum
launcher
postgres  14663  14657  0 15:57 ?          00:00:00 postgres: stats
collector
postgres  14664  14657  0 15:57 ?          00:00:00 postgres: logical
replication launcher
postgres  14677  14626  0 15:58 pts/1      00:00:00 ps -ef
postgres  14678  14626  0 15:58 pts/1      00:00:00 grep --color=auto
postgres
postgres@ip-172-31-27-8:~$
```

Login to the Database using the **/usr/local/pgsql/bin/psql** command as shown following:

```
postgres@ip-172-31-27-8:~$ /usr/local/pgsql/bin/psql
psql (14.5)
```

Type “help” for help.

```
postgres=#
```

Create test database using **/usr/local/pgsql/bin/createdb** command as following:

```
ubuntu@ip-172-31-27-8:~$ sudo su - postgres
postgres@ip-172-31-27-8:~$ /usr/local/pgsql/bin/createdb test
postgres@ip-172-31-27-8:~$ /usr/local/pgsql/bin/psql test
psql (14.5)
```

Type “help” for help.

```
test=# \l+
```

List
of databases

Name	Owner	Encoding	Collate	Ctype	Access
privileges	Size	Tablespace			Description
postgres	postgres	UTF8	C.UTF-8	C.UTF-8	
	8233 kB	pg_default	default	administrative	connection database
template0	postgres	UTF8	C.UTF-8	C.UTF-8	=c/postgres
	+ 8081 kB	pg_default	unmodifiable	empty	database
postgres					postgres=CTc/
template1	postgres	UTF8	C.UTF-8	C.UTF-8	=c/postgres
	+ 8081 kB	pg_default	default	template	for new databases
postgres					postgres=CTc/
test	postgres	UTF8	C.UTF-8	C.UTF-8	
	8233 kB	pg_default			

(4 rows)

Binary installation

Unlike the Source Code installation, this option can save us some steps and time since the executable files are already compiled and ready to use. We just need the proper files for the Linux distribution we want to install.

For convenience, binary installation can also be done using the package manager or by downloading the PostgreSQL packages from the PostgreSQL Community website (that is, <https://www.postgresql.org/download/>) following the instructions given on the webpage as per the chosen operating system.

Go to the **Download** section of the PostgreSQL Community webpage, as shown in *Figure 2.4*. Select the respective Operating System. In this case, we have selected **Linux**, which opens the new list to Linux Distribution, and we need to select **Ubuntu**. Please select the respective operating system depending upon the requirement.

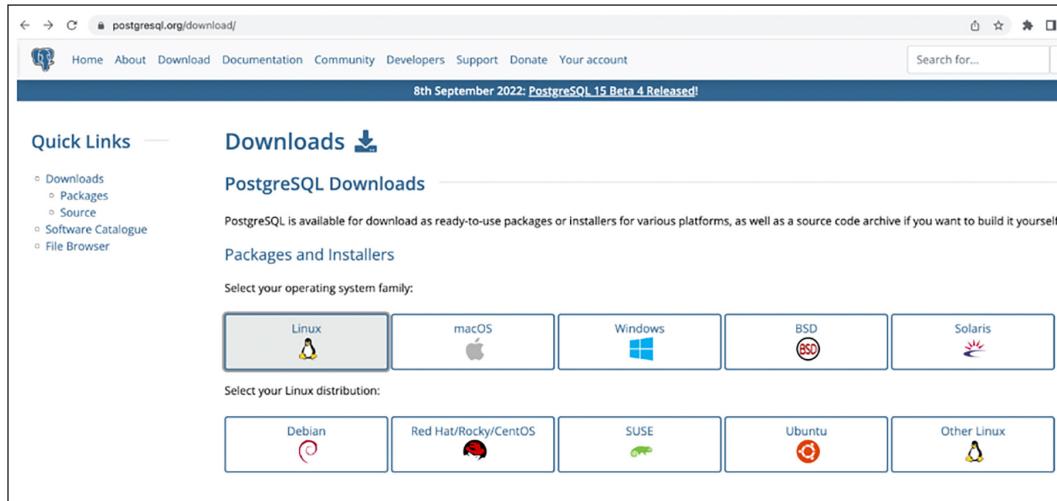


Figure 2.4: PostgreSQL Downloads page on Community Website

Upon selecting **Ubuntu**, it shows the below pretty straightforward steps. Upon following these steps, postgres will be installed:

```
# Create the file repository configuration:  
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'  
  
# Import the repository signing key:  
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -  
  
# Update the package lists:  
sudo apt-get update  
  
# Install the latest version of PostgreSQL.  
# If you want a specific version, use 'postgresql-12' or similar instead of 'postgresql':  
sudo apt-get -y install postgresql
```

Figure 2.5: PostgreSQL Binary Installation Steps for Ubuntu as per Community Website

Create repository configuration

The first step is to add the repository configuration as shown in the code block below:

```
pg@ubuntu-focal:~$ sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'  
  
pg@ubuntu-focal:~$  
  
pg@ubuntu-focal:~$ cat /etc/apt/sources.list.d/pgdg.list  
  
deb http://apt.postgresql.org/pub/repos/apt focal-pgdg main
```

Import the repository signing key

The next step is to import the repository key; this step is required to add the repository key to the trust keys list, this way the package manager **apt** can “talk” with the repository and verify the packages are secure. Pls use the wget command as shown in the following code block:-

```
pg@ubuntu-focal:~$ wget --quiet -O - https://www.postgresql.org/media/
keys/ACCC4CF8.asc | sudo apt-key add -
OK
pg@ubuntu-focal:~$
```

Update the package list

Now that we have added the PostgreSQL repository and its key, we can update the package list. During this step, the package manager will discover all the binary files from the just-added repository. Update the package repositories as shown in the subsequent code block:

```
pg@ubuntu-focal:~$ sudo apt-get update

Hit:1 http://archive.ubuntu.com/ubuntu focal InRelease
...
Get:4 http://apt.postgresql.org/pub/repos/apt focal-pgdg InRelease
[91.7 kB]
...
Get:8 http://apt.postgresql.org/pub/repos/apt focal-pgdg/main amd64
Packages [244 kB]
...
Fetched 24.6 MB in 6s (3813 kB/s)
Reading package lists... Done
```

Installing PostgreSQL

The last step is to install the postgres packages. You can install the latest available version just by installing the **postgresql** packages, as the following:

```
pg@ubuntu-focal:~$ sudo apt-get install postgresql
```

However, from the repository, you can access different versions of PostgreSQL. You can quickly list the available versions using the **apt-cache** command as following:

```
pg@ubuntu-focal:~$ sudo apt-cache search --names-only 'postgresql-[0-9/.]{2,3}$'  
postgresql-10 - The World's Most Advanced Open Source Relational Database  
postgresql-11 - The World's Most Advanced Open Source Relational Database  
postgresql-12 - The World's Most Advanced Open Source Relational Database  
postgresql-13 - The World's Most Advanced Open Source Relational Database  
postgresql-14 - The World's Most Advanced Open Source Relational Database  
postgresql-8.2 - object-relational SQL database, version 8.2 server  
postgresql-8.3 - object-relational SQL database, version 8.3 server  
postgresql-8.4 - object-relational SQL database, version 8.4 server  
postgresql-9.0 - object-relational SQL database, version 9.0 server  
postgresql-9.1 - object-relational SQL database, version 9.1 server  
postgresql-9.2 - object-relational SQL database, version 9.2 server  
postgresql-9.3 - object-relational SQL database, version 9.3 server  
postgresql-9.4 - object-relational SQL database, version 9.4 server  
postgresql-9.5 - The World's Most Advanced Open Source Relational Database
```

```
postgresql-9.6 - The World's Most Advanced Open Source Relational Database
```

Then you can install the version you want, for example, PostgreSQL 14 as shown in the next code block. The next output was shortened just to improve the readability:

```
pg@ubuntu-focal:~$ sudo apt-get install postgresql-14
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libcommon-sense-perl libjson-perl libjson-xs-perl libllvm10 libpq5
  libsensors-config libsensors5 libtypes-serialiser-perl pgdg-keyring
  postgresql-client-14  postgresql-client-common  postgresql-common
  ssl-cert sysstat
Suggested packages:
  lm-sensors postgresql-doc-14 openssl-blacklist isag
The following NEW packages will be installed:
  libcommon-sense-perl libjson-perl libjson-xs-perl libllvm10 libpq5
  libsensors-config libsensors5 libtypes-serialiser-perl pgdg-keyring
  postgresql-14      postgresql-client-14      postgresql-client-common
  postgresql-common  ssl-cert sysstat
0 upgraded, 15 newly installed, 0 to remove and 79 not upgraded.
Need to get 33.9 MB of archives.
After this operation, 137 MB of additional disk space will be used.

Do you want to continue? [Y/n] y
...<truncated>...
Setting up postgresql-14 (14.5-1.pgdg20.04+1) ...
Creating new PostgreSQL cluster 14/main ...
/usr/lib/postgresql/14/bin/initdb -D /var/lib/postgresql/14/main
--auth-local peer --auth-host scram-sha-256 --no-instructions
```

The files belonging to this database system will be owned by user “postgres”.

This user must also own the server process.

The database cluster will be initialized with locale “C.UTF-8”.

The default database encoding has accordingly been set to “UTF8”.

The default text search configuration will be set to “english”.

...<truncated>...

As you can see in the below code block, during the package installation on Ubuntu, the process also initializes a new database cluster, so at the end, you will end with an already up-and-running PostgreSQL. Ubuntu includes the **pg_lsclusters** wrapper. We can list the initialized clusters and their status with it:

```
pg@ubuntu-focal:~$ pg_lsclusters
Ver Cluster Port Status Owner      Data directory          Log file
14  main     5432 online postgres /var/lib/postgresql/14/main /var/log/
                           postgresql/postgresql-14-main.log
```

Once the postgres service is up, validate the data directory, create a test DB, and log in to the database as mentioned in the Source Code Installation steps.

Conclusion

In this chapter, we have tried to cover different installation methodologies, mainly focusing on Source Code Installation and Binary Installation. These methods are widely used and valid for any modern Linux Operating System. You might have to adjust a few details if you work on a different distribution from Ubuntu, which was covered in the examples.

In the next chapter, we will learn about other modern ways to get PostgreSQL, different from the usual approach of installing it on the Operating System. We will review the containers, Kubernetes, and Cloud alternatives for getting PostgreSQL.

Bibliography

- Source Code Install: <https://www.postgresql.org/docs/current/installation.html>
- PostgreSQL on Ubuntu: <https://nextgentips.com/2022/05/23/how-to-install-postgresql-14-on-ubuntu-20-04-from-the-source/?amp=1>
- PostgreSQL on Linux: <https://www.thegeekstuff.com/2009/04/linux-postgresql-install-and-configure-from-source/>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 3

Modern Options to get PostgreSQL

Introduction

As we see in the previous chapter, there are two main ways to install PostgreSQL on modern Linux Operating Systems, but there are also other ways to get it. Modern technologies and new platforms to deploy applications have made it possible to get PostgreSQL ready with just a few lines of code or even “easier” with just some clicks.

Structure

In this chapter, we will walk through some currently existing options to deploy new applications and how to get PostgreSQL in containers, Kubernetes, and some DBaaS/Cloud vendors.

The topics to be covered:

- Other Ways to get PostgreSQL
- Getting PostgreSQL on modern systems

Objectives

You will learn current options for the deployment and configuration of infrastructure and applications, and we will show you some options to get PostgreSQL different from the *standard* installation. By the end of the chapter, you become familiar with the new methods available in the market, so you might consider them when planning your new PostgreSQL adventure.

Other ways to get PostgreSQL

Nowadays, technology has moved from the “old school” style of having bare metal machines where to install the Operating System, all the required libraries, the servers or services, and finally the applications to a modern way to do all this with just a few lines of code. This concept might be out of the scope of most of the newer people in the IT world. So let us take a look at what this is about.

On-premise, virtualization, containers, and cloud

How the systems and applications are created, deployed, and delivered to the final users has changed significantly in the last ten years. The options the companies now have are much more, and also the flexibility they offer. To illustrate, consider a fictional company whose service design consists of the following basic components and how they have adapted it.

- A web page running on a web server.
- A file server where some pictures and multimedia files are stored.
- A database layer as a backend.

On-premise

When we talk about on-premise (on-prem), we talk about the “old fashion way” to get the service working. This is to get bare-metal servers hosted on a Data Center and configure each one with the required software and any other hardware or configuration it needs. For example, disk storage and all the configuration for networking.

Even in the simple example of our imaginary company, the setup of these “just three” boxes will require the participation of a vast group of people. To adequately the physical site, energize it, prepare the network infrastructure, rack the servers, attach the storage, install the software, configure it, and deploy all the apps. The hard work does not finish once we get all these pieces in their correct spot; we also need to consider all the future changes we would need to apply to our application or the database model. The truth is multiple mistakes can occur and delay the deployment or, even worse, take the service down. *Figure 3.1* illustrates how the on-prem deployment looks.

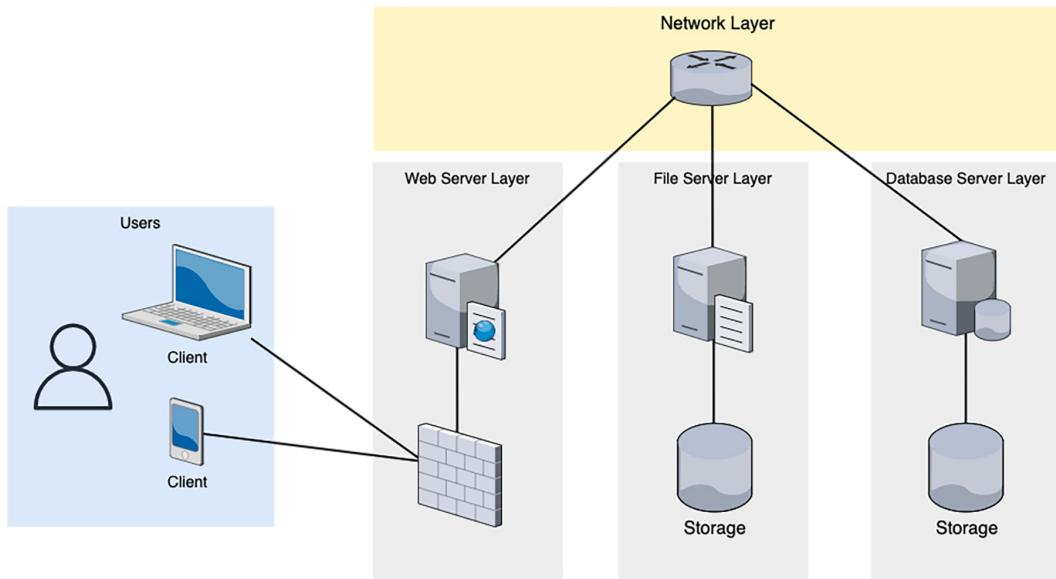


Figure 3.1: On-prem deployment

Virtualization

Differently from the on-prem case, when we work in virtual environments, we can adequate the physical machine (the host) to execute a particular type of software, usually called a hypervisor, able to run many instances of Operating Systems (the guests) called **Virtual Machines (VM)**. *Figure 3.2* shows the components for a virtualized solution:

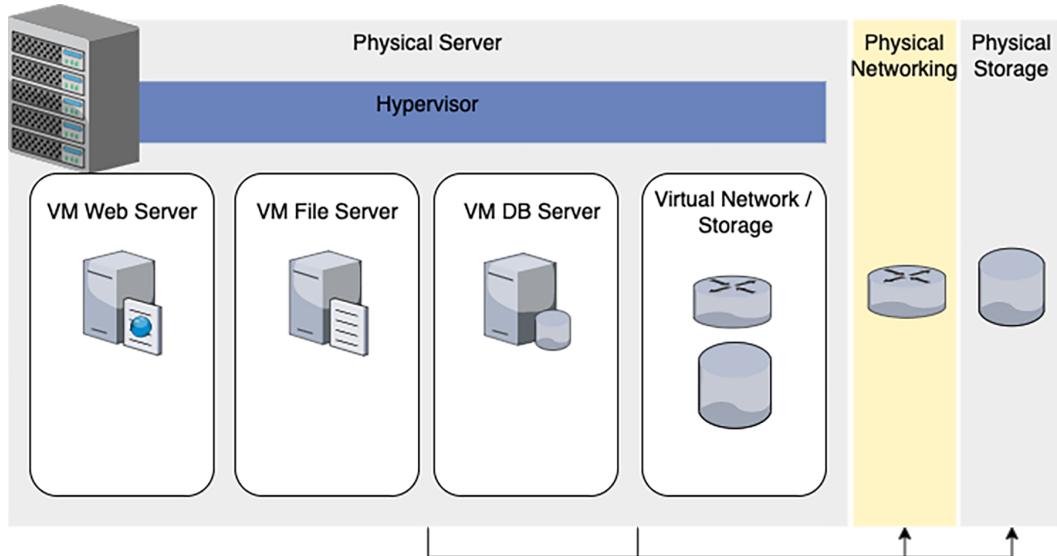


Figure 3.2: Virtualized deployment

In a virtual environment, all the components can be added and configured much easier than in on-prem, so we can adjust the storage we need and resize it online, we can add new VMs and adequate the network quickly, and also we can configure a cluster on a separate physical machine so we can move the VM accordingly the workload, or to bring some tolerance to failures in case any of the physical boxes breaks.

Containers

Virtualization brings multiple benefits compared with the on-prem approach; now, the infrastructure provision is faster, and developers can dedicate more energy to programming. However, the deployment of new applications or new versions still demands specialized people to get involved. Because applying changes to a virtual environment requires a good understanding of the physical layer and the software to host the VMs.

Thinking about how to improve these stages, the containers got into the scene. Unlike the VMs, where every instance is a functional Operating System running in a virtual layer, the containers are “processes” running on top of the host Operating System, sharing the kernel and resources. These processes do not virtualize the hardware but the operating system, and each includes all the dependencies it might need to run and serve a specific application. *Figure 3.3* illustrates the different layers when deploying the solution on containers:

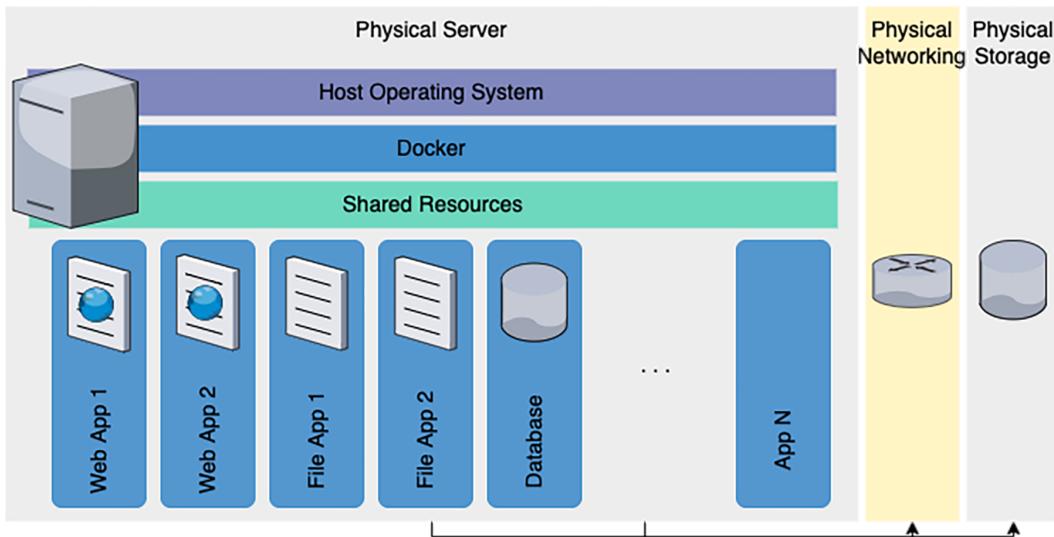


Figure 3.3: Containers

Containers are much lighter, more portable, and fast than VMs. They do not need to carry out a complete guest operating system per instance but leverage and consume the features from the host OS directly. Also, since there is no need for hypervisor software, the containers can run on any host OS.

Mention apart is required for Kubernetes, an open-source system for automating deployment, scaling, and managing containerized applications. (Reference: Kubernetes) It is a powerful tool; when containers enter the enterprise tiers, the size of the systems and the number of required containers scale up to thousands, and controlling them can be difficult. Here Kubernetes provides all the required tools and interfaces to handle that deployment size.

The Cloud

Finally, we have The Cloud. We could say it is the result of summarizing all the previous enhancements. Big companies with enough resources have turned some of their Data Centers into “the cloud,” running many physical servers, which might have hypervisors to spin up VMs or platforms to deploy new containers quickly. As you can imagine, these “ecosystems” can be huge, so they run Kubernetes or similar solutions behind the scene. In *Figure 3.4* you can see how the solution might look in the Cloud:

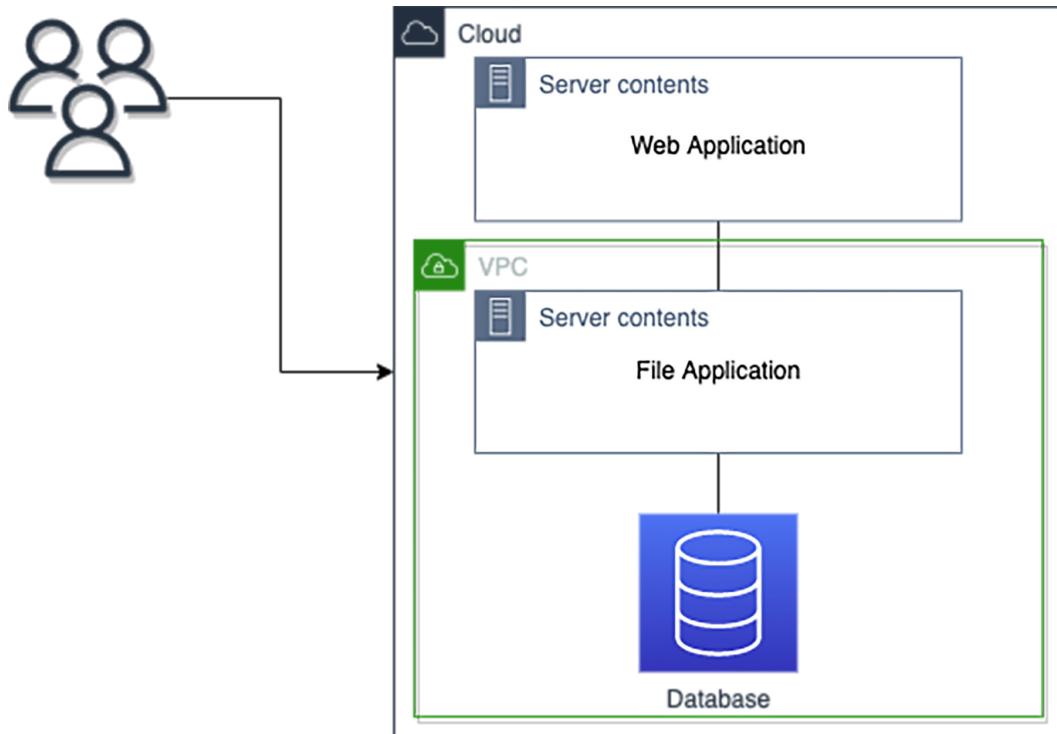


Figure 3.4: The Cloud

Just to name the most popular cloud vendors, we have **Amazon Web Services (AWS)**, **Google Cloud Platform (GCP)**, **Microsoft Azure**, **Digital Ocean (DO)**, and others. Now, they sell their solution as just picking what you need and paying for what you use. This enables an enormous number of people to get the infrastructure they need to run their solutions quicker than ever before, even if they do not have deep knowledge about servers, networking, operating systems, and so on.

This really might sound great, and actually, it is. But remember, “All that glitters is not gold.” Playing the game of the cloud under the vendor’s rules can drive your business to the vendor lock-in. This means that the more you adapt your system to a specific cloud, the more challenging it is to get out of it later.

Getting PostgreSQL on modern systems

By this time, you have already learned how to get PostgreSQL up and running in the “traditional” way, installing from the source or compiling the binaries yourself. Both methods apply for the on-prem, or the VMs approaches, where in the end, you will work directly on an operating system.

As we saw in this chapter, there are *modern* ways to get the infrastructure, and since PostgreSQL is one of the most used open-source database systems, you have some options to get it deployed on these newer platforms. We will review how to get PostgreSQL up and running for Docker and Kubernetes (containers) and some existing offers in the cloud.

PostgreSQL on Docker

Docker is one of the most popular container engines. It can run on Linux and Windows operating systems. And just as we saw before, you can easily and quickly get many different applications running on containers. In a Docker environment, you can get these applications from *images* which are files containing all the definitions and requirements for a given application. Then, when you *execute* this image on Docker, it becomes a *container*.

Docker supports a shared images repository, where people can get and share images for various applications, called Docker Hub. You can find images with different operating systems, web servers, databases, development frameworks, and so on. You can access the Docker Hub at <https://hub.docker.com/>.

To get the latest stable version of postgres on Docker, you can just run the next in the system where you already have the Docker Engine running.

```
docker pull postgres
```

This will download the image in the system if it does not exist. The same Docker Hub page for the PostgreSQL image provides the command to get a running container with the downloaded image; we just need to execute the following command.

```
docker run --name my-postgres \
-e POSTGRES_PASSWORD=mysecretpassword \
-d postgres
```

We can verify our container is running with the next command:

```
docker ps
```

The *Figure 3.5* shows the output of the previous command:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
26671230146e	postgres	"docker-entrypoint.s..."	7 seconds ago	Up 6 seconds	5432/tcp	my-postgres
bash-3.2\$						

Figure 3.5: Docker postgres container.

You got a new PostgreSQL instance up and running; now you can connect the next way:

```
docker exec -it my-postgres \
    psql -h localhost -U postgres postgres
```

Refer *Figure 3.6* the result of connecting to PostgreSQL with the previous command:

```
bash-3.2$ docker exec -it my-postgres psql -h localhost -U postgres postgres
psql (15.1 (Debian 15.1-1.pgdg110+1))
Type "help" for help.

postgres=# select version();
              version
-----
 PostgreSQL 15.1 (Debian 15.1-1.pgdg110+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 10.2.1-6) 10.2.1 20210110, 64-bit
(1 row)
```

Figure 3.6: Connected to Docker postgres container instance

PostgreSQL on Kubernetes

Kubernetes by itself can fill an entire book. It is a potent tool for handling and orchestrating large deployments of containers called *pods* in the Kubernetes argot. For the scope of this chapter section, we will focus on how to get PostgreSQL deployed.

As you can imagine, deploying PostgreSQL in a Kubernetes cluster requires you already have an existing cluster. Getting a running Kubernetes cluster is beyond the scope of this book, but we recommend looking at the `minikube` tool. It is a perfect way to start trying Kubernetes. For detailed steps, go to <https://minikube.sigs.k8s.io/docs/start/>.

Once you got your Kubernetes cluster and the Kubernetes command-line tool `kubectl`, you can follow the next steps to create the required set of resources for the deployment.

1. **ConfigMap**. Store key-value data pairs for the database details, user, password, and database name.
2. **PersistentVolume (PV)** and **PersistentVolumeClaim (PVC)**. These components provide a storage layer to store and persist the database data, even if the containers or pods are deleted. Imagine the equivalent of attaching a disk that will be there even if you change your database installation.
3. A PostgreSQL deployment. Here you will define how Kubernetes should provide the pods for postgres and what PV and PVC use.
4. The PostgreSQL service. This one will define how to expose the postgres deployment so that the users or applications can connect to it.

All the configurations for Kubernetes are made through YAML files. You need a different file for each one of the previously listed resources. The following are examples that can be used to get a PostgreSQL deployment.

ConfigMap

File: postgres-config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: postgres-config
  labels:
    app: postgres
data:
  POSTGRES_DB: postgresdb
  POSTGRES_USER: postgres
  POSTGRES_PASSWORD: secret
```

PersistentVolume (PV) and PersistentVolumeClaim (PVC)

File: postgres-pvc-pv.yaml

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: postgres-volume # PV's name
  labels:
    type: local # Sets PV's type to Local
    app: postgres
spec:
  storageClassName: manual
  capacity:
```

```
storage: 10Gi # PV Volume

accessModes:
  - ReadWriteMany

hostPath:
  path: "/mnt/pgdata" # Local path in the host

---
kind: PersistentVolumeClaim

apiVersion: v1

metadata:
  name: postgres-volume-claim # Name of PVC

  labels:
    app: postgres

spec:
  storageClassName: manual

  accessModes:
    - ReadWriteMany # Read and write access

  resources:
    requests:
      storage: 10Gi # Volume size
```

PostgreSQL deployments

File: `postgres-deployment.yaml`

```
apiVersion: apps/v1

kind: Deployment

metadata:
  name: postgres # Deployment name

spec:
  replicas: 1
```

```
selector:
  matchLabels:
    app: postgres
template:
  metadata:
    labels:
      app: postgres
spec:
  containers:
    - name: postgres
      image: postgres:14.5 # Image
      imagePullPolicy: "IfNotPresent"
      ports:
        - containerPort: 5432 # Exposes container port
  envFrom:
    - configMapRef:
        name: postgres-config # Using the created secrets
  volumeMounts:
    - mountPath: /var/lib/postgresql/data
      name: postgredb
  volumes:
    - name: postgredb
      persistentVolumeClaim:
        claimName: postgres-volume-claim
```

PostgreSQL service

File: `postgres-service.yaml`

```
apiVersion: v1
```

```
kind: Service

metadata:
  name: postgres # Service name

labels:
  app: postgres

spec:
  type: NodePort # Service type

  ports:
    - port: 5432 # Port to run the postgres application

  selector:
    app: postgres
```

With all the previous files correctly filled now, you can apply the configuration from each one to the Kubernetes cluster. The following are the commands in the correct order:

```
kubectl apply -f postgres-config.yaml
kubectl apply -f postgres-pvc-pv.yaml
kubectl apply -f postgres-deployment.yaml
kubectl apply -f postgres-service.yaml
```

You can verify the status of the deployment and service with the following command.

```
kubectl get all
```

Figure 3.7 illustrates the output when checking the deployed kubernetes components:

```
bash-3.2$ kubectl get all
NAME                           READY   STATUS    RESTARTS   AGE
pod/postgres-5cc787ddbc-5mbtm  1/1     Running   0          34s

NAME              TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)        AGE
service/kubernetes  ClusterIP   10.96.0.1    <none>           443/TCP       2m33s
service/postgres   NodePort    10.107.212.206  <none>           5432:30886/TCP 3s

NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/postgres  1/1     1           1           34s

NAME                DESIRED   CURRENT   READY   AGE
replicaset.apps/postgres-5cc787ddbc  1         1         1         34s
bash-3.2$
```

Figure 3.7: PostgreSQL Kubernetes deployment.

Now that you have all the resources deployed and running, you can try connecting to postgres using a command like the next.

```
kubectl exec -it <postgres-pod-name> -- psql -h localhost -U <user-name> --password -p 5432 <db-name>
```

Remember to replace the placeholders with the current values you used for the configuration and the name of the actual running pod. See in *Figure 3.8* how the command's output looks:



```
bash-3.2$ kubectl exec -it pod/postgres-5cc787ddbc-5mbtm -- psql -h localhost -U admin --password -p 5432 postgresdb
Password:
psql (14.5 (Debian 14.5-1.pgdg110+1))
Type "help" for help.

postgresdb=# select version() ;
              version
-----
 PostgreSQL 14.5 (Debian 14.5-1.pgdg110+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 10.2.1-6) 10.2.1 20210110, 64-bit
(1 row)

postgresdb=#
```

Figure 3.8: Connected to Kubernetes postgres deployment

PostgreSQL on The Cloud

Finally, another *modern* way to get PostgreSQL ready to work is getting it from one of the cloud vendors' options. As we learned in the previous section, the cloud consists of multiple services hosted in massive data centers ready to be consumed on-demand. You can easily go to the web page of your favorite cloud vendor, register and create an account and start using services just out of the box. Some vendors offer free trial periods so that you can test some services.

When choosing the cloud, you need to remember that they group the services into tiers or classes, each with different characteristics and prices. Usually, the database services are grouped into General Purpose, Memory Optimized, and Processor Optimized. Then each group has different classes, depending on the resources: (1) Number of CPUs, (2) RAM Memory, (3) Disk bandwidth (IOPS, Input/Output operations per second), and (4) Network performance (Gbps, Gigabits per second). The amount they will charge you depends on the type and class of database instance you choose.

Even when this looks easy, remember that you must protect your infrastructure in a production environment, so enforcing security is very important here. This book is not intended to go deep on these topics, but it is also advisable you can review them.

The next example shows how you can get a PostgreSQL instance from AWS; they call their service **Relational Database Service (RDS)**. This example was referenced from AWS Documentation. (Reference: RDS)

The steps can be used to create a new PostgreSQL RDS instance:

1. **Access to the AWS Console.** You can use an existing account or create a new one as shown in the following figure:

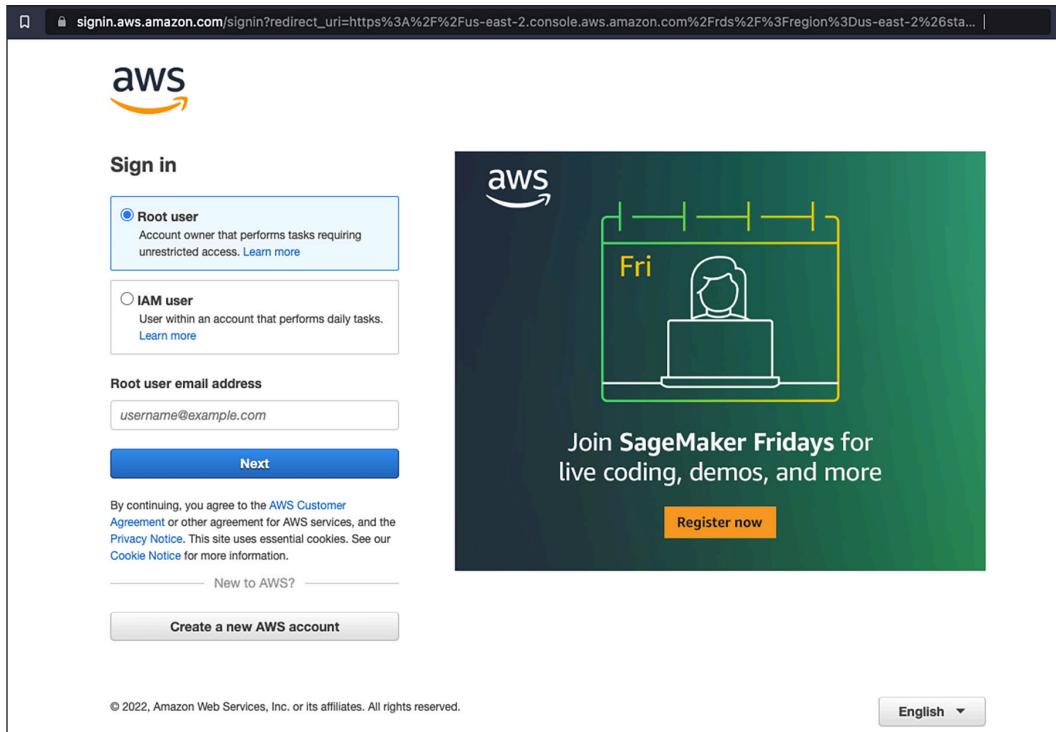


Figure 3.9: AWS Console

2. From the AWS Services, find the RDS option. (*Figure 3.10*):

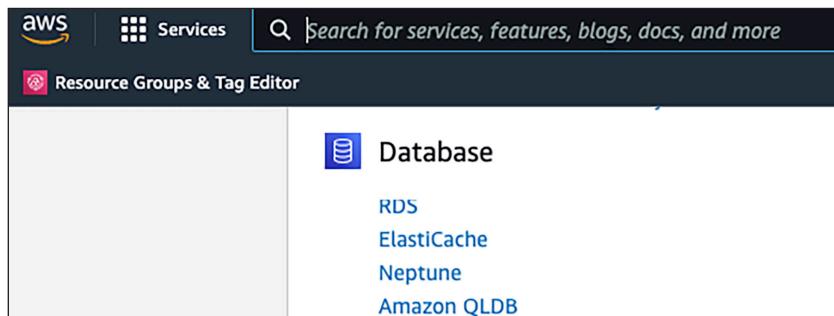


Figure 3.10: AWS Management Console

3. Click the “**Create database**” button as shown in *Figure 3.11*:

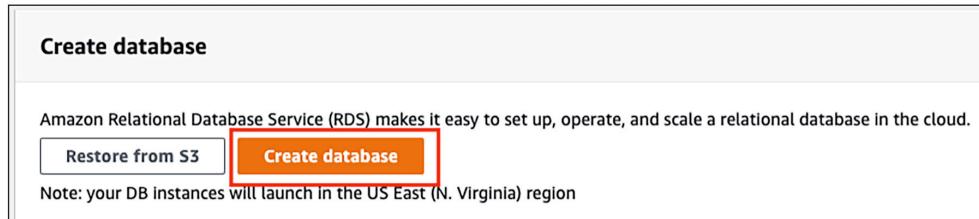


Figure 3.11: Amazon RDS Create database

4. Select PostgreSQL as the database engine as illustrated in *Figure 3.12*:

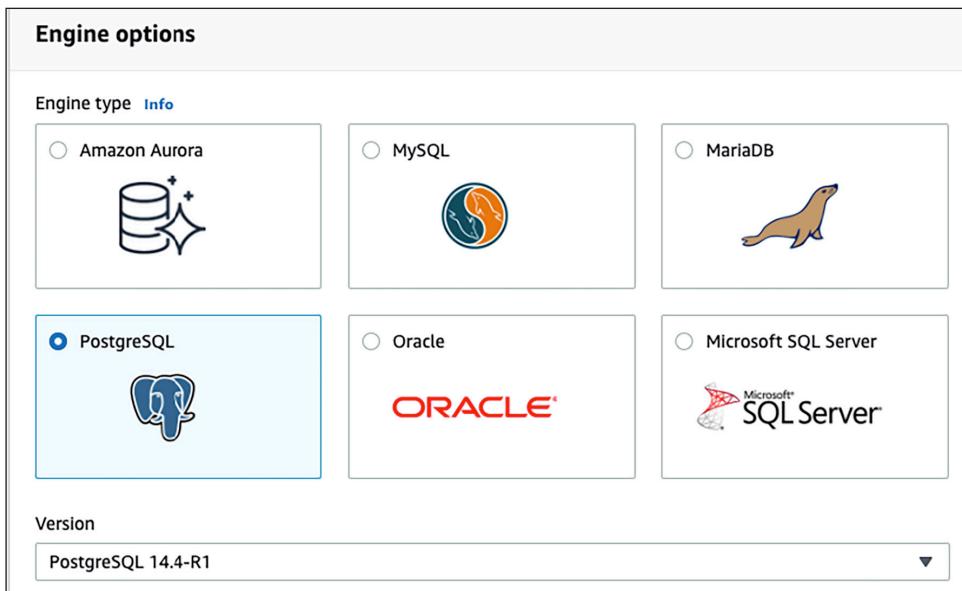


Figure 3.12: Amazon RDS Engine

5. Fill in the general details of the new RDS database and *master* user as shown in *Figure 3.13*:

The screenshot shows the 'Settings' section. The 'DB instance identifier' field contains the value 'rds-postgresql-10minTutorial', which is highlighted with a red box. A note below the field states: 'The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens (1 to 15 for SQL Server). First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.'

Figure 3.13: Amazon RDS Settings

6. Now you need to select the database class and resources as shown in the following figure:

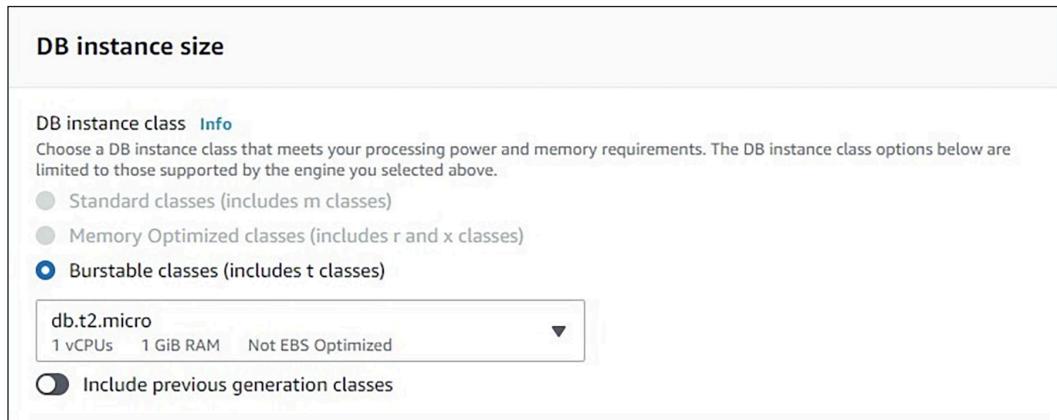


Figure 3.14: Amazon RDS Instance class

7. You can leave the Connectivity details in their default values, but be sure you select the “**Publicly accessible**” to be able to connect from outside the AWS Cloud as depicted in *Figure 3.15*:

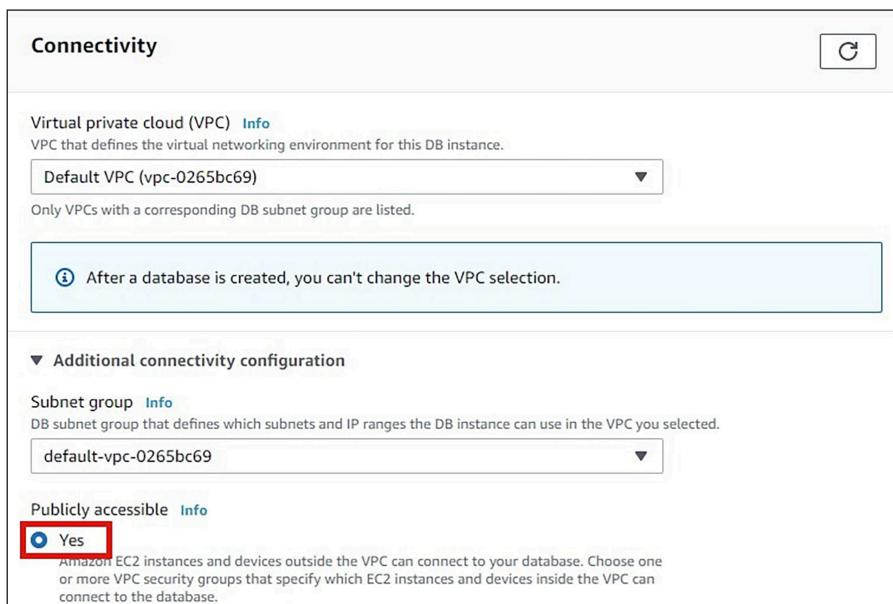


Figure 3.15: Amazon RDS Connectivity

8. You will get a window showing the estimated monthly costs before creating the database. If looks OK, click the “**Create database**” button as illustrated in *Figure 3.16*:

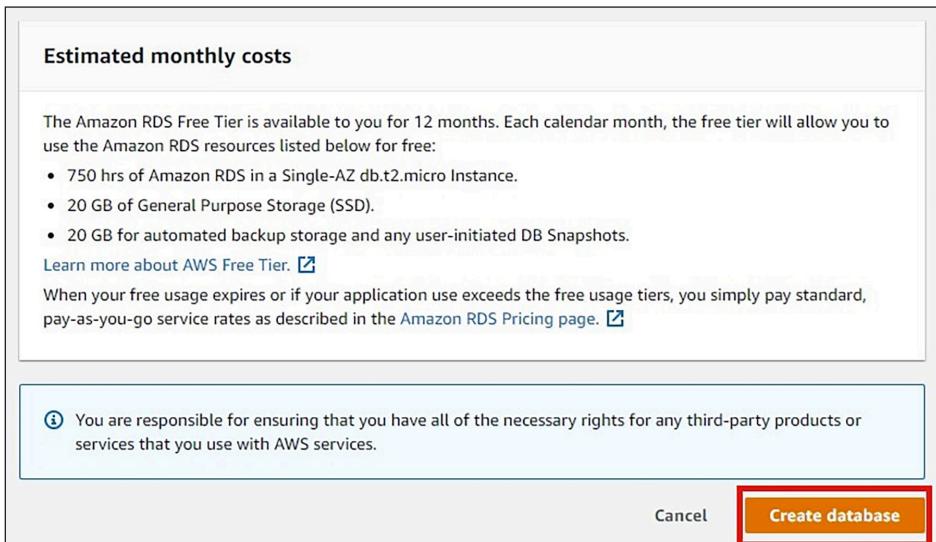


Figure 3.16: Amazon RDS Pricing

- After some minutes, the new database will show up in the “**Databases**” section within the RDS service. You can get the “**Endpoint**” and “**Port**”, you need these details to configure a connection from an external client as shown in following figure:

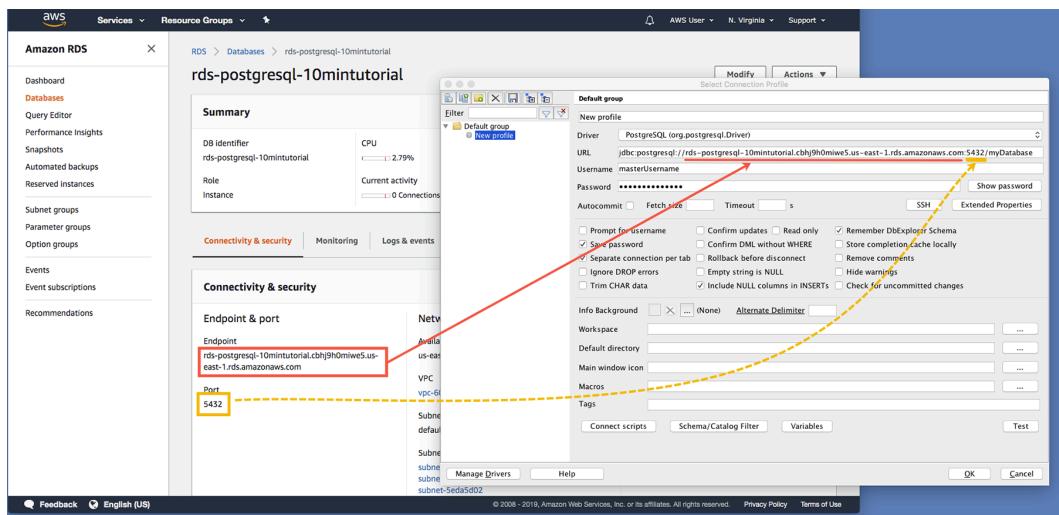


Figure 3.17: Connecting to RDS Database

Conclusion

This chapter covered some alternative ways to get PostgreSQL in modern ways. Different from the “classic” installation methods, some new technologies bring more options to get PostgreSQL ready quickly and easily.

Now you know about the existence of containers, Kubernetes, and the Cloud. These technologies open the door to more flexible and programmatic options to deploy your environments.

In the next chapter, we are going to explore global objects in PostgreSQL like the creation of Databases, Tablespaces, Roles, Users & Groups.

Bibliography

- Kubernetes: <https://kubernetes.io/>
- RDS: <https://aws.amazon.com/getting-started/hands-on/create-connect-postgresql-db/>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 4

Global Objects in PostgreSQL

Introduction

PostgreSQL has some objects created across the database cluster installation and are not tied to any individual database. These objects can operate or be shared on multiple databases and database objects (tables, indexes, functions, and so on.) Knowing them and being familiar is crucial to take advantage of the flexibility and customization level we can get in our PostgreSQL systems.

Structure

During this chapter, we will review the main global objects we can create and are divided into the following topics:

- Users/Groups/Roles
- TablespacesDatabases

Objectives

In this chapter, you will learn about the global objects available in PostgreSQL, what they are, how to create and delete them, and some tips and advice aligned to the best practices for a secure and functional setup.

Users/Groups/Roles

You should have heard about database users or groups before, right? When talking about databases, no matter the product or technology, some entities cannot be missed because through them is how we can access our data; these are the database *users*. Also, it is prevalent; since it is a best practice concept, your database has some *groups*; they exist to organize the privileges and some properties and make their management easier and more efficient. But what about *roles*? Let us see what all this means in the PostgreSQL world.

As stated before, users and groups are two concepts that are widely used in database systems. Before version 8.1 of PostgreSQL, these exist as two separate entities. But after that, and up to the latest available version (15), the *role* entity represents the concepts of *user* and *group*.

So, when we talk about a role in PostgreSQL, it might mean a database user or a set of users (group), depending on how it was set up. Then the roles can be used to authenticate against the database and authorize access to specific objects. In this way, we can control who can access what.

Roles can own different database objects and hold multiple privileges, so they can be assigned to another role or even more to grant *membership* to other **roles** so that the members can get the same privileges. *Figure 4.1* shows how to list the existing roles using the **psql** meta-command **\du** operator:

Role name	Attributes	List of roles	Member of
demouser	Create role, Create DB	+	{}
percona	Create role, Create DB	+	{rds_superuser}
postgres	Password valid until infinity	+	+ {rds_superuser}
rds_ad	Cannot login		{}
rds_iam	Cannot login		{}
rds_password	Cannot login		{}
rds_replication	Cannot login		{}
rds_superuser	Cannot login		{pg_monitor,pg_signal_backend,rds_replication,rds_password}
rdsadmin	Superuser, Create role, Create DB, Replication, Bypass RLS+	+	{}
rdsrepladmin	Password valid until infinity		{}
rep	No inheritance, Cannot login, Replication		{}

Figure 4.1: Listing roles.

As stated in the introduction of this chapter, the roles are part of the global objects. This means they are created across a database cluster installation (and not per individual database).

When you install a new PostgreSQL instance and initialize it with the **initdb** command, a new role is created with the **superuser** attribute. This role has all the

possible privileges, so it can be used to start adding new roles, set their privileges, and create databases. This “admin” user is named as the operating system user who initialized PostgreSQL, usually called **postgres**. We need to protect the credentials of this user since the superuser role is the most powerful, and any uncontrolled access to it can damage the system.

To create a new role, we can use the following syntax. Optionally we can add some options or attributes for our new user, for example, **LOGIN** to set it ready to connect and **PASSWORD** to set a new password for it:

```
CREATE ROLE demouser LOGIN PASSWORD 'supersecurepass';
```

If you already have access to a psql prompt, you can review the available options for the **CREATE ROLE** with the next **psql** meta-command. *Figure 4.2* shows the output:

```
\h CREATE ROLE
```

```
postgres=> \h CREATE ROLE
Command: CREATE ROLE
Description: define a new database role
Syntax:
CREATE ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

    SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid

URL: https://www.postgresql.org/docs/14/sql-createrole.html
```

```
postgres=> ■
```

Figure 4.2: Create role options.

A good practice when creating roles is to add a role with the **CREATEROLE** and **CREATEDB** attributes but not the **SUPERUSER**. And use this role to add any other role and database you need to create.

On the other hand, if you need to drop a role, you just need to use the opposite SQL command:

```
DROP ROLE demouser;
```

As we saw in *Figure 4.1*, you can list the existing roles with the **psql** meta-command **\du**. If you are connected to the database with another client than **psql**, you can list the roles querying the **pg_roles** view.

```
SELECT * FROM pg_roles ;
```

We have reviewed whether it is possible to create roles to act as database users, including the **LOGIN** attribute. These roles or users can start a new session in the database. However, we must remember that a role can act as a group to hold multiple privileges and then extend their membership to other roles to get the privileges. This is the recommended way to set up privileges, granting them to groups rather than users and then just “adding” the users to the groups.

When creating a role to act as a group, we usually do not want or need it to start a session against the database, so the **LOGIN** attribute and even a **PASSWORD** would not be expected. To create the role then, we just need the following SQL command.

```
CREATE ROLE admins;
```

Then we can grant some privileges to the group, for example:

```
GRANT SELECT ON public.table1 TO admins;
```

```
GRANT SELECT ON public.table2 TO admins;
```

And finally, “add” one or more users to the group by granting the role to them.

```
GRANT admins TO demouser, testuser, produser;
```

After this, all of our users **demouser**, **testuser**, and **produser** will be able to **SELECT** from **public.table1** and **public.table2** tables.

It is possible to create some hierarchy control using the **INHERIT** and **NOINHERIT** options when creating a role. By default, all the roles can inherit the privileges from any other granted role, but if you specify **NOINHERIT** when you create the role, then this role won’t get the privileges from any granted role so that you can design some control. Consider the next:

```

CREATE ROLE sampleuser LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE reader NOINHERIT;
GRANT admin TO sampleuser;
GRANT reader TO admin;

```

Any privilege you grant to the **admin** role will be accessible by **sampleuser** since it inherits. However, even when **sampleuser** is also an indirect member of **reader**, it won't have any privilege granted to **reader**, because the membership is made via **admin**, and it doesn't inherit.

Finally, we can review a list of special roles called predefined roles. There get created when the PostgreSQL system is initialized. These roles group some specific attributes or privileges and are named accordingly to be identified depending on their purpose. They can facilitate many management tasks related to user control.

The next can be consulted in the PostgreSQL community documentation. (Reference: Database Roles) It is important to review this list on every release since the attributes for the roles might change. *Table 4.1* presents the predefined roles for PostgreSQL 14:

Role	Allowed Access
<code>pg_read_all_data</code>	Allows to read all data from tables, views, and sequences, as if having <code>SELECT</code> rights on those objects, and <code>USAGE</code> rights on all schemas.
<code>pg_write_all_data</code>	Allows to write all data in tables, views, or sequences, as if having <code>INSERT</code> , <code>UPDATE</code> , and <code>DELETE</code> rights on those objects, and <code>USAGE</code> rights on all schemas..
<code>pg_read_all_settings</code>	Allows to read all configuration variables, including those only visible to superusers.
<code>pg_read_all_stats</code>	Allows to read all <code>pg_stat_*</code> views and use various statistics related extensions.
<code>pg_monitor</code>	Allows to read/execute various monitoring views and functions. This is the best option to setup a monitoring application user.
<code>pg_signal_backend</code>	Allows to send a signal to another backend to cancel a query or terminate its session.
<code>pg_read_server_files</code>	Allow reading files from any location the database can access on the server.

Role	Allowed Access
<code>pg_write_server_files</code>	Allow writing to files in any location the database can access on the server.
<code>pg_execute_server_program</code>	Allow executing programs on the database server as the user the database runs.

Table 4.1: Predefined roles in PostgreSQL 14

Tablespaces

Now, let us study another import global object called tablespaces. A tablespace in the PostgreSQL world is just a path in the Operating System to tell the database system where to store the database objects, such as tables and indexes.

When we initialize a new PostgreSQL database cluster, two default tablespaces are created on a relative path to the Data Directory, the `pg_global`, and the `pg_default`. The first one is used to store all the catalog system data, and the second one is the default tablespace to store all the database data for the template0 and template1 databases, so if we do not specify anything different when creating a new database, this property will be set the same as the template databases. We will learn more about the template databases in the next section of this chapter.

To create a new tablespace, we need the path in the local filesystem to exist and be owned by the same user running the `postgres` instance, usually the operating system user `postgres`. If these conditions are covered, a database user with the superuser attribute can create a tablespace. *Figure 4.3* shows the options to create it:

```
postgres=# \h CREATE TABLESPACE
Command: CREATE TABLESPACE
Description: define a new tablespace
Syntax:
CREATE TABLESPACE tablespace_name
[ OWNER { new_owner | CURRENT_ROLE | CURRENT_USER | SESSION_USER } ]
LOCATION 'directory'
[ WITH ( tablespace_option = value [, ... ] ) ]
```

URL: <https://www.postgresql.org/docs/14/sql-createtablespace.html>

Figure 4.3: Create tablespace options

Consider the next example:

```
CREATE TABLESPACE newspace LOCATION '/ssd1/postgres/newspace';
```

Here, we will add a new tablespace called **newspace** and all the data for the database objects we create using the target tablespace as **newspace** will write in the physical operating system path: **/ssd1/postgres/newspace**.

We can list the existing tablespaces with the psql meta-command **\db** as shown in *Figure 4.4*, or querying the **pg_tablespace** view:

List of tablespaces							
Name	Owner	Location	Access privileges	Options	Size	Description	
pg_default	postgres				25 MB		
pg_global	postgres				560 kB		
(2 rows)							

Figure 4.4: Listing existing tablespaces

Tablespaces can be used mainly for two purposes: (1) “extend” the capacity of the disk where the data directory was created by adding data to a separate disk device, (2) to improve the performance of some operations and leverage the capacities of different storage devices.

Regarding the second option, consider whether you have a heavily accessed index; you might improve performance by creating (or rebuilding) that index on a separate tablespace created on a solid-state disk. On the other hand, if you have a table with historical data rarely accessed, you can move it to a tablespace on a cheaper mechanical storage disk.

Databases

To understand the relation between instance and database, let us try to understand the instance first.

An instance consists of port, data area (or data directory) and it contains one service. One server can contain one or more database instances provided all has separate port and separate path for data directory. A database cluster contains multiple databases, users/roles having multiple table spaces. One postgres instance always manages the data of exactly one database cluster. (Reference: Server Applications)

As shown in *Figure 4.5*, database is part of **instance/cluster** and each **DB Instance/DB Cluster** contains multiple databases.

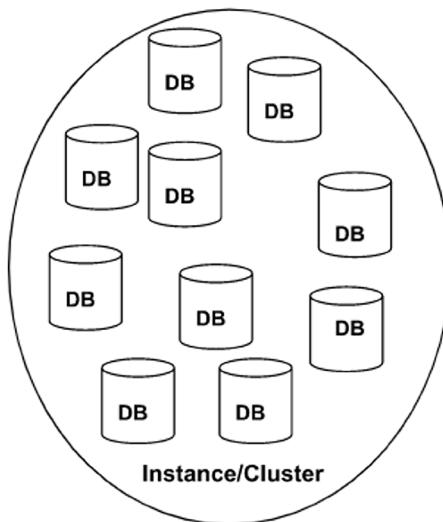


Figure 4.5: Databases inside the DB Instance

A database is a collection of multiple database schemas which in turn contains database objects like tables, views, packages, functions and what not. Database is part of instance and comes at the top of the hierarchy of postgresql objects.

Let's try to understand the same using an example - Imagine the house having multiple rooms and each room contain multiple ward-robies. Each wardrobe has separate shelves, drawers, and lockers. House is an instance and each room can be compared to the database inside that instance. Each wardrobe is like the database schema containing the tables, functions and other database objects.

House | Rooms | Ward-robies | Shelves, Drawers, Lockers

Instance | Database | Schemas | DB Objects like tables, functions and so on

Once **postgres** service is up and running, databases can be created using multiple ways. As shown in *Figure 4.6*, by default 3 databases are already created in the fresh installation/instance of the PostgreSQL. **postgres** is one of the default databases along with **template0** and **template1**. We will discuss more about **template0** and **template1** later in this chapter.

List of databases						
Name	Owner	Encoding	Collate	Ctype	Access privileges	
postgres	postgres	UTF8	English_United States.1252	English_United States.1252	=c/postgres	+
template0	postgres	UTF8	English_United States.1252	English_United States.1252	postgres=CTc/postgres	
template1	postgres	UTF8	English_United States.1252	English_United States.1252	=c/postgres	+
(3 rows)					postgres=CTc/postgres	

Figure 4.6: Default PostgreSQL Databases

CREATE DATABASE command

As the name suggests **CREATE DATABASE** command is used to create the database. It is similar to database creation commands in most other RDBMS with almost similar attributes. As shown in *Figure 4.7*, database creation contains the following attributes which is shown using the **psql** meta-command **\h** as discussed previously:

```
postgres=# \h CREATE DATABASE
Command: CREATE DATABASE
Description: create a new database
Syntax:
CREATE DATABASE name
    [ [ WITH ] [ OWNER [=] user_name ]
        [ TEMPLATE [=] template ]
        [ ENCODING [=] encoding ]
        [ LOCALE [=] locale ]
        [ LC_COLLATE [=] lc_collate ]
        [ LC_CTYPE [=] lc_ctype ]
        [ TABLESPACE [=] tablespace_name ]
        [ ALLOW_CONNECTIONS [=] allowconn ]
        [ CONNECTION LIMIT [=] connlimit ]
        [ IS_TEMPLATE [=] istemplate ] ]
URL: https://www.postgresql.org/docs/14/sql-createdatabase.html
postgres=#
```

Figure 4.7: CREATE DATABASE command

Simply Database can be created using the following syntax in case you do not want to use any other attribute:

```
CREATE DATABASE <database_name>;
```

Figure 4.8 shows database creation command which will create a database named **demo**.

```
postgres=# create database demo;
CREATE DATABASE
postgres=#

```

Figure 4.8: CREATE DATABASE example

All the other attributes are self-understood as described in `\h CREATE DATABASE` command. **TEMPLATE** option in the database creation indicates that default template databases can be used to create the database. Whatever objects we create in template0 & template1 databases and mention them in **TEMPLATE** attribute than those specifics objects will already be created in newly created fresh database.

createdb program

Createdb is a wraparound program for **CREATE DATABASE SQL** command and works exactly the same way. The only difference is that it gets executed directly from command prompt instead of SQL prompt as shown in the *Figure 4.9:*

```
PS C:\Program Files\PostgreSQL\14\bin> .\createdb.exe -U postgres -O postgres -p 5433 testing
Password:
PS C:\Program Files\PostgreSQL\14\bin> .\psql.exe -U postgres -p 5433
Password for user postgres:
psql (14.5)
WARNING: Console code page (437) differs from Windows code page (1252)
          8-bit characters might not work correctly. See psql reference
          page "Notes for Windows users" for details.
Type "help" for help.

postgres=# \l
              List of databases
   Name    | Owner | Encoding |           Collate           |           Ctype           | Access privileges
---+-----+-----+-----+-----+-----+-----+-----+
demo | postgres | UTF8 | English_United States.1252 | English_United States.1252 | 
postgres | postgres | UTF8 | English_United States.1252 | English_United States.1252 | 
template0 | postgres | UTF8 | English_United States.1252 | English_United States.1252 | =c/postgres      +
                                                               postgres=CTc/postgres
template1 | postgres | UTF8 | English_United States.1252 | English_United States.1252 | =c/postgres      +
                                                               postgres=CTc/postgres
testing | postgres | UTF8 | English_United States.1252 | English_United States.1252 | 
(5 rows)
```

Figure 4.9: createdb program example

One can check the help using the OS command for help from command prompt. In the example shown in *Figure 4.10*, Windows Operating System has been used to get the details on the command:

```

PS C:\Program Files\PostgreSQL\14\bin> .\createdb.exe --help
createdb creates a PostgreSQL database.

Usage:
  createdb [OPTION]... [DBNAME] [DESCRIPTION]

Options:
  -D, --tablespace=TABLESPACE  default tablespace for the database
  -e, --echo                  show the commands being sent to the server
  -E, --encoding=ENCODING     encoding for the database
  -l, --locale=LOCALE          locale settings for the database
      --lc-collate=LOCALE       LC_COLLATE setting for the database
      --lc-ctype=LOCALE         LC_CTYPE setting for the database
  -O, --owner=OWNER            database user to own the new database
  -T, --template=TEMPLATE      template database to copy
  -V, --version                output version information, then exit
  -?, --help                   show this help, then exit

Connection options:
  -h, --host=HOSTNAME         database server host or socket directory
  -p, --port=PORT              database server port
  -U, --username=USERNAME     user name to connect as
  -w, --no-password           never prompt for password
  -W, --password               force password prompt
  --maintenance-db=DBNAME     alternate maintenance database

By default, a database with the same name as the current user is created.

Report bugs to <pgsql-bugs@lists.postgresql.org>.
PostgreSQL home page: <https://www.postgresql.org/>

```

Figure 4.10: createdb program help for Windows OS

Using pgAdmin Wizard

Let us now try to understand how to create database using the GUI tool from PostgreSQL Community which is known as pgAdmin.

Create Database option command by right clicking the Databases cluster in the treeview structure of pgAdmin which comes at the left side. Please follow the instructions given in the screen shots of this example where pgAdmin4 tool has been used.

Right Click the **Database** node in the PgAdmin Treeview as shown in *Figure 4.11*:

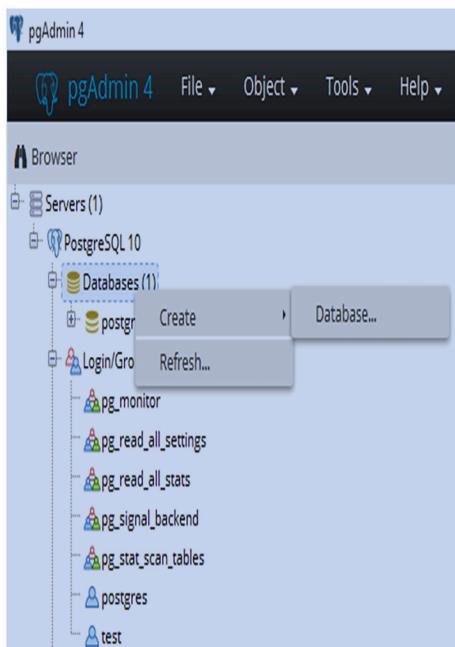


Figure 4.11: Treeview Structure for Database creation using pgAdmin4

On clicking **Create | Database** option , following screen will open up as show in *Figure 4.12*:

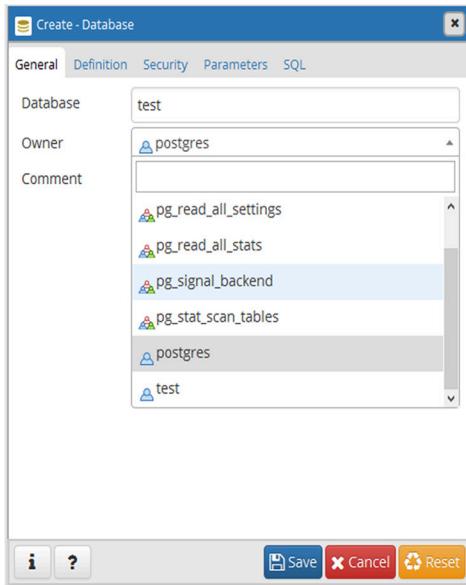


Figure 4.12: Giving meta data for Database creation using pgAdmin4

Fill the required details for **Database** name, Owner of the database along with putting comments (if any). It is a good practice to keep comments for each DB Component. (*Figure 4.13*):

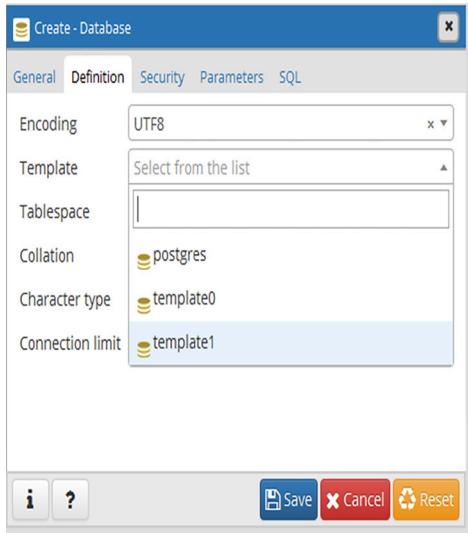


Figure 4.13: Entering Tablespace details for Database creation using pgAdmin4

In the **Definition** tab as shown in *Figure 4.13*, give tablespace name (if any) and normally we do not need to change anything else at this screen as default value are good enough. In case of any preferred option, select the values accordingly:

```

CREATE DATABASE test
WITH OWNER = test
ENCODING = 'UTF8'
CONNECTION LIMIT = -1;

COMMENT ON DATABASE test
IS 'test database';

```

Figure 4.14: Database creation script using pgAdmin4

Once all information are given for all tabs, SQL script get automatically gets generated in the last tab as shown in *Figure 4.14*.

Along with database creation, we have **ALTER DATABASE**, **DROP DATABASE** and **dropdb** are also available for managing the databases in PostgreSQL.

Conclusion

As we learned in this chapter, once you get PostgreSQL installed and running, there are some objects at the Global level. These exist across all the database cluster and can interact with the different databases we add.

It is important to remember that the roles and tablespaces can be used to improve our database system's security and take advantage of the physical storage layout and that a single DB cluster can host several different databases.

In the next chapter we are going to learn about the Architecture of PostgreSQL which will include more details on Memory Architecture, PostgreSQL Background Processes and Physical Architecture of PostgreSQL.

Bibliography

- Database Roles: <https://www.postgresql.org/docs/current/predefined-roles.html>
- Server Applications: <https://www.postgresql.org/docs/current/app-postgres.html>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 5

Architecture of PostgreSQL

Introduction

Like other modern **Relational Database Management Systems (RDBMS)**, PostgreSQL results from multiple components existing on top of an operating system layer. Each one is responsible for some specific aspects of the whole functionality. Understanding these different parts will prepare you for any technical interview or for debugging and resolving issues on your running systems.

Structure

In this chapter, we will learn about the three main components of the PostgreSQL architecture and review some details about their internal functionality.

- Memory architecture
- Background processes
- Physical files

Objectives

In the content of this chapter, we will review the design of the PostgreSQL architecture and how its different components work and interact. Every sub-section will focus on one main component and explain what is involved and how it works. At the end of the chapter, you will become familiar with the various memory areas, the processes responsible for the different tasks, and the physical file layout.

Memory architecture

We could say in a simplified way the purpose of PostgreSQL (and of any other RDBMS) *is to retrieve the queried data as fast as possible*. To achieve that goal, the Database system attempts to operate most of the data access and manipulations (READS and WRITES) directly in memory (RAM). Reading and writing data in memory is much faster than in any other media, such as disks.

This is why memory areas are very important when talking about databases. The design of PostgreSQL relies on multiple memory segments. They are grouped into two types: one is shared memory and other one is process memory. As you might imagine, the first group is for all those memory areas that can be accessed by all the processes, and their state is available all the time from different tasks. At the same time, process memory refers to the memory areas dedicated to the user or background processes and are used for specific tasks and operations from a single process.

Shared memory

The **shared memory** is allocated when the PostgreSQL instance is started and divided into multiple fixed-size sub-areas. *Figure 5.1* shows the sub-areas:

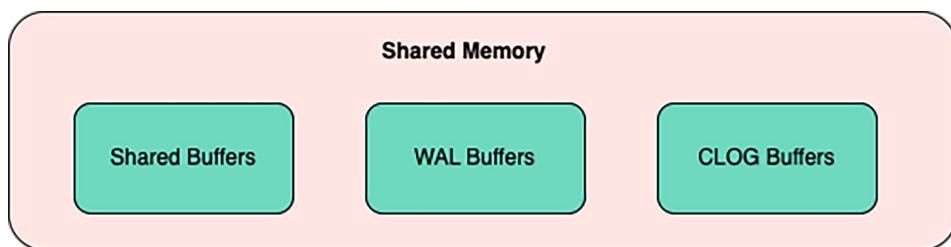


Figure 5.1: Shared Memory areas

Shared buffers

We might say the Shared Buffers sub-area is one of the most important since it is where the database data will be written and read as much as possible. In other Database technologies, the equivalent to this area is also known as the *database cache*. When a row (tuple in the postgres argot) is read the first time, it needs to be retrieved from disk, which is an expensive and slower task, but then the row is stored in this memory area, so any new query that might require it can get it quickly from here without going to the disk. Also, every time a row is added or modified, the writes are done here.

When a data page (or data block), which is the most elemental data unit (equivalent to 8KB for default), changes its contents in the Shared Buffer, then is marked as a “dirty page”. Subsequently, the dirty data is flushed to disk by some of the background processes, and it is persisted in the physical files called “data files”. We will get a deeper look at these later in this chapter.

At the configuration level, the area of the shared buffer is controlled by the parameter **shared_buffers**. Usually, 25% of the total RAM is set as the size of the shared buffer. Modifying this parameter required a PostgreSQL restart.

WAL buffers

The **Write Ahead Log (WAL)** is a transaction log mechanism used in PostgreSQL to keep a log of all the metadata related to the data changes. Hence, it is possible to redo the data operations in the case of data recovery.

Every time a COMMIT is executed, the changed data is stored in the Shared Buffers, as we saw before, and the transaction-related information is added to the WAL Buffers. Then, a separate background process will write the dirty data from this memory section into physical files known as WAL Files or Segments.

The WAL buffer area size is controlled via the **wal_buffers** parameter. By default, the size of this area is calculated as 1/32 of the **shared_buffers**. Changing the value requires a service restart.

CLOG buffers

CLOG stands for **Commit Log**, and the CLOG Buffers are the memory space dedicated to keeping the commit status of all the concurrent transactions in the Database system. This information is essential for the **Multi-version Concurrency Control (MVCC)** mechanism PostgreSQL uses to handle the concurrency in the

database; we will review this in detail in another chapter. The status of a given transaction could be any of the following:

IN_PROGRESS, COMMITTED, ABORTED, or SUB-COMMITTED

Any configuration parameter does not directly control this area's size, but PostgreSQL calculates it automatically.

Process memory

The **process memory**, as its name suggests, is allocated per process. We need to consider the number of processes we expect on the system when tuning the values for these memory areas. The more processes, the more memory will be used. *Figure 5.2* shows the process memory sub-areas.

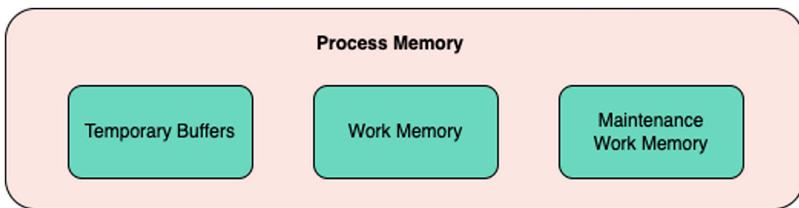


Figure 5.2: Process Memory areas

Temporary buffers

This memory sub-area is used when one or more temporary tables are created during a user session. These are special kinds of tables visible only in the user session context; since this memory area is not shared, no other process or session can “see” the temporary tables.

The **temp_buffers** parameter controls the size of the temporary buffer. The default size is 8MB; we can change this value dynamically, so no service restart is required.

Work memory

The work memory sub-area is the “main” portion of memory per user session. Here all the sort and hash-table operations are processed. A sort operation can be executed when a query includes **ORDER BY** or **DISTINCT**, for example and the hash-table operations are executed with every hash-join or hash-based aggregation. If the work memory area is insufficient to process the whole operation, then PostgreSQL will use temporary files to continue the operation. Since this last involves IO calls, it is bad for performance. We will dig deeper into this in future chapters.

We can control the size of this memory sub-area with the `work_mem` parameter. The default size is 4MB. We can modify this value dynamically.

Maintenance work memory

This sub-area is dedicated to the maintenance tasks such as `VACUUM`, `CREATE INDEX`, or adding a `FOREIGN KEY` to a table. Multiple sessions usually do not execute these operations concurrently, so this area can be significantly larger than the work memory.

The `maintenance_work_mem` parameter defines the size of this area. Its default value is 64MB, and changing it doesn't need a service restart.

Background processes

Unlike many other RDBMS, postgres uses `processes` and not threads. Whenever the service of Postgres is started, multiple background processes starts along which acts as backbone for serving the multiple concurrent requests from user. Please find the various background processes as shown in *Figure 5.3*. Depending upon various settings configured in the postgres, these background process may vary. For example, Archiver process will only be active in case archival is on in the configuration of postgres.

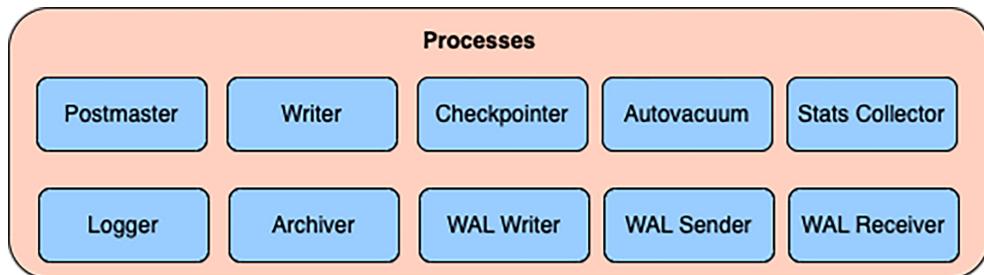


Figure 5.3: PostgreSQL background processes

Let us try to understand each process one by one.

Postmaster

Postmaster is like brain or heart of PostgreSQL background process. This is the first process which gets started when postgres service is started. It is like a controller for all other processes. If this process is killed then postgres stops working. Apart from being the first process, a new postmaster process will be invoked upon each DB

connection. The process ID (PID) of initial postmaster process acts as a parent id for other processes which gets started upon new connection.

Checkpointer

To understand the writer process, it is important to understand what is checkpoint and what is its importance.

Whenever the checkpoint is issued, it means there is a logical end to the transaction and from that point onwards all the data is marked to be written to the hard disk from the temporary buffers. In simple words, data is moved from RAM (which is faster memory but temporary) to the Hard Disk (where permanent data is stored but it is comparatively slower than RAM). In postgres terminology, Checkpoint is an event or occurrence which marks all dirty pages to be written to the disk.

Checkpoint occurs at every **checkpoint_timeout** or **max_wal_size** - whichever is first as per configuration set in Postgres instance level. (that is postgres.conf file which we will discuss later in this book).

So the question is why cannot database write data directly to the disk and performs this additional step of moving data from buffers to disk. Dirty buffers work in sequential manner and does not need to search the block which occupies time, it just writes the data and is faster as it is stored in RAM to reduce the DB response time. When checkpoint is issued all dirty blocks are moved from WALs to Disk which is a permanent storage. At the time of crash, DB will try to have the data available which is saved until last checkpoint which by default occurs every 5 mins. While starting/restarting postgres service, it first applies any unsaved WALs to data files and then up the service. Also, whenever DB service is stopped or restarted normally, automatically postgres issues checkpoint so that all data is available when DB is started the next time and it can avoid applying WAL files to Data files.

This concept will be more clear when we will discuss about next background process which is Writer. (*Reference: PostgreSQL Checkpoint, PostgreSQL Autovacuum launcher*)

Writer or background writer

When the postmaster service is started, background writer is also started along. The main task of background writer is to make sure that as much as possible keep buffers free. Same as checkpoint, background writers also writes data to disks. Along with this, they also perform the checkpoint which in turn helps checkpointer process to reduce IO operations as background process will majorly flushes dirty blocks to disks. (*Reference: PostgreSQL Background Writer*)

Autovacuum

Vacuuming is one of the maintenance activities to be performed by DBA to remove the bloated data from tables and indexes. Also, internally autovacuum also runs based on default configurations. Autovacuum launcher and autovacuum workers are responsible for performing as well as maintaining track of vacuum operations on all tables in DB. Autovacuum process is the main service responsible to carry on this activity. Since by default autovacuum is on, any postgres instance will have this process running in the background. (*Reference: PostgreSQL Autovacuum launcher*)

We will discuss more on vacuum further in this book.

Stats collector

Data about data is the meta data which is collected by stats collector process. Stat collector works to keep `pg_stats_*` meta data views updated. For example, whenever any activity is happening in tables than it gets recorded into `pg_stat_all_tables` via stats collector process. In the same `pg_stat_all_indexes` will have index related meta data and so on. Other view like `pg_tables`, `pg_indexes` will also get populated in turn.

Apart from meta data view other views like `pg_stat_activity` (which stores data of currently executing queries) are also populated using this process. This in turn helps in troubleshooting the issues. (*Reference: PostgreSQL stats collector*)

Logger

This service is on, in case `logging_collector` is on in `postgresql.conf` file. By default this parameter is off. However, it is recommended to set the `logging_collector = on` as it ll be helpful to generate postgresql logs which in turn helps in troubleshooting any issue. There is a separate section in `postgresql.conf` file to enable logs in desired format. (*Reference: PostgreSQL logger and WAL writer*)

Archiver

Archiver is not the mandatory process. In case `archive_mode` is enabled then one can see this process. However, in the production environment archive should be kept on. As the name suggests, archival of existing transaction logs will be done if `archive_mode = on` and this activity will be handled by archiver background process. In case of crash, system will recover the transaction files from archive files. The status of archive can be checked using `pg_stat_archiver` view. Also, along with `archive_mode`, `archive_command` determines how and at which location (may be cloud, or

separate mount point or some safe directory) archives are set. *Figure 5.4* shows the rough diagram of the archiver process of PostgreSQL.

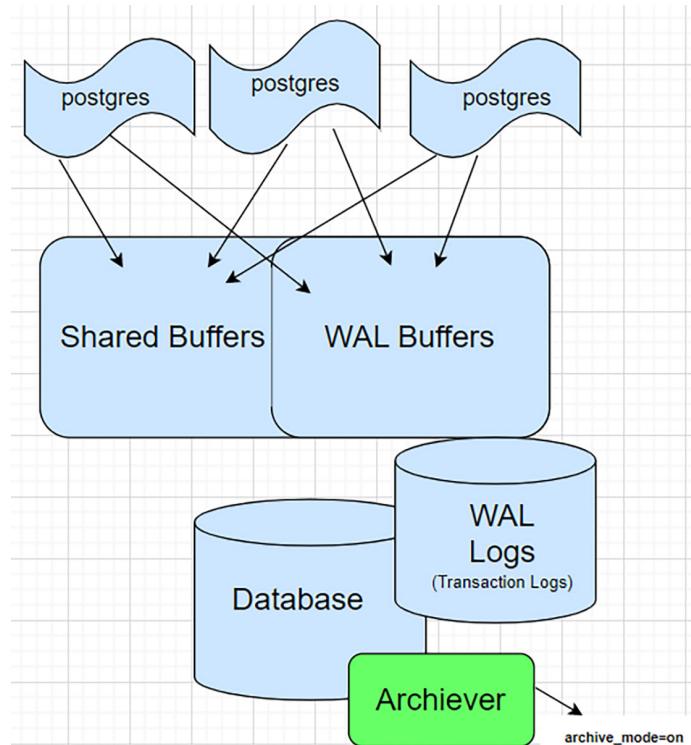


Figure 5.4: Archival process

WAL writer

WAL is a short form of **Write Ahead Logs**. WAL Files are nothing but the transaction files of the database. The responsibility of the WAL writer process is to write the data from WAL segments to WAL files or transaction files whenever commit is issued in the session. (Reference: PostgreSQL logger and WAL writer)

WAL sender

WAL Sender process is active only when replication is setup creating master and slave architecture. The main responsibility of the WAL sender is to send the data from the primary server to the secondary server. WAL sender process can only be seen on the master database.

WAL receiver

In the replication setup, WAL receiver can be found on secondary or slave server. As the name suggests, WAL receiver is responsible for receiving the data from the primary server. This process can be only be found on the slave DB server when the replication is configured.

Figure 5.5 shows the default background processes in a fresh postgresql setup.

```
postgres 1605 1 0 16:07 ? 00:00:00 /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
postgres 1607 1605 0 16:07 ? 00:00:00 postgres: checkpointer
postgres 1608 1605 0 16:07 ? 00:00:00 postgres: background writer
postgres 1609 1605 0 16:07 ? 00:00:00 postgres: walwriter
postgres 1610 1605 0 16:07 ? 00:00:00 postgres: autovacuum launcher
postgres 1611 1605 0 16:07 ? 00:00:00 postgres: stats collector
postgres 1612 1605 0 16:07 ? 00:00:00 postgres: logical replication launcher
```

Figure 5.5: Default background processes

Physical files

PostgreSQL has a well-defined physical file structure. As we saw in binary installation section of *chapter 2, Getting PostgreSQL to Work*, is just after getting postgres installed, we will see some directories within the data directory. Each one of those has a specific role. *Figure 5.6* shows the physical directory layout for a PostgreSQL instance:

```
[postgres@pg1-srv ~]$ echo $PGDATA
/var/lib/pgsql/14/data
[postgres@pg1-srv ~]$ ls -l $PGDATA
total 72
drwx----- 7 postgres postgres 71 Sep 30 22:50 base
-rw----- 1 postgres postgres 30 Oct 7 22:23 current_logfiles
drwx----- 2 postgres postgres 4096 Oct 7 22:23 global
drwx----- 2 postgres postgres 58 Oct 1 00:00 log
drwx----- 2 postgres postgres 6 Sep 30 21:07 pg_commit_ts
drwx----- 2 postgres postgres 6 Sep 30 21:07 pg_dynshmem
-rw----- 1 postgres postgres 4705 Sep 30 21:47 pg_hba.conf
-rw----- 1 postgres postgres 1636 Sep 30 21:07 pg_ident.conf
drwx----- 4 postgres postgres 68 Oct 7 22:23 pg_logical
drwx----- 4 postgres postgres 36 Sep 30 21:07 pg_multixact
drwx----- 2 postgres postgres 6 Sep 30 21:07 pg_notify
drwx----- 2 postgres postgres 6 Sep 30 22:18 pg_replslot
drwx----- 2 postgres postgres 6 Sep 30 21:07 pg_serial
drwx----- 2 postgres postgres 6 Sep 30 21:07 pg_snapshots
drwx----- 2 postgres postgres 6 Sep 30 21:56 pg_stat
drwx----- 2 postgres postgres 25 Oct 7 22:24 pg_stat_tmp
drwx----- 2 postgres postgres 18 Sep 30 21:07 pg_subtrans
drwx----- 2 postgres postgres 6 Sep 30 21:07 pg_tblspc
drwx----- 2 postgres postgres 6 Sep 30 21:07 pg_twophase
-rw----- 1 postgres postgres 3 Sep 30 21:07 PG_VERSION
drwx----- 3 postgres postgres 4096 Sep 30 22:58 pg_wal
drwx----- 2 postgres postgres 18 Sep 30 21:07 pg_xact
-rw----- 1 postgres postgres 192 Sep 30 22:48 postgresql.auto.conf
-rw----- 1 postgres postgres 28750 Sep 30 21:07 postgresql.conf
-rw----- 1 postgres postgres 58 Oct 7 22:23 postmaster.opts
-rw----- 1 postgres postgres 94 Oct 7 22:23 postmaster.pid
```

Figure 5.6: Physical PostgreSQL directory layout

Table 5. 1 shows the purpose of each one of these. (Reference: Database File Layout)

Item	Description
PG_VERSION	A file containing the major version number of PostgreSQL
base	Subdirectory containing per-database subdirectories
current_logfiles	File recording the log file(s) currently written to by the logging collector
global	Subdirectory containing cluster-wide tables, such as pg_database
pg_commit_ts	Subdirectory containing transaction commit timestamp data
pg_dynshmem	Subdirectory containing files used by the dynamic shared memory subsystem
pg_logical	Subdirectory containing status data for logical decoding
pg_multixact	Subdirectory containing multitransaction status data (used for shared row locks)
pg_notify	Subdirectory containing LISTEN / NOTIFY status data
pg_replslot	Subdirectory containing replication slot data
pg_serial	Subdirectory containing information about committed serializable transactions
pg_snapshots	Subdirectory containing exported snapshots
pg_stat	Subdirectory containing permanent files for the statistics subsystem
pg_stat_tmp	Subdirectory containing temporary files for the statistics subsystem
pg_subtrans	Subdirectory containing subtransaction status data
pg_tblspc	Subdirectory containing symbolic links to tablespaces
pg_twophase	Subdirectory containing state files for prepared transactions
pg_wal	Subdirectory containing WAL (Write Ahead Log) files
pg_xact	Subdirectory containing transaction commit status data
postgresql.auto.conf	A file used for storing configuration parameters that are set by ALTER SYSTEM
postmaster.opts	A file recording the command-line options the server was last started with

Item	Description
<code>postmaster.pid</code>	A lock file recording the current Postmaster Process ID (PID), cluster data directory path, postmaster start timestamp, port number, Unix-domain socket directory path (could be empty), first valid <code>listen_address</code> (IP address or <code>*</code> , or empty if not listening on TCP), and shared memory segment ID (this file is not present after server shutdown)
<code>postgresql.conf</code>	A file containing all the configuration parameters for the PostgreSQL instance. This is read from top to bottom, so in case there is any duplicated parameter, the last occurrence value will be the one applied.
<code>pg_hba.conf</code>	A file controls client authentication. It defines which hosts are allowed to connect, how clients are authenticated, which PostgreSQL user names they can use, and which databases they can access.
<code>pg_ident.conf</code>	A file to map the Operating System usernames with PostgreSQL usernames, so the corresponding authentication from the <code>pg_hba.conf</code> file can be used.

Table 5.1: PostgreSQL PGDATA layout description

The configuration files such as the **`postgresql.conf`**, **`pg_hba.conf`**, and **`pg_ident.conf`** are usually located in the same `$PGDATA` location, however, this can be adjusted. In Debian-based Operating Systems, for example, these files are located in the `/etc/postgresql/<VERSION>/<CLUSTER>/` path.

Now that we have listed the standard directory layout of PostgreSQL, we can review the files accordingly to their category. *Figure 5.7* shows the different file categories accordingly to their function.

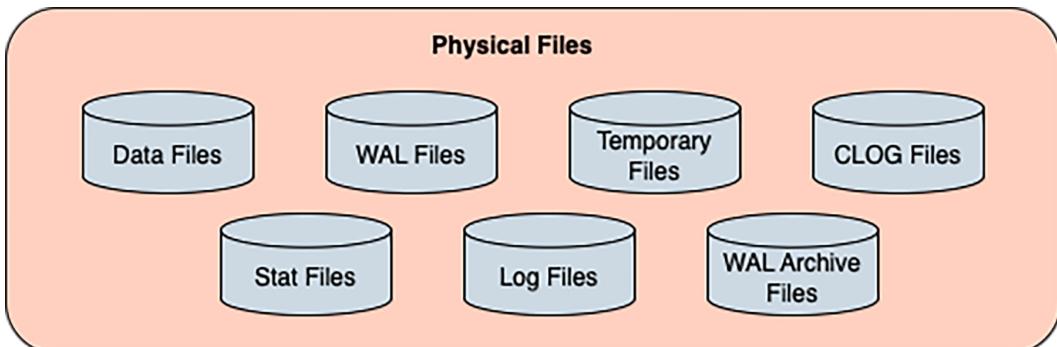


Figure 5.7: PostgreSQL physical files

Data files

In this category are all those physical files that store the database data, which might be tables data or indexes data. These data files exist per database and are stored within the **\$PGDATA/base** path, in which there is a directory per database and then the data files. These files hold the data written from the Shared Buffers we saw before, and are named after the **filenode** identifier, which PostgreSQL assigns to each table or index.

The data files are 1 GB size tops, and if the table they belong to is larger, another file named the same **filenode** is added, including a serial integer number to make a distinction. *Figure 5.8* shows the details of one table name **pgbench_accounts**, its size, and **filenode**, then the physical files within the **\$PGDATA** path:

```
pgbench=# SELECT relname AS tablename,
                  pg_size.pretty(pg_table_size(oid)) AS tablesize,
                  pg_relation_filenode(oid)
            FROM pg_class
           WHERE relname = 'pgbench_accounts';
      tablename | tablesize | pg_relation_filenode
-----+-----+-----+
 pgbench_accounts | 2562 MB | 16398
(1 row)

pgbench=# ! ls -lh $PGDATA/base/16385/16398*
-rw----- 1 postgres postgres 1.0G Oct 10 00:18 /var/lib/postgresql/14/main/base/16385/16398
-rw----- 1 postgres postgres 1.0G Oct 10 00:18 /var/lib/postgresql/14/main/base/16385/16398.1
-rw----- 1 postgres postgres 514M Oct 10 00:19 /var/lib/postgresql/14/main/base/16385/16398.2
-rw----- 1 postgres postgres 664K Oct 10 00:19 /var/lib/postgresql/14/main/base/16385/16398_fsm
-rw----- 1 postgres postgres 88K Oct 10 00:19 /var/lib/postgresql/14/main/base/16385/16398_vm
```

Figure 5.8: Physical data files

In the same *Figure 5.8*, you can see a couple of extra files with the suffixes **_fsm** and **_vm**; these are not strictly data files but exist per every table and index in the database. Their function is to hold details about the free space in the table pages or blocks and the number of *dead tuples*. We will cover these topics in future chapters.

WAL files

These files contain the data written from the WAL Buffers. They are stored within the **\$PGDATA/pg_wal** path. Every time a **COMMIT** occurs in the database the information from the WAL Buffers is flushed to these files. By default, the WAL Files are 16 MB in size.

The WAL Files are named accordingly to the **Log Sequence Number (LSN)**, which is a unique identifier in the transaction log and represents the position in the WAL stream. So PostgreSQL can know the right order of the transaction changes following this sequence and rebuild operations in the case of a recovery event. *Figure 5.9* shows how the WAL Files look:

```
pgbench=# \! ls -lh $PGDATA/pg_wal/ | head
total 1.1G
-rw----- 1 postgres postgres 16M Oct 10 00:18 000000010000000000000000000087
-rw----- 1 postgres postgres 16M Oct 10 00:18 000000010000000000000000000088
-rw----- 1 postgres postgres 16M Oct 10 00:18 000000010000000000000000000089
-rw----- 1 postgres postgres 16M Oct 10 00:18 00000001000000000000000000008A
-rw----- 1 postgres postgres 16M Oct 10 00:18 00000001000000000000000000008B
-rw----- 1 postgres postgres 16M Oct 10 00:18 00000001000000000000000000008C
-rw----- 1 postgres postgres 16M Oct 10 00:18 00000001000000000000000000008D
-rw----- 1 postgres postgres 16M Oct 10 00:18 00000001000000000000000000008E
-rw----- 1 postgres postgres 16M Oct 10 00:18 00000001000000000000000000008F
```

Figure 5.9:Physical WAL files

Temporary files

The temporary files (for operations such as sorting more data than can fit in the `work_mem` memory area) are created within `$PGDATA/base/pgsql_tmp`, or within a `pgsql_tmp` subdirectory of a tablespace directory if a tablespace other than `pgsql_default` is specified for them. The name of a temporary file has the form `pgsql_tmpPPP.NNN`, where `PPP` is the PID of the owning backend, and `NNN` distinguishes different temporary files of that backend.

The size of these files is variable, depending on how big is the data set being processed. However, we can set a limit to their size with the `temp_file_limit` parameter; any transaction requesting more temporary files beyond the limit will be canceled. One thing to keep in mind is this parameter limits the size of temporary files per process, so if multiple processes grow up at the same time up to the limit the available disk space can be compromised.

CLOG files

The **Commit LOG** Files (**CLOG** Files) are those used to store the information for the status of the transactions. As we saw in the Shared Memory subtitle from the *Memory Architecture* section of this chapter, this transaction status information is required by the MVCC process. These files are created in the **\$PGDATA/pg_xact** directory.

Stat files

The statistics in PostgreSQL are a collection of metadata describing various details about how the data is being stored, distributed, and accessed in the database system. As we already studied, a couple of background processes are in charge of gathering and collecting this information. The same is stored in the **\$PGDATA/pg_stat** and **\$PGDATA/pg_stat_tmp** paths.

Log files

There is no doubt that database log files are truly useful when you need to troubleshoot or verify a database system's general health and operation. PostgreSQL has a good number of parameters to control what to log, in what format, where to store it, and how long to keep or rotate.

Tuning the logger to log all the information you might need is crucial, definitely taking a look at the PostgreSQL Community documentation is worth it. You can visit it at <https://www.postgresql.org/docs/14/runtime-config-logging.html>.

WAL archive files

Finally, the WAL archive files deserve their own mention. The files are just an identical copy of the WAL Files we saw before in this section. The purpose of having "another" copy of the transaction metadata is to ensure we are able to restore backups to consistent points, create and maintain replicas, and recover running systems.

The design of the WAL files is to be kept in the system just the minimum necessary, then they are reused or removed. This behavior is controlled with the **wal_keep_segments** parameter for the versions up to PostgreSQL 12 or **wal_keep_size** for versions up to PostgreSQL 13. Then if we want to keep the WAL files for longer

periods, such as necessary to ensure a good backup recovery, we need to create these copies, called WAL Archive Files.

Then once a PostgreSQL system is configured to have the archive enabled, using the **archive_mode** parameter, we can configure a routine to copy the WAL files to the storage we want using the **archive_command** parameter. This parameter can receive a bash command for example, or the path to a script that will be executed at the operating system level. This brings an enormous level of customization. Next are a couple of examples of how an **archive_command** parameter might look.

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/
server/archivedir/%f'
```

The above command will *test* if the given file (%f) does not exist on **/mnt/server/archivedir** path, if true (it does not exist) then will be copied, from the source path (%p).

```
archive_command = /usr/bin/pg_archive.sh %p %f
```

In the above example, the archive command will execute a shell script that received the source path of the WAL file (%p) and the single name of the same (%f). In this case, all the logic of what is done is contained in the script, this needs to have the right privileges to be executed by the Operating System user which runs the PostgreSQL service, usually named *postgres*.

Conclusion

In this chapter, we have tried to cover the PostgreSQL internals and have touched the most important concepts from PostgreSQL Architecture perspective which is Physical, Memory and Background processes. PostgreSQL is enhancing day by day and we may see minor changes in this architecture depending on which version of PostgreSQL is being used. Here we have tried to cover architecture from PostgreSQL 14.

Figure 5.10 shows the full architecture of PostgreSQL in brief:

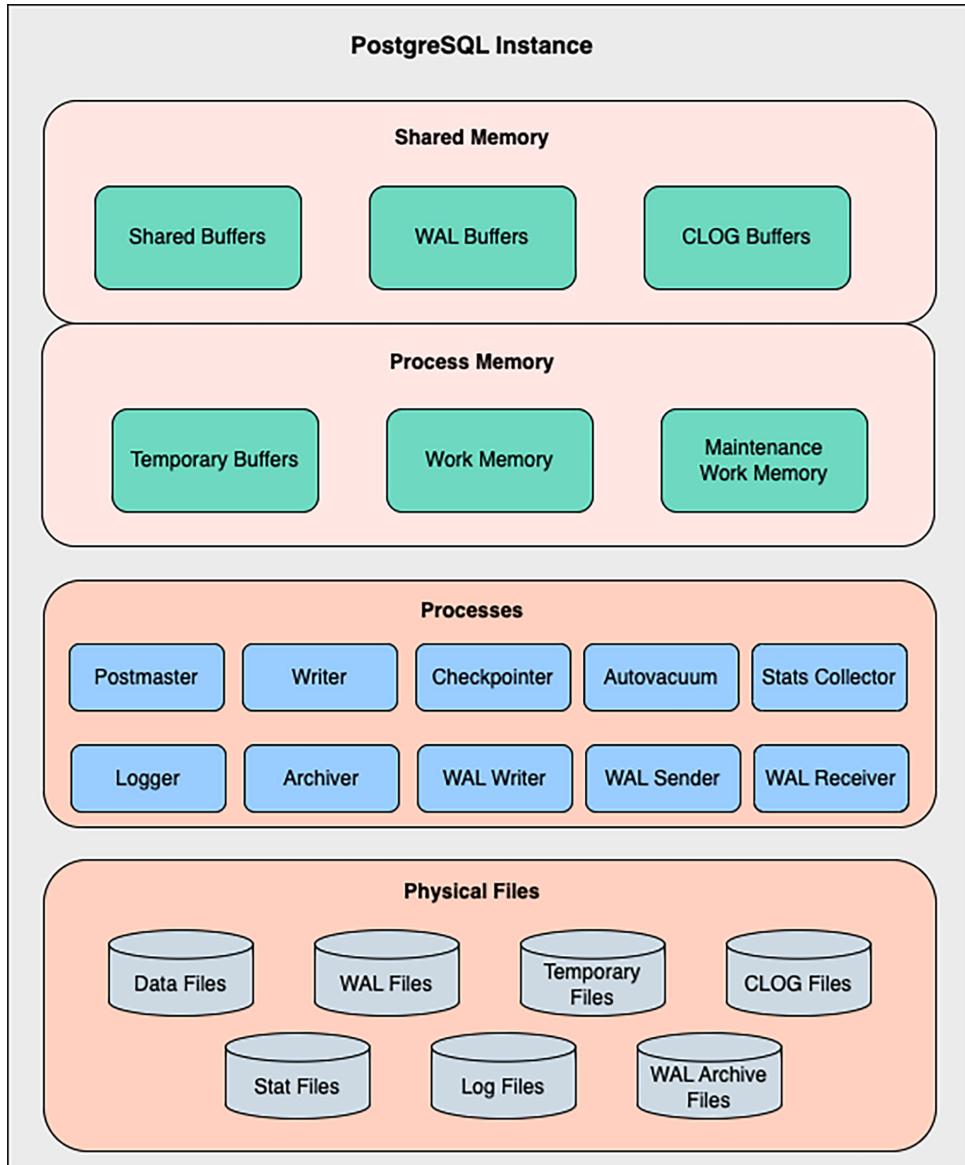


Figure 5.10: Architecture of PostgreSQL

In the next chapter, we will touch upon few more concepts of PostgreSQL Internal like ACID, MVCC, Transaction Isolation Levels, Query Processing and Vacuum.

Bibliography

- PostgreSQL Checkpoint: <https://postgreshelp.com/postgresql-checkpoint/>
- PostgreSQL Autovacuum launcher: <https://postgreshelp.com/postgresql-autovacuum-launcher/>
- PostgreSQL Background Writer: <https://postgreshelp.com/postgresql-background-writer/>
- PostgreSQL stats collector: <https://postgreshelp.com/postgresql-stats-collector/>
- PostgreSQL logger and WAL writer: <https://postgreshelp.com/wal-writer-and-logger/>
- Database File Layout: <https://www.postgresql.org/docs/current/storage-file-layout.html>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 6

PostgreSQL Internals

Introduction

PostgreSQL is highly efficient, and its code is very optimized. Just out of the box, it can start working and delivering a really good user experience, like a *magic* box that receives queries and returns data. This is possible because multiple subsystems work cooperatively and coordinately, each responsible for a specific part of the process. Knowing what is inside PostgreSQL and how it works is essential in the DBA's path.

Structure

In this chapter, we will learn about the internals of PostgreSQL. This chapter is going to cover following main concepts:

- ACID
- MVCC
- Vacuum
- Transaction Isolation Levels
- Query Processing

Objectives

In this chapter, we will learn the different internal subsystems and concepts involved in the PostgreSQL functionality. Once you complete it, you will relate the different stages from a user session starting a query until the result is gotten. How they affect each other in a concurrent system, and the different subsystems that enable and maintain PostgreSQL ready to operate with multiple user sessions at the best performance.

ACID

The full form of ACID is **Atomicity, Consistency, Isolation, and Durability**. All RDBMS majorly follow ACID properties to manage the transaction level. Let us try to understand each property.

Atomicity

PostgreSQL makes sure that transactions are fully complete, or they fail entirely in case of any failure. It does not leave the transaction state in between or partially complete state.

Consistency

At any point, transactions are in a consistent state and adhere to a set of rules defined. At any given time, data integrity checks (constraints) maintain consistency within the database.

Isolation

Concurrent transactions happen within the database. Isolation property makes sure that each transaction is separate and maintains its state within that transaction. One transaction cannot interfere with any other transaction unless any input or output, or processing needs to be shared amongst the transactions.

Durability

At any given point, database transactions, once committed or rolled back, must maintain as it is, no matter whether the same DB is accessed at the same time or years after the transaction. It should be durable and show the same result despite failovers occurring in the Data Center or Database. If there are any corruptions or

any other unavoidable circumstance then we may not be able to access the data but otherwise data should be constant irrespective of when or from where it has been accessed.

MVCC

The Full form of MVCC is **MultiVersion Concurrency Control**. MVCC Architecture of PostgreSQL ensures that all ACID properties are compliant. At any given time, whenever concurrent transactions are performed on the same DB Objects or different DB Objects, DB will be in the same consistent state, and all transactions are isolated.

Readers do not block the writers, and writers do not block readers either. While querying any DB Object, PostgreSQL ensures that DB is in the same consistent state as it was a while ago unless any change has occurred. Multiple copies of DB Objects are created in case the same table/view is being accessed concurrently. The moment transaction gets committed, then the object gets updated, and from the next time onward, the updated records are visible to the user.

Each transaction is given transaction id (*xid*), and if the current *xid* > the *xid* which was committed a while ago, then the current *xid* can see the updated records.

Let us try to understand the same using an example. Let us say that Account A has 2 Debit Cards. One debit card is used by the mother and another by the child studying at a different location. As mentioned earlier, the underlying account is the same for both debit cards. It can very well happen that both mother and child can access the same account using both their own debit cards at the same time concurrently.

Both will be reading the same account balance as readers do not block other readers. When both will try to withdraw money from ATMs, a writer will block the other writer. In this scenario, either both will be declined to withdraw money as that might create inconsistent data, or one transaction will be successful, and the other will not be as both are reading the same data while inserting ATM Debit Card, but when both will withdraw the money that time the transaction value should be withdrawn from both cards and balance needs to be updated accordingly.

Say the Balance is \$1000, and withdrawing \$100 from each card should make the balance $\$1000 - \$100 - \$100 = \800 . If ACID & MVCC are not followed, both will be shown $\$1000 - \$100 = \$900$, which is incorrect. That is why due to MVCC, either both will be declined, or only one person will be successful. Then the ones whose transaction declined will try a second time; he/she will be shown the balance as \$900, and then withdrawing \$100 will make the final balance \$800.

In this way, transactions are in a consistent, durable, and isolated state at any point so that MVCC get accomplished.

We will see transaction management in more detail further in this chapter.

Vacuum

This is one of the routine maintenance activities for PostgreSQL Databases. To understand the vacuum, let us first try to understand how the **UPDATE** and **DELETE** of a row work in PostgreSQL databases.

At the time of an **UPDATE** operation, the PostgreSQL database performs two major steps:

1. It inserts new records with updated values.
2. It marks the existing row for deletion and deactivates the same.

Eventually, when a user selects the records, only updated records are visible. At the time of **DELETE**, rows are marked for deletion and deactivated but physically not removed.

In this way, all the rows marked for deletion are known as *dead tuples*. These *dead tuples* in the PostgreSQL database, are removed physically by performing **VACUUM** activity.

VACUUM can be compared with the fragmentation and defragmentation concept of the Operating System where internal disk space is formatted by cleaning each block of disk space. This process helps in freeing the space within the data files pages.

The vacuum operations can be executed on the database from different sources and levels. In the following subsections, we will learn about them.

Autovacuum

As the name suggests, it is automatically performed by PostgreSQL, and by default, autovacuum is enabled in **postgresql.conf** file. However, this is an optional setting the PostgreSQL community highly recommends this setting be kept on.

Autovacuum is controlled by the autovacuum process of PostgreSQL, which is active when the autovacuum is on. Autovacuum is performed as per the configuration set in the configuration file. One can customize the same for specific tables and indexes. Autovacuum will help remove *dead tuples*, and it utilizes that space released by the removal of *dead tuples* whenever new DML operations are performed on the database.

It will not free the disk space back to the operating system; however, DB / Object Size will have that space to be used for new data. This is purely an online activity that helps in increasing performance while querying the tables.

VACUUM FULL

This is a manual process that needs downtime for that particular object on which the VACUUM FULL is running. It can be executed on the Database Level or Table Level. This helps in giving space back to Operating System. Bloated data or dead tuples are completely removed to free the space.

In the background, performing VACUUM FULL creates a copy of the DB object and copies only the active or required data. Toward the end, it will remove the old table with the bloated size and keep the new one without dead tuples. That is why the object on which the VACUUM FULL is being performed becomes inaccessible till the vacuum is completed.

Also, at the start of the activity, a pre-requisite VACUUM FULL needs to double the table size. For example, if any table is of 50 GB size, then a minimum of 100 GB disk space is needed to perform VACUUM FULL on that table. At the end of the activity, the table size will reduce depending upon the bloat data size of the table. That is why one must check the bloats in the table before performing VACUUM FULL. If bloats are less or negligible, then it is not recommended to perform VACUUM FULL on the table.

Manual VACUUM

It is the same as autovacuum, with the only difference that it will not be triggered automatically and needs to be performed manually.

Please find the most frequent options used with **VACUUM**:

- **FREEZE**: The age of the table can be reset using this option.
- **VERBOSE**: It prints the detailed log of each stage, metadata count, and so on. details on the screen.
- **ANALYZE**: It will update the metadata which can be helpful for the query planner while executing / processing any query.

Whenever the vacuum is running, the below queries might help to get details of the ongoing maintenance of the DB.

```
SELECT now()-query_start age, * FROM pg_stat_activity where query
ILIKE '%vacuum%';
```

This query gives the current vacuum without disclosing at which phase of the vacuum is currently in progress. Rather it can give the details like when was vacuum started, from which IP/Host and the like.

```
SELECT relid::regclass,* FROM pg_stat_progress_vacuum;
```

This query gives details about the different phases of the vacuum:

- Initializing
- Scanning heap
- Vacuuming indexes
- Vacuuming heap
- Cleaning up indexes
- Truncating heap

Performing final cleanup(*Reference: Progress Reporting*)

```
SELECT relname, last_vacuum, last_autovacuum, last_analyze, last_autoanalyze  
FROM pg_stat_user_tables  
WHERE relname = '<table_name>';
```

This query gives details of the last vacuum performed on a given table.

pg_repack

This is an open-source extension that can be used instead of VACUUM FULL without minimal downtime. This helps reclaim the disk space and occupy the lock for a very short duration (usually in milliseconds or seconds) at the start and end of the activity.

When the **pg_repack** command is issued on any table or index, it will acquire a lock to create an initial copy of that object. It will remove the dead tuples on the object copy, and at the end of the activity, it will again acquire a lock that will rename the new table to the existing name and process the data which was modified during the repack process.

This is one of the good options as an alternative to VACUUM FULL. When **pg_repack** is in progress, DDL operations cannot be performed on the table as it holds an access share lock during the repack process.

Preventing transaction ID wraparound failures

MVCC transaction semantics depend on comparing transaction ID (*xid*) numbers: a row version with an insertion *xid* greater than the current transaction's *xid* is "in the future" and should not be visible to the current transaction. But since transaction IDs have a limited size (32 bits), a cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound.

To avoid this, every table in the database must be vacuumed at least once every billion transactions. Autovacuum, if enabled, takes care of transaction wraparound issues.

Transaction isolation levels

As we learned in the first section of this chapter, the "T" from the ACID acronym stands for Isolation. This is one of the features the SQL standard defines for a **Relational Database Management System (RDBMS)** and means how the concurrent transaction affects each other.

PostgreSQL allows four transaction isolation levels: read uncommitted, read committed, repeatable read, and serializable. Each guarantees transactions free from the different effects of other concurrent transactions, known as phenomena. Let us start reviewing the phenomena and their effects to understand these concepts better.

Phenomena

There are also four possible phenomena types: dirty read, nonrepeatable read, phantom read, and serialization anomaly.

Dirty read

The dirty read refers to the effect of one transaction being able to read the uncommitted changes from other transactions. In PostgreSQL, this is not possible at any isolation level. *Figure 6.1* illustrates how this would affect two concurrent transactions:

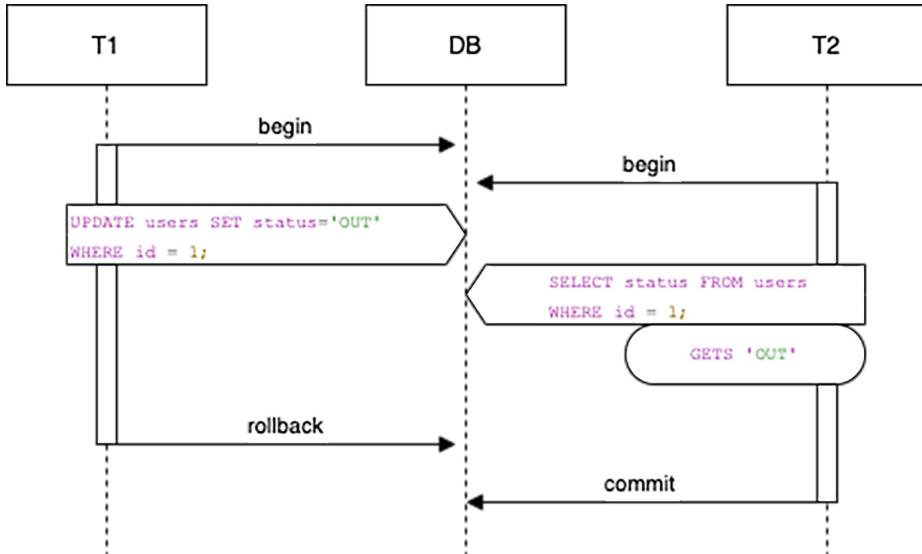


Figure 6.1: Dirty reads

Non-repetable read

This refers to the effect of one transaction reading different results from the same query during its duration. This is because other transactions commit changes. Figure 6.2 shows how this looks:

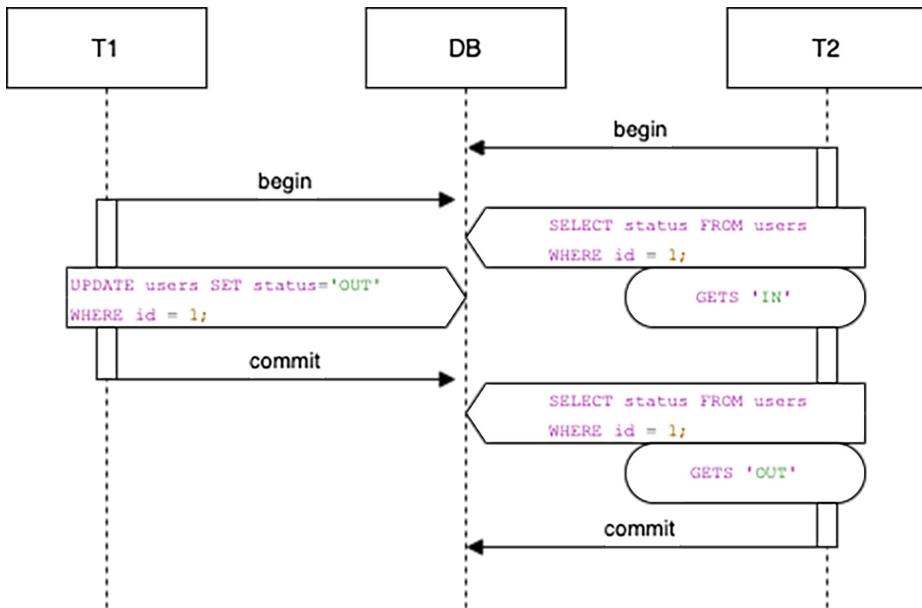


Figure 6.2: Nonrepeatable reads

Phantom read

This means one transaction reading a different set of rows from one previous read. This is because another transaction has added or removed rows from the resulting set and has committed these changes. *Figure 6.3* illustrates the flow of this:

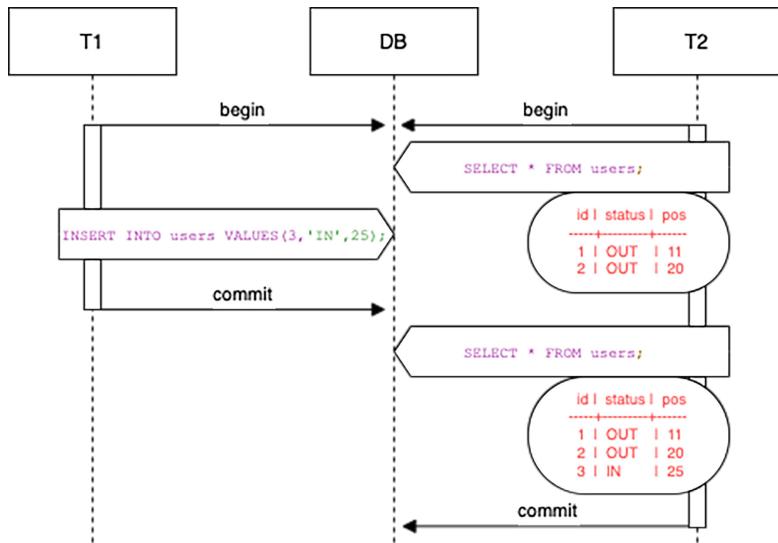


Figure 6.3: Phantom read

Serialization anomaly

This refers to the effect of successfully committing a group of transactions whose final result can not be reproduced by executing the transactions one at a time in any order:

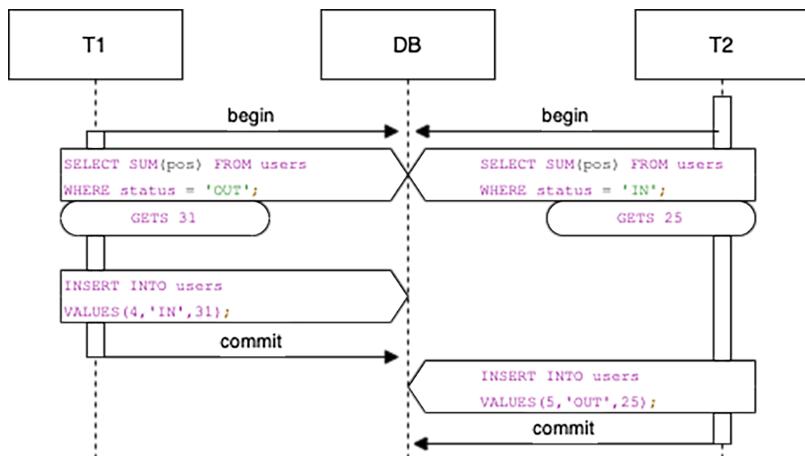


Figure 6.4: Serialization anomaly

As shown in *Figure 6.4*, each transaction executes a **SELECT** to get the summary of the column **pos**; the first filters by the column **status** equal to 'OUT', and the second filters by **status** equal to 'IN'. Then each one inserts a new row setting the column **pos** value the same as the result of their initial **SELECT** and with the opposite value for the **status** column. It is not possible to get the same results for each statement executing the transactions one by one since the initial result for the **SELECT** will be different depending on which **INSERT** happens first.

Isolation levels

The phenomena effects can impact some application designs. The different isolation levels supported by PostgreSQL add protection against them. Each one increases the protection level for different phenomena. *Table 6.1* describes the protection at each level:

	Protection against phenomena			
Isolation level	Dirty read	Nonrepeatable read	Phantom read	Serialization anomaly
Read uncommitted	YES	NO	NO	NO
Read committed	YES	NO	NO	NO
Repeatable read	YES	YES	YES	NO
Serializable	YES	YES	YES	YES

Table 6.1: Protection against phenomena on each isolation level

Read uncommitted/committed

In PostgreSQL the read uncommitted isolation level behaves the same as the read committed, which is the default mode. In this isolation level, each statement within a transaction is executed with its own snapshot. We could imagine a snapshot as "photography" of the database data at a given time.

This is why all the phenomena, except the dirty read, are possible in this mode. However, this makes sense for some workflows. Considering the flow illustrated in *Figure 6.2*, now think the transactions from the example are banking operations updating and consulting the balance from an account; it would be expected that T2 would see the changes T1 did. So if the balance is updated in T1, T2 will operate with the new value afterward.

Remember that other, more complex workloads can experience undesired results due to the phenomena allowed in this isolation level. As a general rule, we could say the read uncommitted / committed is effective for workloads with simple operations, but more complex logic might require an extra isolation level.

Repeatable read

When using this isolation level, PostgreSQL guarantees free from all phenomena except the serialization anomaly. This is achieved by working with the same snapshot since the beginning of the transaction, so any statement executed after the first one will see the same data.

A transaction running at this level will see only the data committed before it begins, and the only visible changes will be those done within the same transaction. The applications that rely on this isolation level must be prepared to repeat the transactions if required due to the possibility of serialization failures.

If a transaction uses this isolation level and another concurrent transaction updates or deletes one row and commits the change, and then the repeatable read transaction tries to update the same, the transaction will roll back with the error message shown in *Figure 6.5*:

<pre>T1 (18:06:37) =# BEGIN; BEGIN T1 (18:06:59) =#* T1 (18:07:11) =#* T1 (18:07:11) =#* T1 (18:07:22) =#* SELECT status FROM users WHERE id = 1; status ----- IN (1 row) T1 (18:07:40) =#* UPDATE users SET status = 'OUT' WHERE id = 1; UPDATE 1 T1 (18:08:20) =#* T1 (18:08:29) =#* COMMIT ; COMMIT T1 (18:08:39) =# T1 (18:08:44) =# []</pre>	<pre>T2 (18:06:37) =# BEGIN; BEGIN T2 (18:06:59) =#* SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; SET T2 (18:07:07) =#* T2 (18:07:22) =#* SELECT status FROM users WHERE id = 1; status ----- IN (1 row) T2 (18:07:40) =#* T2 (18:08:20) =#* T2 (18:08:29) =#* T2 (18:08:35) =#* T2 (18:08:39) =#* T2 (18:08:44) =#* UPDATE users SET status = 'EMPTY' WHERE id = 1; ERROR: could not serialize access due to concurrent update ← T2 (18:09:08) =#! ROLLBACK ; ROLLBACK T2 (18:09:14) =# []</pre>
--	---

Figure 6.5: Repeatable read serialization failure.

Figure 6.5 shows how this isolation level protects against the nonrepeatable read and phantom read phenomena, but the cost is the transaction being rolled back. It is necessary to keep this in mind when designing the application.

Serializable

This is the highest isolation level PostgreSQL offers as described in *Table 6.1*, this level protects against all phenomena. Using this level emulates the execution of the transaction as if all are executed serially rather than concurrently.

This level behaves the same as the repeatable read, so a single data snapshot is used since the beginning of the first statement within the transaction. But it also adds some extra monitoring to detect any condition in which the serial execution of the transactions (one at a time) in any order could produce inconsistent results, preventing the serialization anomaly. This monitoring doesn't add extra locking in the database operations but adds extra overhead to the system in general.

Just as the repeatable read isolation level, the applications using this level must be prepared to handle the transaction failures and retry them. *Figure 6.6* illustrates the error message thrown if the serialization anomaly is detected.

```

T1 (18:36:25) =# BEGIN;
BEGIN
T1 (18:36:29) =#* SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
T1 (18:36:47) =#* SELECT sum(pos) FROM users WHERE status = 'OUT';
sum
-----
31
(1 row)

T1 (18:37:01) =#* INSERT INTO users VALUES(4,'IN',31);
INSERT 0 1
T1 (18:37:16) =#* COMMIT;
COMMIT
T1 (18:37:21) =# []

T2 (18:36:25) =# BEGIN;
BEGIN
T2 (18:36:29) =#* SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET
T2 (18:36:47) =#* SELECT sum(pos) FROM users WHERE status = 'IN';
sum
-----
25
(1 row)

T2 (18:37:01) =#
T2 (18:37:30) =#* INSERT INTO users VALUES(5,'OUT',25);
ERROR: could not serialize access due to read/write dependencies among t
ransactions ←
DETAIL: Reason code: Canceled on identification as a pivot, during write
.
HINT: The transaction might succeed if retried.
T2 (18:37:34) =#! ROLLBACK ;
ROLLBACK
T2 (18:37:50) =# ]

```

Figure 6.6: Serialization anomaly detected.

We can conclude each isolation level adds an extra layer of protection against the concurrent transaction effects, known as phenomena, but with a concurrency cost. So, the higher the isolation level, the lower the concurrency performance.

Query processing

We could say that query processing is PostgreSQL's most sophisticated and complicated subsystem. We will learn about it in the simplest and easy-to-understand way.

As you already should imagine, this is related to the list of steps postgres executes to resolve the queries from a client and return the requested data in the case of a SELECT or perform the required data modifications in the case of any other **Data Manipulation Language (DML)** instruction, such as INSERT, UPDATE or DELETE.

We learned in the background processes section from *Chapter 5* a background process is created for each client session, which receives the queries from the client side and communicates with the server to accomplish the following stages for every query.

- Parser
- Rewriter
- Planner
- Executor

Figure 6.7 illustrates the flow of these stages and the products of each one to the next:

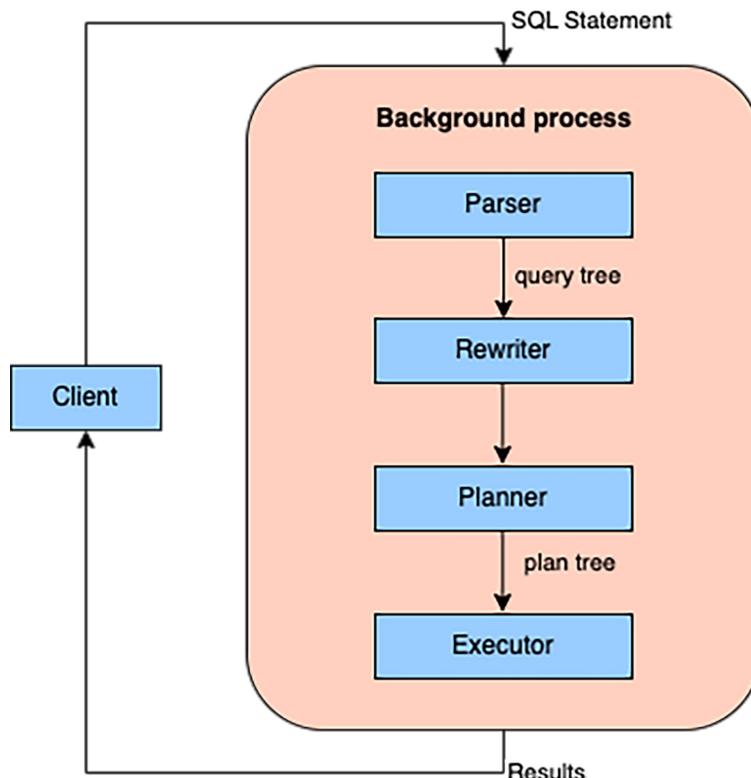


Figure 6.7: Query processing stages

Parser

This is the first stage when a client sends a query. The query text is checked for correct syntax and creates a *query tree*. The *query tree* separates the statement into all the different components (Reference: *Query Processing Components*)

- **Type:** What kind of statement is processing, **SELECT**, **INSERT**, **UPDATE**, or **DELETE**.
- **Range Table Entry (RTE):** A list of all the tables, subqueries, views, and so on., involved in the query.
- **Result relation:** To where the results for **INSERT**, **UPDATE** or **DELETE** will land.
- **Target list:** The results from the query. These are the columns listed in a **SELECT** or the values for **INSERT / UPDATE**.
- **Qualification:** A boolean value specifies where the operation should be executed or not over the final result. This corresponds to the **WHERE** clause.
- **Join tree:** The list of **JOIN** operations after the **FROM** clause, if any.
- **Others:** Some extra operations, such as the **ORDER BY** clause.

Rewriter

This stage takes the *query tree* from the previous stage and rewrites it with any rule for the set of tables or rows in the system catalog if it applies. We will learn more about rules in *Chapter 12, Advance Database Objects*.

A practical example of what happens in the rewriter stage is when a statement is querying from a view. In this stage, the view call will be replaced by its current definition to access the tables contained in it.

Planner

Easily, this is the most critical stage of the query processing path. Here PostgreSQL will take the last rewritten query plan from the previous stages and search for the fastest way to access and retrieve the required data.

To accomplish this goal, the planner verifies different paths leading to the same result and calculates the associated cost of each one, and the cheapest path is chosen. The cost of every option is calculated regarding the required resources to process it, such as CPU cycles, I/O operations, and the like.

To determine these calculations, the planner relies on the data statistics, which are metadata that describes the data distribution alongside the table/index data files and other details so the planner can decide the specific steps to reach the data. This is why having fresh statistics is crucial for a good performance. We will learn more about this in *Chapter 13, Performance Tuning*.

Once the optimal plan is determined, the planner creates a *plan tree* with nodes representing physical operations, such as Index Scan, Sequential Scan, and the like.

Figure 6.8 illustrates a simple plan tree with a single Seq Scan node. This *plan tree* is the output the next and final stage can use.

SELECT status FROM users WHERE id = 1;						
#	exclusive	inclusive	rows_x	rows	loops	node
1.	0.015	0.015	↑ 6.0	1 - 3	1	→ Seq_Scan on users users (cost=0..25 rows=6 width=32) (actual time=0.014..0.015 rows=1 loops=1) Filter: (users.id = 1) Rows Removed by Filter: 3 Buffers: shared hit=1
Planning I/O : Buffers: shared hit=35						
Execution time : 0.06 ms						

Figure 6.8: Visual representation of a plan tree. (Reference: Explain)

Executor

The executor stage takes the *plan tree* from the previous stage and processes it recursively. Then it retrieves rows in the way represented by the plan. Each time a plan node is executed, the executor delivers a row or reports back to notify that it has finished.

Depending on the statement's type, the rows are returned accordingly:

- **SELECT**: The rows are returned directly to the client as the resultset.
- **INSERT**: The returned rows are inserted into the specified table.
- **UPDATE**: Each returned row includes all the updated column values and the row ID of the target row. Then a row with the updated values is inserted, and the old row is marked as deleted.
- **DELETE**: Like the UPDATE, each returned row includes the row ID, which marks the row as deleted.

Conclusion

The PostgreSQL internals are a collection of well-designed and tuned subsystems. The conjunction of all of them makes possible all the cool stuff postgres is capable of.

We have seen how transactions are managed internally to achieve the MVCC and ensure it follows ACID Compliance. And the stages happening when a query is sent to the DB to be executed. Along with this, we have touched on one of the major maintenance activities of PostgreSQL VACUUM.

In the next chapter, we will see another important concept for PostgreSQL DBAs: Backup and Restore.

Bibliography

- Progress Reporting: <https://www.postgresql.org/docs/10/progress-reporting.html>
- Explain: <https://explain.depesz.com/>
- Query Processing Components: <https://learn.microsoft.com/en-us/training/modules/understand-postgresql-query-process/2-identify-components>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Backup and Restore in PostgreSQL

Introduction

One of the important aspects of administering the database is to take regular backups and test the same by restoring the same. At any point in time, DBA might need to restore the previous dates' backups, which are important to perform the point-in-time recovery and used to roll back the accidental changes done to the database. Backups are extremely important from the high availability standpoint as well. They can build or rebuild standby servers as an initial copy.

Structure

In this chapter, we will learn how to take a backup of the PostgreSQL database and how to restore the same. This chapter is going to cover the following concepts:

- Backup
- Restore
- Useful Backup and restore tools

Objectives

In this chapter you will learn about the different types of backups you can take from your PostgreSQL database, their differences, and how to restore them. Knowing this is essential to build your path as PostgreSQL DBA. Also, we will review a few of the most used backup tools provided by the community, which provide extra functionalities and enhancements.

Backup

Database should be backed up at regular intervals, which can help in maintaining the copy of the data. In case any unexpected changes happen to the database can be reverted by restoring the backup. Backup can also help in performing point-in-time recovery if needed. Backups are important to build / rebuild the slave databases too.

As shown in *Figure 7.1*, please find the different backup types and the tools given by the community for the same:

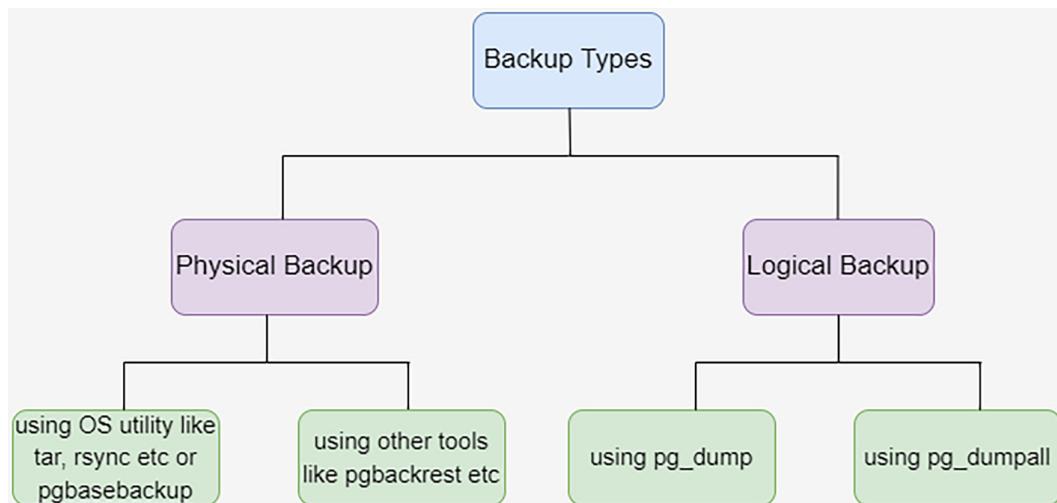


Figure 7.1: Backup Types

Physical backup

Physical backups are filesystem-level backups that can be taken, by copying the data directory along with the custom tablespace directory from one server to another server. The only challenge here is to take backup consistently. However, one can take backup at the cluster level, and single DB object level backup like single DB level or table level backup is not possible. As far as consistency of the backup is concerned, one can adhere to the high-level steps to take physical backup as following.

1. Create a checkpoint using the **CHECKPOINT** command and **pg_start_backup** ('**any_label_name**') command to tell the database that it has to issue a logical end to ongoing transactions. Basically, it will switch the **Write Ahead Log (WAL)** files and keep a mark internally saying that backup has been taken till the marked point.
2. Copy the physical data directory along with tablespace directory (if created outside data directory) using any OS utility like **tar** or **rsync**
3. Issue **pg_stop_backup()** command once the copy is completed as mentioned in step 2.

pg_basebackup

A utility given by PostgreSQL community to take backup of the whole PostgreSQL database cluster. (Reference: **pg_basebackup**)

Following is the simplest example of how the **pg_basebackup** utility can be used to take a backup of the data directory using the “-D” option

```
postgres@ip-172-31-27-8:~$ pg_basebackup -U postgres -h localhost -x  
-D /usr/local/pgsql/data
```

Figure 7.2 shows the various options which can be used with the **pg_basebackup**. Note, how the OS help command has been used to fetch the usage of the command.

```

postgres@ip-172-31-27-8:~$ pg_basebackup --help
pg_basebackup takes a base backup of a running PostgreSQL server.

Usage:
  pg_basebackup [OPTION]...

Options controlling the output:
  -D, --pgdata= DIRECTORY receive base backup into directory
  -F, --format= plt      output format (plain (default), tar)
  -r, --max-rate=RATE   maximum transfer rate to transfer data directory
                        (in kB/s, or use suffix "K" or "M")
  -R, --write-recovery-conf
                        write configuration for replication
  -T, --tablespace-mapping=OLDDIR=NEWDIR
                        relocate tablespace in OLDDIR to NEWDIR
  --waldir=WALDIR      location for the write-ahead log directory
  -X, --wal-method=none|fetch|stream
                        include required WAL files with specified method
  -z, --gzip            compress tar output
  -Z, --compress=0-9    compress tar output with given compression level

General options:
  -c, --checkpoint=fast|spread
                        set fast or spread checkpointing
  -C, --create-slot    create replication slot
  -l, --label=LABEL     set backup label
  -n, --no-clean        do not clean up after errors
  -N, --no-sync         do not wait for changes to be written safely to disk
  -P, --progress        show progress information
  -S, --slot=SLOTNAME   replication slot to use
  -v, --verbose         output verbose messages
  -V, --version         output version information, then exit
  --manifest-checksums=SHA{224,256,384,512}|CRC32C|NONE
                        use algorithm for manifest checksums
  --manifest-force-encode
                        hex encode all file names in manifest
  --no-estimate-size   do not estimate backup size in server side
  --no-manifest        suppress generation of backup manifest
  --no-slot             prevent creation of temporary replication slot
  --no-verify-checksums
                        do not verify checksums
  -?, --help            show this help, then exit

Connection options:
  -d, --dbname=CONNSTR connection string
  -h, --host=HOSTNAME  database server host or socket directory
  -p, --port=PORT       database server port number
  -s, --status-interval=INTERVAL
                        time between status packets sent to server (in seconds)
  -U, --username=NAME   connect as specified database user
  -w, --no-password    never prompt for password
  -W, --password        force password prompt (should happen automatically)

Report bugs to <pgsql-bugs@lists.postgresql.org>.
PostgreSQL home page: <https://www.postgresql.org/>
postgres@ip-172-31-27-8:~$ 
```

Figure 7.2: Options in pg_basebackup

Point in time Recovery/Archival

Point in Time Recovery, or PITR in short, can also be achieved in postgres by enabling archiving. In simple words, one can restore the database to a specific time and recover the database. To do the same, WAL/archives must be restored till the specific time frame.

The following parameters must be enabled in **postgresql.conf** file to achieve PITR: (Reference: Point In Time Recovery)

```
archive_mode=on
```

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/
server/archivedir/%f' # Unix
```

```
archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"' # Windows
```

We will see in the restore section later in this chapter on how to restore the database to a specific time frame.

Pros and cons of physical backup

- The major advantage is backup contains a filesystem-level backup of the data directory, which in turn also contains the backup of configuration files like **postgresql.conf**, **pg_hba.conf**, and the like. It is as simple as copying the data directory along with custom tablespaces to the destination server and starting the postgres service.
- The major disadvantage is that it can only take cluster-level backup. Backup for individual DB or single table is not possible.
- It is difficult to find the consistency of the backup.
- The speed of the data to be copied depends on the server hardware and the network speed (in case data is copied over the network to a remote server).

Logical backup

Logical backup is used to take a backup of the database in a plain human-readable format. It can also be used to take backup of the specific DB object, like only a particular DB from the whole DB cluster or the single/multiple tables. The PostgreSQL community has given multiple utilities which can help in taking logical backups of the PostgreSQL DB. Let us try to dig deeper into the most frequently used commands one by one.

pg_dump

pg_dump is one of the most common logical backup utilities that help take backup of a single database. It also can take backup of single or multiple tables and sequences. One can take backup in plain text format and compressed format.

pg_dump can also be used as a tool to export specific DB Objects in “**.sql**” file. File extracted in **.sql** format can be imported using the **psql** command. Any file which has been exported in custom or compressed format can be restored using the **pg_restore** command. Also, it is important to note that this lightweight utility neither blocks readers nor writers while taking backup of any object.

pg_dumpall

pg_dumpall utility is used to take logical backup of the whole database cluster, unlike **pg_dump**, which takes backup of only specific DB or specific DB Object. This utility is majorly used to export cluster-level objects like one can generate a script of global objects - tablespaces, users, and the like. It works exactly the same as **pg_dump**, except it works for cluster-level scripts/backups. custom/compressed format backups can be restored using **pg_restore**, and **.sql** (plain text file) format files can be executed using the **psql** command.

Figure 7.3 shows the options which can be used with **pg_dump**.

```

postgres@ip-172-31-27-8:~$ pg_dump --help
pg_dump dumps a database as a text file or to other formats.

Usage:
  pg_dump [OPTION]... [DBNAME]

General options:
  -f, --file=FILENAME      output file or directory name
  -F, --format=clditlp     output file format (custom, directory, tar,
                           plain text <default>)
  -j, --jobs=NUM            use this many parallel jobs to dump
  -v, --verbose             verbose mode
  -V, --version              output version information, then exit
  -Z, --compress=0~9        compression level for compressed formats
  --lock-wait-timeout=TIMEOUT
  --no-sync                 fail after waiting TIMEOUT for a table lock
  --no-sync                  do not wait for changes to be written safely to disk
  -?, --help                  show this help, then exit

Options controlling the output content:
  -a, --data-only           dump only the data, not the schema
  -b, --blobs                include large objects in dump
  -B, --no-blobs              exclude large objects in dump
  -c, --clean                 clean (drop) database objects before recreating
  -C, --create                 include commands to create database in dump
  -e, --extension=PATTERN    dump the specified extension(s) only
  -E, --encoding=ENCODING    dump the data in encoding ENCODING
  -n, --schema=PATTERN       dump the specified schema(s) only
  -N, --exclude-schema=PATTERN
  -O, --no-owner               skip restoration of object ownership in
                               plain-text format
  -s, --schema-only          dump only the schema, no data
  -S, --superuser=NAME        superuser user name to use in plain-text format
  -t, --table=PATTERN         dump the specified table(s) only
  -T, --exclude-table=PATTERN
  -x, --no-privileges        do NOT dump the specified table(s)
  --binary-upgrade           do not dump privileges (grant/revoke)
  --column-inserts            for use by upgrade utilities only
  --disable-dollar-quoting   dump data as INSERT commands with column names
  --disable-triggers          disable dollar quoting, use SQL standard quoting
  --enable-row-security       disable triggers during data-only restore
  --enable-row-security       enable row security (dump only content user has
                             access to)
  --exclude-table-data=PATTERN
  --extra-float-digits=NUM    do NOT dump data for the specified table(s)
  --if-exists                  override default setting for extra_float_digits
  --include-foreign-data=PATTERN
                             use IF EXISTS when dropping objects

                             include data of foreign tables on foreign
                             servers matching PATTERN
  --inserts                   dump data as INSERT commands, rather than COPY
  --load-via-partition-root   load partitions via the root table
  --no-comments                do not dump comments
  --no-publications           do not dump publications
  --no-security-labels        do not dump security label assignments
  --no-subscriptions          do not dump subscriptions
  --no-synchronized-snapshots  do not use synchronized snapshots in parallel jobs
  --no-tablespaces             do not dump tablespace assignments
  --no-toast-compression      do not dump TOAST compression methods
  --no-unlogged-table-data    do not dump unlogged table data
  --on-conflict-do-nothing    add ON CONFLICT DO NOTHING to INSERT commands
  --quote-all-identifiers     quote all identifiers, even if not key words
  --rows-per-insert=NROWS      number of rows per INSERT; implies --inserts
  --section=SECTION            dump named section (pre-data, data, or post-data)
  --serializable-deferrable   wait until the dump can run without anomalies
  --snapshot=SNAPSHOT          use given snapshot for the dump
  --strict-names                require table and/or schema include patterns to
                               match at least one entity each
  --use-set-session-authorization
                             use SET SESSION AUTHORIZATION commands instead of
                             ALTER OWNER commands to set ownership

Connection options:
  -d, --dbname=DBNAME        database to dump
  -h, --host=HOSTNAME         database server host or socket directory
  -p, --port=PORT              database server port number
  -U, --username=NAME         connect as specified database user
  -w, --no-password            never prompt for password
  -W, --password                force password prompt (should happen automatically)
  --role=ROLENAME              do SET ROLE before dump

If no database name is supplied, then the PGDATABASE environment
variable value is used.

Report bugs to <pgsql-bugs@lists.postgresql.org>.
PostgreSQL home page: <https://www.postgresql.org/>
postgres@ip-172-31-27-8:~$
```

Figure 7.3: Options in pg_dump

Figure 7.4 shows the options used in the `pg_dumpall` command:

```
postgres@ip-172-31-27-8:~$ pg_dumpall --help
pg_dumpall extracts a PostgreSQL database cluster into an SQL script file.

Usage:
  pg_dumpall [OPTION]...

General options:
  -f, --file=FILENAME          output file name
  -v, --verbose                verbose mode
  -V, --version                output version information, then exit
  --lock-wait-timeout=TIMEOUT  fail after waiting TIMEOUT for a table lock
  -?, --help                   show this help, then exit

Options controlling the output content:
  -a, --data-only              dump only the data, not the schema
  -c, --clean                  clean (drop) databases before recreating
  -E, --encoding=ENCODING      dump the data in encoding ENCODING
  -g, --globals-only           dump only global objects, no databases
  -O, --no-owner               skip restoration of object ownership
  -r, --roles-only             dump only roles, no databases or tablespaces
  -s, --schema-only            dump only the schema, no data
  -S, --superuser=NAME         superuser user name to use in the dump
  -t, --tablespaces-only       dump only tablespaces, no databases or roles
  -x, --no-privileges          do not dump privileges (grant/revoke)
  --binary-upgrade            for use by upgrade utilities only
  --column-inserts             dump data as INSERT commands with column names
  --disable-dollar-quoting     disable dollar quoting, use SQL standard quoting
  --disable-triggers           disable triggers during data-only restore
  --exclude-database=PATTERN   exclude databases whose name matches PATTERN
  --extra-float-digits=NUM     override default setting for extra_float_digits
  --if-exists                 use IF EXISTS when dropping objects
  --inserts                   dump data as INSERT commands, rather than COPY
  --load-via-partition-root   load partitions via the root table
  --no-comments               do not dump comments
  --no-publications           do not dump publications
  --no-role-passwords         do not dump passwords for roles
  --no-security-labels        do not dump security label assignments
  --no-subscriptions          do not dump subscriptions
  --no-sync                   do not wait for changes to be written safely to disk
  --no-tablespaces             do not dump tablespace assignments
  --no-toast-compression      do not dump TOAST compression methods
  --no-unlogged-table-data    do not dump unlogged table data
  --on-conflict-do-nothing    add ON CONFLICT DO NOTHING to INSERT commands
  --quote-all-identifiers     quote all identifiers, even if not key words
  --rows-per-insert=NROWS      number of rows per INSERT; implies --inserts
  --use-set-session-authorization use SET SESSION AUTHORIZATION commands instead of
                                  ALTER OWNER commands to set ownership

Connection options:
  -d, --dbname=CONNSTR         connect using connection string
  -h, --host=HOSTNAME           database server host or socket directory
  -l, --database=DBNAME         alternative default database
  -p, --port=PORT                database server port number
  -U, --username=NAME           connect as specified database user
  -w, --no-password              never prompt for password
  -W, --password                force password prompt (should happen automatically)
  --role=ROLENAME                do SET ROLE before dump

If -f--file is not used, then the SQL script will be written to the standard
output.

Report bugs to <pgsql-bugs@lists.postgresql.org>.
PostgreSQL home page: <https://www.postgresql.org/>
postgres@ip-172-31-27-8:~$
```

Figure 7.4: Options in `pg_dumpall`

Pros and cons of logical backup

- Specific parts of the database can be backed up with the help of logical backup.
- It is a simple and human-readable backup.
- Since it allows taking the backup in the plain text/SQL file, backup taken in one OS can easily be restored in another OS with the least minimal issues, unlike Physical backup.
- One needs to take a backup of configuration files separately as logical backup does not include the same as physical backup.
- Since a single DB object can be taken, it is faster than a physical backup.
- PITR is not supported in Logical backup.

Restore

Taking a backup is less than half of the job done. An important aspect of the backup is that it should get successfully restored. Backups can also be used to perform PITR or create copies of the database in another server and so on.

Same as backups, there are two different types of restore, as mentioned in *Figure 7.5*:

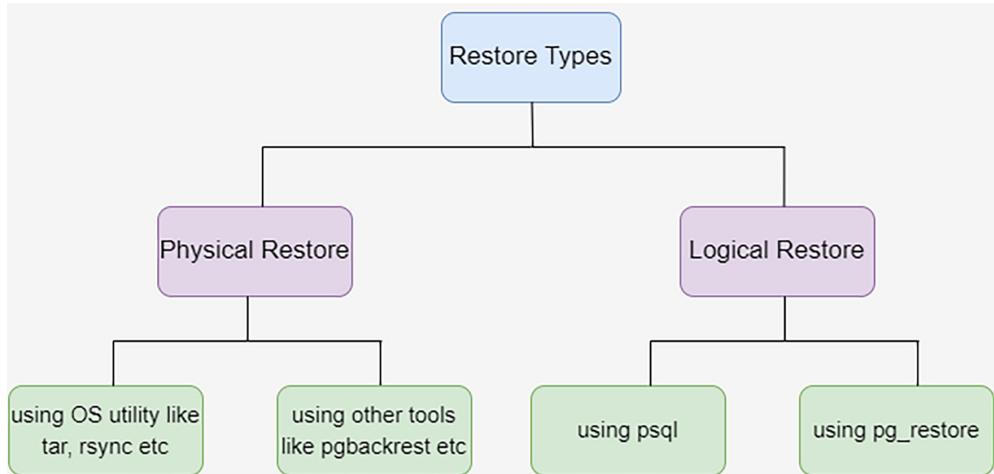


Figure 7.5: Restore Types

psql

psql command is used to restore the backup, which is taken in “**.sql**” format or plain text format. This utility can be compared to the *import* utility of any other

RDBMS, which helps execute the “**.sql**” files. The other use of **psql** is to log in to the **postgresql** database.

Figure 7.6 shows the various options which **psql** command offers:

```
ubuntu@ip-172-31-28-98:~$ psql --help
psql is the PostgreSQL interactive terminal.

Usage:
  psql [OPTION]... [DBNAME [USERNAME]]

General options:
  -c, --command=COMMAND      run only single command (SQL or internal) and exit
  -d, --dbname=DBNAME        database name to connect to (default: "ubuntu")
  -f, --file=FILENAME        execute commands from file, then exit
  -l, --list                  list available databases, then exit
  -v, --set=, --variable=NAME=VALUE
                             set psql variable NAME to VALUE
                             (e.g., -v ON_ERROR_STOP=1)
  -V, --version               output version information, then exit
  -X, --no-psqlrc             do not read startup file (~/.psqlrc)
  -1 ("one"), --single-transaction
                             execute as a single transaction (if non-interactive)
  -?, --help[=options]
    --help=commands            list backslash commands, then exit
    --help=variables           list special variables, then exit

Input and output options:
  -a, --echo-all              echo all input from script
  -b, --echo-errors           echo failed commands
  -e, --echo-queries          echo commands sent to server
  -E, --echo-hidden           display queries that internal commands generate
  -L, --log-file=FILENAME     send session log to file
  -n, --no-readline           disable enhanced command line editing (readline)
  -o, --output=FILENAME       send query results to file (or |pipe)
  -q, --quiet                 run quietly (no messages, only query output)
  -s, --single-step            single-step mode (confirm each query)
  -S, --single-line            single-line mode (end of line terminates SQL command)

Output format options:
  -A, --no-align               unaligned table output mode
  --csv                        CSV (Comma-Separated Values) table output mode
  -F, --field-separator=STRING
                             field separator for unaligned output (default: "|")
  -H, --html                    HTML table output mode
  -P, --pset=VAR[=ARG]          set printing option VAR to ARG (see \pset command)
  -R, --record-separator=STRING
                             record separator for unaligned output (default: newline)
  -t, --tuples-only            print rows only
  -T, --table-attr=TEXT         set HTML table tag attributes (e.g., width, border)
  -x, --expanded                turn on expanded table output
  -z, --field-separator-zero   set field separator for unaligned output to zero byte
  -0, --record-separator-zero  set record separator for unaligned output to zero byte

Connection options:
  -h, --host=HOSTNAME          database server host or socket directory (default: "/var/run/postgresql")
  -p, --port=PORT               database server port (default: "5432")
  -U, --username=USERNAME       database user name (default: "ubuntu")
  -w, --no-password             never prompt for password
  -W, --password                force password prompt (should happen automatically)

For more information, type "\?" (for internal commands) or "\help" (for SQL
commands) from within psql, or consult the psql section in the PostgreSQL
documentation.

Report bugs to <pgsql-bugs@lists.postgresql.org>.
PostgreSQL home page: <https://www.postgresql.org/>
```

Figure 7.6: options in psql

pg_restore

pg_restore helps in restoring the logical backups taken in a custom format. Please find the attributes used by the **pg_restore** command in *Figure 7.7*:

```
postgres@ip-172-31-27-8:~$ pg_restore --help
pg_restore restores a PostgreSQL database from an archive created by pg_dump.

Usage:
  pg_restore [OPTION]... [FILE]

General options:
  -d, --dbname=NAME      connect to database name
  -f, --file=FILENAME    output file name (- for stdout)
  -F, --format=cldit     backup file format (should be automatic)
  -l, --list              print summarized TOC of the archive
  -v, --verbose           verbose mode
  -V, --version           output version information, then exit
  -?, --help               show this help, then exit

Options controlling the restore:
  -a, --data-only          restore only the data, no schema
  -c, --clean               clean (drop) database objects before recreating
  -C, --create              create the target database
  -e, --exit-on-error      exit on error, default is to continue
  -I, --index=NAME          restore named index
  -j, --jobs=NUM            use this many parallel jobs to restore
  -l, --use-list=FILENAME   use table of contents from this file for
                           selecting/ordering output
  -n, --schema=NAME         restore only objects in this schema
  -N, --exclude-schema=NAME do not restore objects in this schema
  -O, --no-owner             skip restoration of object ownership
  -P, --function=NAME(args) restore named function
  -s, --schema-only         restore only the schema, no data
  -S, --superuser=NAME      superuser user name to use for disabling triggers
  -t, --table=NAME           restore named relation (table, view, etc.)
  -T, --trigger=NAME        restore named trigger
  -x, --no-privileges       skip restoration of access privileges (grant/revoke)
  -1, --single-transaction  restore as a single transaction
  --disable-triggers        disable triggers during data-only restore
  --enable-row-security     enable row security
  --if-exists               use IF EXISTS when dropping objects
  --no-comments              do not restore comments
  --no-data-for-failed-tables do not restore data of tables that could not be
                             created
  --no-publications         do not restore publications
  --no-security-labels      do not restore security labels
  --no-subscriptions        do not restore subscriptions
  --no-tablespaces           do not restore tablespace assignments
  --section=SECTION          restore named section (pre-data, data, or post-data)
  --strict-names             require table and/or schema include patterns to
                             match at least one entity each
  --use-set-session-authorization use SET SESSION AUTHORIZATION commands instead of
                                 ALTER OWNER commands to set ownership

Connection options:
  -h, --host=HOSTNAME       database server host or socket directory
  -p, --port=PORT            database server port number
  -U, --username=NAME        connect as specified database user
  -w, --no-password          never prompt for password
  -W, --password             force password prompt (should happen automatically)
  --role=ROLENAME            do SET ROLE before restore

The options -I, -n, -N, -P, -t, -T, and --section can be combined and specified
multiple times to select multiple objects.

If no input file name is supplied, then standard input is used.

Report bugs to <pgsql-bugs@lists.postgresql.org>.
PostgreSQL home page: <https://www.postgresql.org/>
postgres@ip-172-31-27-8:~$
```

Figure 7.7: Options in pg_restore

Let us see few examples of logical backup and restore. Refer *Table 7.1*:

Description	Command
To dump a database called mydb into a SQL-script file:	<code>pg_dump mydb > "/opt/mydb_dump.sql"</code>
To reload such a script into a (freshly created) database named newdb:	<code>psql -d newdb -f "/opt/mydb_dump.sql"</code>
To dump a database into a custom-format archive file:	<code>pg_dump -Fc mydb > "/opt/mydb_dump.dump"</code>
To reload an archive file into a (freshly created) database named newdb:	<code>pg_restore -d newdb "/opt/mydb_dump.dump"</code>
To dump a single table named mytab	<code>pg_dump -t mytab mydb > "/opt/mydb_dump.sql"</code>
To dump the database with the log file using user test	<code>pg_dump -U test -v -f "/opt/mydb_dump.sql" mydb 2>> "/opt/mydb_log.log"</code> password for user test :
To dump only structure without data	<code>pg_dump -sU test -v -f "/opt/mydb_dump.sql" mydb 2>> "/opt/mydb_log.log"</code> password for user test :
To take insert scripts of particular table	<code>pg_dump --column-inserts -a -t mytab -U test mydb > "/opt/mytab_inserts.sql"</code>
To dump only specific tables with data	<code>pg_dump -U test -n schema1 -t BCL_* -f "/opt/BCL_TABLES.sql" mydb</code>

Table 7.1: Logical Backup and Restore example

Useful backup and restore tools

Now that we know the importance of backups when working with PostgreSQL and how they are taken and restored, we will review some of the most popular backup tools we can find and use.

PostgreSQL, directly out of the box, provides the necessary interfaces and functions to take any of the two types of backups, logical (`pg_dump`, `pg_dumpall`) and physical (`pg_basebackup`). Also, it is possible to perform continuous archiving (`archive_command`). However, due to the extraordinary community behind postgres and the continuous commitment to improving the existing options, we have many different tools that can help to create and manage better and more flexible backup strategies.

Some of these tools provide the next benefits:

- Full, incremental, and differential backups.
- Backup compression and checksum.
- Different backup repositories, such as on-prem storage and cloud repositories.
- Parallelism.
- Backups catalog.
- Alerting and notifications.

Then let us review some of the most useful open-source tools, their options, and how to perform a backup and restore it.

pgBackRest

This is one of the most popular and robust backup tools nowadays. You can configure it as a central repository to handle multiple databases backup. It can store the backups in a given path in the backup server or use a cloud bucket for the same; also, you can configure it to back up to both locations simultaneously.

With pgBackRest, you can perform **full, incremental, and differential** backups. You can set the backup operation to run with parallelism, so if your system has multiple CPUs, you can take advantage of them and speed up the backup and restore. Also, you can configure it to take the backup from a standby rather than the primary server, avoiding adding load to the primary server.

Additionally, the continuous archiving can be set as *asynchronous*, meaning pgBackRest will confirm the **postgres** database the WAL was successfully archived, but the actual archive (copying the WAL file to its final destination) happens sometime after. This improves the archive rate and releases some overhead from the database when the WAL creation rate is high due to the heavy workload.

Figure 7.8 shows an example considering two database servers and one backup server running **pgBackRest** and using a cloud bucket (AWS S3) as a repository to store the backups.

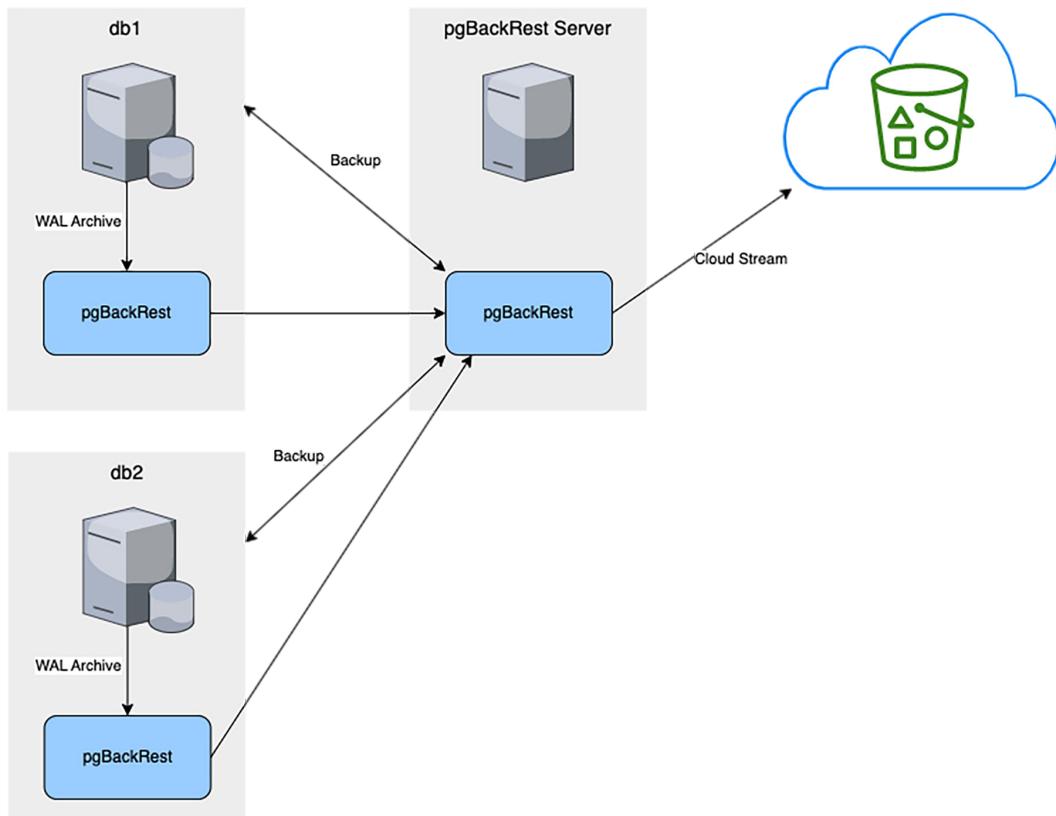


Figure 7.8: pgBackRest for multiple DB servers and using cloud storage

To get this set up working, it is necessary to configure **pgBackRest** in the backup server and the database server because the WAL archiving happens on the database side, so it is necessary to configure PostgreSQL to use **pgBackRest** when a WAL file needs to be archived.

The following is an example of what the **pgBackRest** configuration looks like for the setup in *Figure 7.8*. Remember, the password-less SSH should be configured between the **pgBackRest** server and the database servers:

```
postgres@repository:~$ ssh db1 'hostname'
db1

postgres@repository:~$ ssh db2 'hostname'
db2
```

pgBackRest server:

```
[root@repository ~]# cat /etc/pgbackrest.conf
[global]
repo1-path=/pgbackrest
repo1-retention-full=2
start-fast=y
repo1-type=s3
repo1-s3-bucket=pgbackrest-backups
repo1-s3-key=<aws access key>
repo1-s3-key-secret=<aws secret key>
repo1-s3-token=<aws session token>

[db1]
pg1-path=/var/lib/postgresql/13/main
pg1-host=db1
pg1-host-user=postgres

[db2]
pg1-path=/var/lib/postgresql/14/main
pg1-host=db2
pg1-host-user=postgres
```

db1 server:

```
[root@db1 ~]# cat /etc/pgbackrest.conf
[global]
log-level-file=detail
repo1-host=repository
repo1-host-user=postgres
```

```
[db1]  
pg1-path=/var/lib/postgresql/13/main
```

db2 server:

```
[root@db2 ~]# cat /etc/pgbackrest.conf  
[global]  
log-level-file=detail  
repo1-host=repository  
repo1-host-user=postgres
```

```
[db2]  
pg1-path=/var/lib/postgresql/14/main
```

Now you can create the *stanza*, which means the repository for the defined database. In the above example, since we will take backups from two different database servers, we would need to create two stanzas (*db1* and *db2*). *Figure 7.9* illustrates the creation of the stanza for the *db1*:

```
postgres@repository:~$ pgbackrest --stanza=db1 --log-level-console=info stanza-create  
2022-11-15 22:07:40.185 P00  INFO: stanza-create command begin 2.41: --exec-id=8303-e6b7ee75 --log-level-console=info --pg1-host=db1  
--pg1-host-user=postgres --pg1-path=/var/lib/postgresql/13/main --repo1-path=/pgbackrest --repo1-s3-bucket=pgbackrest-backups --rep  
o1-s3-endpoint=s3.amazonaws.com --repo1-s3-key=<redacted> --repo1-s3-key-secret=<redacted> --repo1-s3-region= --rep  
o1-s3-token=<redacted> --repo1-type=s3 --stanza=db1  
2022-11-15 22:07:41.504 P00  INFO: stanza-create for stanza 'db1' on repo1  
2022-11-15 22:07:43.192 P00  INFO: stanza-create command end: completed successfully (3008ms)  
postgres@repository:~$
```

Figure 7.9: pgBackRest stanza creation

Considering we created the stanza for both database servers, we can now get the first backups. By default, **pgBackRest** will get a **full** backup if there is no previous backup set in the stanza, but if it finds an existing **full** backup, then it will take an **incremental** backup unless we use the **--type** flag and specify another backup type (full, differential).

Figures 7.10 the current scenario's backup operations from the first database server:

```
postgres@repository:~$ pgbackrest --stanza=db1 --log-level-console=info backup
2022-11-15 22:55:30.815 P00  INFO: backup command begin 2.41: --exec-id=10577-9694cdb2 --log-level-console=info
--pg1-host=db1 --pg1-host-user=postgres --pg1-path=/var/lib/postgresql/13/main --repo1-path=/pgbackrest --repo1-r
etention-full=2 --repo1-s3-bucket=pgbackrest-backups --repo1-s3-endpoint=s3.           .amazonaws.com --repo1-s3
-key=<redacted> --repo1-s3-key-secret=<redacted> --repo1-s3-region=           --repo1-s3-token=<redacted> --repo1-
type=s3 --stanza=db1 --start-fast
WARN: no prior backup exists, incr backup has been changed to full
2022-11-15 22:55:32.313 P00  INFO: execute non-exclusive pg_start_backup(): backup begins after the requested im
mediate checkpoint completes
2022-11-15 22:55:32.839 P00  INFO: backup start archive = 000000010000000000000001A, lsn = 0/1A000028
2022-11-15 22:55:32.839 P00  INFO: check archive for prior segment 0000000100000000000000019
2022-11-15 22:58:17.586 P00  INFO: execute non-exclusive pg_stop_backup() and wait for all WAL segments to archi
ve
2022-11-15 22:58:17.793 P00  INFO: backup stop archive = 000000010000000000000001A, lsn = 0/1A000138
2022-11-15 22:58:18.282 P00  INFO: check archive for segment(s) 000000010000000000000001A:000000010000000000000001
A
2022-11-15 22:58:19.693 P00  INFO: new backup label = 20221115-225454F
2022-11-15 22:58:21.811 P00  INFO: full backup size = 479.5MB, file total = 1233
2022-11-15 22:58:21.811 P00  INFO: backup command end: completed successfully (170997ms)
2022-11-15 22:58:21.811 P00  INFO: expire command begin 2.41: --exec-id=10577-9694cdb2 --log-level-console=info
--repo1-path=/pgbackrest --repo1-retention-full=2 --repo1-s3-bucket=pgbackrest-backups --repo1-s3-endpoint=s3.
           .amazonaws.com --repo1-s3-key=<redacted> --repo1-s3-key-secret=<redacted> --repo1-s3-region=
--repo1-s3-token=<redacted> --repo1-type=s3 --stanza=db1
2022-11-15 22:58:22.335 P00  INFO: expire command end: completed successfully (524ms)
postgres@repository:~$
```

Figure 7.10: pgBackRest backup for database server db1

Figure 7.11 show the current scenario's backup operations from the second database server:

```
postgres@repository:~$ pgbackrest --stanza=db2 --log-level-console=info backup
2022-11-15 22:59:19.516 P00  INFO: backup command begin 2.41: --exec-id=11028-bdc7d9ab --log-level-console=info
--pg1-host=db2 --pg1-host-user=postgres --pg1-path=/var/lib/postgresql/14/main --repo1-path=/pgbackrest --repo1-r
etention-full=2 --repo1-s3-bucket=pgbackrest-backups --repo1-s3-endpoint=s3.           .amazonaws.com --repo1-s3
-key=<redacted> --repo1-s3-key-secret=<redacted> --repo1-s3-region=           --repo1-s3-token=<redacted> --repo1-
type=s3 --stanza=db2 --start-fast
WARN: no prior backup exists, incr backup has been changed to full
2022-11-15 22:59:21.570 P00  INFO: execute non-exclusive pg_start_backup(): backup begins after the requested im
mediate checkpoint completes
2022-11-15 22:59:22.094 P00  INFO: backup start archive = 000000010000000000000001A, lsn = 0/1A000028
2022-11-15 22:59:22.094 P00  INFO: check archive for prior segment 0000000100000000000000019
2022-11-15 23:02:11.363 P00  INFO: execute non-exclusive pg_stop_backup() and wait for all WAL segments to archi
ve
2022-11-15 23:02:11.570 P00  INFO: backup stop archive = 000000010000000000000001A, lsn = 0/1A000138
2022-11-15 23:02:12.058 P00  INFO: check archive for segment(s) 000000010000000000000001A:000000010000000000000001
A
2022-11-15 23:02:13.583 P00  INFO: new backup label = 20221115-225932F
2022-11-15 23:02:15.794 P00  INFO: full backup size = 482MB, file total = 1253
2022-11-15 23:02:15.795 P00  INFO: backup command end: completed successfully (176280ms)
2022-11-15 23:02:15.795 P00  INFO: expire command begin 2.41: --exec-id=11028-bdc7d9ab --log-level-console=info
--repo1-path=/pgbackrest --repo1-retention-full=2 --repo1-s3-bucket=pgbackrest-backups --repo1-s3-endpoint=s3.
           .amazonaws.com --repo1-s3-key=<redacted> --repo1-s3-key-secret=<redacted> --repo1-s3-region=
--repo1-s3-token=<redacted> --repo1-type=s3 --stanza=db2
2022-11-15 23:02:16.358 P00  INFO: expire command end: completed successfully (563ms)
postgres@repository:~$
```

Figure 7.11: pgBackRest backup for database server db2.

With **pgBackRest**, you can also get quick information about the existing backups in the repository by consulting the catalog per *stanza*. *Figure 7.12* demonstrates how the catalog looks for the *db1* stanza after the **full** backup:

```
postgres@repository:~$ pgbackrest --stanza=db1 --log-level-console=info info
stanza: db1
  status: ok
  cipher: none

  db (current)
    wal archive min/max (13): 000000010000000000000018/000000010000000000000001A

    full backup: 20221115-225454F
      timestamp start/stop: 2022-11-15 22:54:54 / 2022-11-15 22:57:39
      wal start/stop: 000000010000000000000001A / 000000010000000000000001A
      database size: 479.5MB, database backup size: 479.5MB
      repo1: backup set size: 27.8MB, backup size: 27.8MB
postgres@repository:~$
```

Figure 7.12: pgBackRest backup catalog.

Now, let us think that for any reason, we lost the *db1* server, there was an issue with the storage, the data center was lost, or somebody mistakenly deleted all the contents from the **data_directory** (ouch). In this situation, we can recover our system by restoring one of our backups. *Figures 7.13, 7.14, and 7.15* illustrates the deletion of the database data, the restore operation, and the again working system.

```
postgres@db1:~$ du -sch /var/lib/postgresql/13/main
513M  /var/lib/postgresql/13/main
513M  total
postgres@db1:~$ psql -c'\l+ pgbench'
          List of databases
   Name   | Owner   | Encoding | Collate | Ctype | Access privileges | Size | Tablespace | Description
   pgbench | pgbench | UTF8     | C.UTF-8 | C.UTF-8 |              | 456 MB | pg_default | 
(1 row)

postgres@db1:~$ rm -rf /var/lib/postgresql/13/main/*
postgres@db1:~$ du -sch /var/lib/postgresql/13/main
4.0K   /var/lib/postgresql/13/main
4.0K   total
postgres@db1:~$ psql -c'\l+ pgbench'
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed: FATAL:  could not open file "global/pg_filenode.map": No such file or directory
postgres@db1:~$
```

Figure 7.13: Deletion of the database data in the data_directory.

We can recover the **full** backup easily using the pgBackRest configuration we already have. *Figure 7.14* shows the process:

```
postgres@db1:~$ pgbackrest --log-level-console=info --stanza=db1 restore
2022-11-15 23:39:32.034 P00  INFO: restore command begin 2.41: --exec-id=15929-4f9f0233 --log-level-console=info --log-level-file=detail --pg1-path=/var/lib/postgresql/13/main --repo1-host=repository --repo1-host-user=postgres --stanza=db1
2022-11-15 23:39:33.658 P00  INFO: repo1: restore backup set 20221115-225454F, recovery will start at 2022-11-15 22:54:54
2022-11-15 23:42:33.552 P00  INFO: write updated /var/lib/postgresql/13/main/postgresql.auto.conf
2022-11-15 23:42:33.577 P00  INFO: restore global/pg_control (performed last to ensure aborted restores cannot be started)
2022-11-15 23:42:33.578 P00  INFO: restore size = 479.5MB, file total = 1233
2022-11-15 23:42:33.578 P00  INFO: restore command end: completed successfully (181545ms)
postgres@db1:~$
```

Figure 7.14: pgBackRest restore of db1.

After recovering the **full** backup, we can verify the size of the **data_directory** and connect using **psql** to check the size of the databases. *Figure 7.15* shows the successful result:

```
postgres@db1:~$ date ; du -sch /var/lib/postgresql/13/main
Tue Nov 15 23:43:57 UTC 2022
512M  /var/lib/postgresql/13/main
512M  total
postgres@db1:~$ psql -c'\l+ pgbench'
          List of databases
   Name   | Owner    | Encoding | Collate | Ctype | Access privileges | Size | Tablespace | Description
   +-----+-----+-----+-----+-----+-----+-----+-----+
pgbench | pgbench | UTF8  | C.UTF-8 | C.UTF-8 |           | 456 MB | pg_default | 
(1 row)

postgres@db1:~$
```

Figure 7.15: PostgreSQL restored from the pgBackRest backup.

Remember, we can perform such full restorations by taking the existing backups and applying the archived WAL files until the latest. But also possible to restore to a specific point in time if we specify the **--type=time** flag and specify a target timestamp.

These and many other options are supported with **pgBackRest**. Worth to take some time and reviewing the official documentation at <https://pgbackrest.org/>.

Barman

Barman is another open-source tool that has been around for a while and has gotten the attention of many projects and solutions. The same as pgBackRest, Barman, can be used as a central repository to back up multiple database servers. It creates and maintains a backup catalog to get a quick insight into the existing backups.

Barman can produce **full** backups, and when configured as **rsync** mode, it can get **incremental** backups, run with parallelism, and ensure data deduplication and network compression to speed up the transmission when taking the backup remotely.

Barman does not support cloud repositories as storage, it can just use local filesystems. However, for continuous archiving, it supports the standard method (copy or file transmission) or WAL streaming, just as a standby replica. Also, you can use custom *hook scripts* to execute customized routines before or after each backup or WAL archive.

Figure 7.16 illustrates a Barman solution to back up two different PostgreSQL servers.

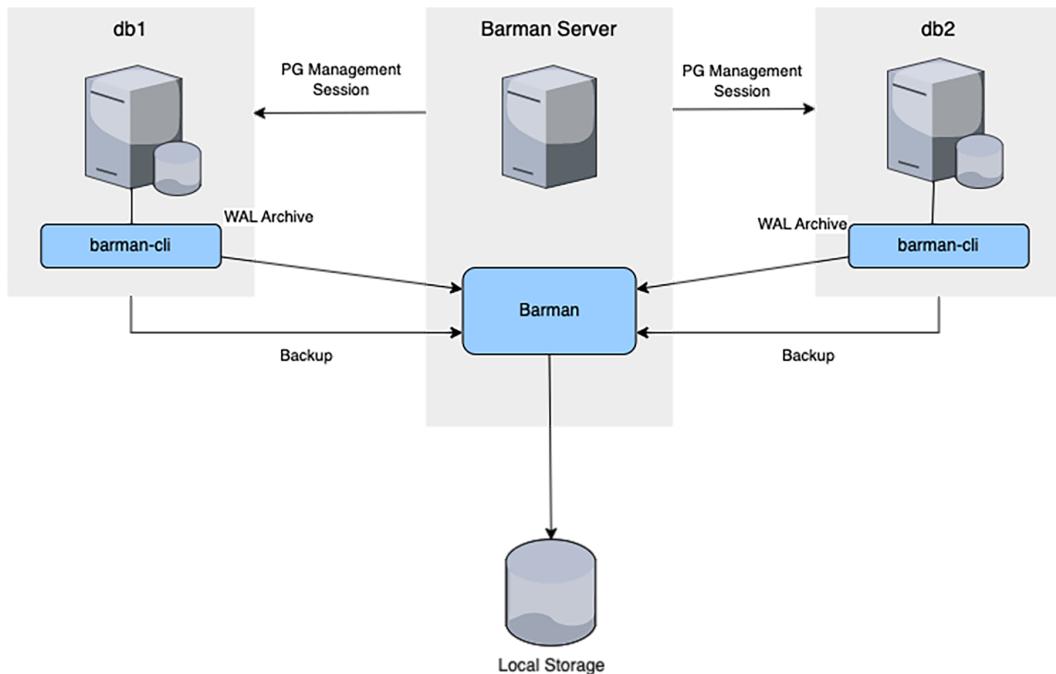


Figure 7.16: Barman for multiple database servers backup.

Using Barman, you need to consider installing the **barman-cli** in the database nodes; this provides an interface for the **archive_command**, so PostgreSQL can push the WAL files for archiving. All the other backup execution happens in the backup server. Also, consider adding the proper configuration to the PostgreSQL servers so the Barman server can connect to **postgres** itself and the Operating System.

The following is an example of the required configuration.

Barman server:

```
barman@repository:~$ cat /etc/barman.conf

[barman]
barman_user = barman
configuration_files_directory = /etc/barman.d
barman_home = /var/lib/barman
log_file = /var/log/barman/barman.log
log_level = INFO
```

```
compression = gzip
```

```
barman@repository:~$ cat /etc/barman.d/ssh-db[1,2].conf
```

```
[ssh-db1]
```

```
description = "PostgreSQL Database db1 (via SSH)"
```

```
ssh_command = ssh postgres@db1
```

```
conninfo = host=db1 user=barman dbname=postgres
```

```
backup_method = rsync
```

```
reuse_backup = link
```

```
backup_options = concurrent_backup
```

```
archiver = on
```

```
-----
```

```
[ssh-db2]
```

```
description = "PostgreSQL Database db2 (via SSH)"
```

```
ssh_command = ssh postgres@db2
```

```
conninfo = host=db2 user=barman dbname=postgres
```

```
backup_method = rsync
```

```
reuse_backup = link
```

```
backup_options = concurrent_backup
```

```
archiver = on
```

db1 and db2 servers:

```
postgres@db1:~$ createuser -s -P barman
```

To get this configuration working you need to allow passwordless SSH from the Barman server to the database servers, from the **barman** user (it runs the barman executable) to the **postgres** user (it runs the postgres service).

```
barman@repository:~$ ssh postgres@db1 'hostname'
```

```
db1
```

```
barman@repository:~$ ssh postgres@db2 'hostname'
db2
```

With all the previous configuration in place, you can get verify all is correctly set and then get the first backups. *Figure 7.17* shows how to verify the configuration from the Barman server for a given database node.

```
barman@repository:~$ barman check ssh-db1
Server ssh-db1:
    PostgreSQL: OK
    superuser or standard user with backup privileges: OK
    wal_level: OK
    directories: OK
    retention policy settings: OK
    backup maximum age: OK (no last_backup_maximum_age provided)
    backup minimum size: OK (0 B)
    wal maximum age: OK (no last_wal_maximum_age provided)
    wal size: OK (0 B)
    compression settings: OK
    failed backups: OK (there are 0 failed backups)
    minimum redundancy requirements: OK (have 0 backups, expected at least 0)
    ssh: OK (PostgreSQL server)
    systemid coherence: OK (no system Id stored on disk)
    archive_mode: OK
    archive_command: OK
    continuous archiving: OK
    archiver errors: OK
barman@repository:~$
```

Figure 7.17: Barman configuration check.

If all the previous checks are OK, you can take a backup. See *Figure 7.18*, which illustrates a backup operation using Barman:

```
barman@repository:~$ barman backup ssh-db1
Starting backup using rsync-concurrent method for server ssh-db1 in /var/lib/barman/ssh-db1/base/20221116T040407
Backup start at LSN: 0/2C000028 (00000030000000000000002C, 00000028)
This is the first backup for server ssh-db1
WAL segments preceding the current backup have been found:
    000000300000000000000018 from server ssh-db1 has been removed
    00000030000000000000001C from server ssh-db1 has been removed
    00000030000000000000001D from server ssh-db1 has been removed
    00000030000000000000001E from server ssh-db1 has been removed
    00000030000000000000001F from server ssh-db1 has been removed
    000000300000000000000020 from server ssh-db1 has been removed
    000000300000000000000021 from server ssh-db1 has been removed
    000000300000000000000022 from server ssh-db1 has been removed
    000000300000000000000023 from server ssh-db1 has been removed
    000000300000000000000024 from server ssh-db1 has been removed
    000000300000000000000025 from server ssh-db1 has been removed
    000000300000000000000026 from server ssh-db1 has been removed
    000000300000000000000027 from server ssh-db1 has been removed
    000000300000000000000028 from server ssh-db1 has been removed
    000000300000000000000029 from server ssh-db1 has been removed
    00000030000000000000002A from server ssh-db1 has been removed
Starting backup copy via rsync/SSH for 20221116T040407
Copy done (time: 18 seconds)
This is the first backup for server ssh-db1
Asking PostgreSQL server to finalize the backup.
Backup size: 484.2 MiB. Actual size on disk: 484.2 MiB (-0.00% deduplication ratio).
Backup end at LSN: 0/2C000138 (00000030000000000000002C, 00000138)
Backup completed (start time: 2022-11-16 04:04:07.591253, elapsed time: 24 seconds)
Processing xlog segments from file archival for ssh-db1
    000000300000000000000028
    00000030000000000000002C
    00000030000000000000002C.00000028.backup
barman@repository:~$
```

Figure 7.18: Barman backup operation

As we stated before, Barman includes a backup catalog so that you can get a quick view of the existing backups for a given database server and the status of the server. *Figure 7.19* shows how this catalog views look.

```
barman@repository:~$ barman list-backups ssh-db1
ssh-db1 20221116T040407 - Wed Nov 16 04:02:36 2022 - Size: 484.2 MiB - WAL Size: 16.0 KiB
barman@repository:~$ barman status ssh-db1
Server ssh-db1:
  Description: PostgreSQL Database db1 (via SSH)
  Active: True
  Disabled: False
  PostgreSQL version: 13.9
  Cluster state: in production
  pgespresso extension: Not available
  Current data size: 484.6 MiB
  PostgreSQL Data directory: /var/lib/postgresql/13/main
  Current WAL segment: 0000000300000000000000000000000E
  PostgreSQL 'archive_command' setting: barman-wal-archive repository ssh-db1 %p
  Last archived WAL: 0000000300000000000000000000000D, at Wed Nov 16 04:02:40 2022
  Failures of WAL archiver: 0
  Server WAL archiving rate: 4.75/hour
  Passive node: False
  Retention policies: not enforced
  No. of available backups: 1
  First available backup: 20221116T040407
  Last available backup: 20221116T040407
  Minimum redundancy requirements: satisfied (1/0)
barman@repository:~$
```

Figure 7.19: Barman backup list and server status.

Obviously, with Barman, we can also recover our databases from a backup. Considering we lost the *db1* server, and we were able to rebuild the machine with the same storage paths but empty. We can restore our data and get postgres up and running, as *Figure 7.20* shows:

```
barman@repository:~$ barman recover --remote-ssh-command "ssh postgres@db1" ssh-db1 20221116T040407 /var/lib/postgresql/13/main
Starting remote restore for server ssh-db1 using backup 20221116T040407
Destination directory: /var/lib/postgresql/13/main
Remote command: ssh postgres@db1
Copying the base backup.
Copying required WAL segments.
Generating archive status files
Identify dangerous settings in destination directory.

IMPORTANT
These settings have been modified to prevent data losses

postgresql.auto.conf line 6: archive_command = false

Recovery completed (start time: 2022-11-17 04:37:46.317480+00:00, elapsed time: 24 seconds)
Your PostgreSQL server has been successfully prepared for recovery!
barman@repository:~$
```

Figure 7.20: Barman recovers from the backup server to the remote *db1* server.

Consider that the operation shown in *Figure 7.20* required the PostgreSQL service on *db1* was stopped. Also, you can see the **IMPORTANT** message, the **archive_command** was set to false, causing any new archive from this just restored server to fail.

This is expected because this restore might happen against a different database server, such as a spare server. In such a case, the database might completely diverge from the original *db1*. We need to configure a new database server in the Barman configuration to continue backing up the “new” *db1*.

```
barman@repository:~$ cat /etc/barman.d/ssh-db1-new.conf
[ssh-db1-new]
description = "PostgreSQL Database db1-new (via SSH)"
ssh_command = ssh postgres@db1
conninfo = host=db1 user=barman dbname=postgres
backup_method = rsync
reuse_backup = link
backup_options = concurrent_backup
archiver = on
```

Barman has been around in the PostgreSQL since a while, so it has been proved and used in a large number of projects. You can find some other features and details we did not cover in this chapter by looking at its documentation at <https://pgbarman.org/documentation/>.

pg_probackup

We might tell **pg_probackup** is the “youngest” tool from this section, just as the previous ones, it is an open-source project. And, like other open-source projects, it brings some extra functions to extend the flexibility for different types of PostgreSQL implementations.

pg_probackup offers interesting options like incremental restore, skipping those data pages already present in the **data_directory** that have not changed from the ones in the backup, automatic and on-demand data consistency checks, backup merge, and deduplication.

It also supports other features similar to the previous tools, like parallelism, remote operations, backup from a standby, backup catalog, and more.

Figure 7.21 shows what a setup of **pg_probackup** backing up two different PostgreSQL servers looks like:

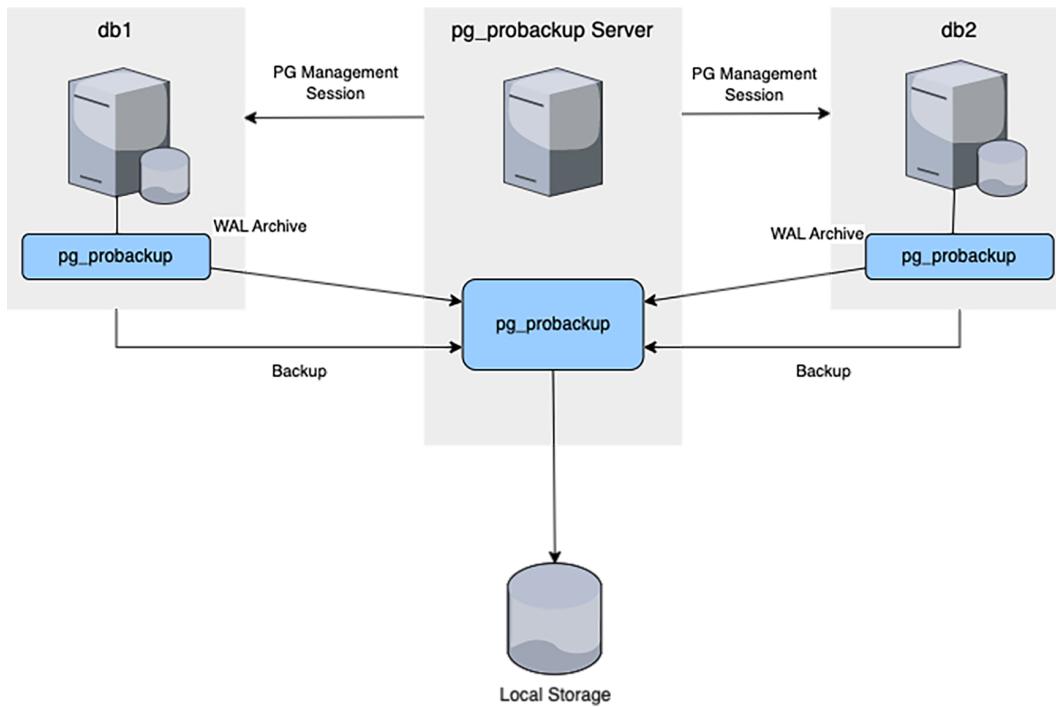


Figure 7.21: pg_probackup for multiple database servers backup.

Different from the previous tools, **pg_probackup** does not use configuration files, or well, it does but accordingly to the documentation, these should not be edited manually. Rather, you need to configure your setup using the different command options included with the tool.

As illustrated in *Figure 7.21*, you need to install the tool in the backup server and the database servers, and you have to install the corresponding package for the PostgreSQL version. Since we are using the same db1 and db2 servers from the previous examples, one is version 13, and the other is version 14, you need to install the packages accordingly, two in the backup server and the corresponding single one in the database servers.

Once installed, the next step is to create the backup repository in the local system of the backup server and initialize the catalog on it. *Figure 7.22* shows this action:

```
postgres@repository:~$ pg_probackup-13 init -B /pg_probackup/db1
INFO: Backup catalog '/pg_probackup/db1' successfully initd
postgres@repository:~$ pg_probackup-14 init -B /pg_probackup/db2
INFO: Backup catalog '/pg_probackup/db2' successfully initd
```

Figure 7.22: pg_probackup catalog init.

The next step is to add the instances to be backed up, in this case, two remote instances. See *Figure 7.23*, which illustrates this point:

```
postgres@repository:~$ pg_probackup-13 add-instance -B /pg_probackup/db1 -D /var/lib/postgresql/13/main --instance db1 --remote-host=db1 --remote-user=postgres --remote-path=/usr/bin
INFO: Instance 'db1' successfully initd
postgres@repository:~$ pg_probackup-14 add-instance -B /pg_probackup/db2 -D /var/lib/postgresql/14/main --instance db2 --remote-host=db2 --remote-user=postgres --remote-path=/usr/bin
INFO: Instance 'db2' successfully initd
postgres@repository:~$ █
```

Figure 7.23: pg_probackup adding remote instances.

Just as with the other two tools, to perform the remote backup operations, we need to configure the password-less SSH between the backup server and the database servers. Also, since the continuous archive is required, we must set the **archive_command** properly. These details are deeply covered in the online documentation of the tool.

Now, considering we have set the previous details, we can get the backup from the remote instances. *Figure 7.24* shows the backup operation from the **db2** instance:

```
postgres@repository:~$ pg_probackup-14 backup -B /pg_probackup/db2 --instance db2 -b FULL
INFO: Backup start, pg_probackup version: 2.5.8, instance: db2, backup ID: RLIT51, backup mode: FULL, wal mode: A
RCHIVE, remote: true, compress-algorithm: none, compress-level: 1
WARNING: This PostgreSQL instance was initialized without data block checksums. pg_probackup have no way to detect data block corruption without them. Reinitialize PGDATA with option '--data-checksums'.
INFO: Database backup start
INFO: wait for pg_start_backup()
INFO: Wait for WAL segment /pg_probackup/db2/wal/db2/000000010000000000000027 to be archived
INFO: PGDATA size: 486MB
INFO: Current Start LSN: 0/27000028, TLI: 1
INFO: Start transferring data files
INFO: Data files are transferred, time elapsed: 16s
INFO: wait for pg_stop_backup()
INFO: pg_stop backup() successfully executed
INFO: stop_lsn: 0/28001E48
INFO: Wait for LSN 0/28001E48 in archived WAL segment /pg_probackup/db2/wal/db2/000000010000000000000028
INFO: Getting the Recovery Time from WAL
INFO: Syncing backup files to disk
INFO: Backup files are synced, time elapsed: 0
INFO: Validating backup RLIT51
INFO: Backup RLIT51 data files are valid
INFO: Backup RLIT51 resident size: 486MB
INFO: Backup RLIT51 completed
postgres@repository:~$ █
```

Figure 7.24: pg_probackup backup from remote PostgreSQL.

With **pg_probackup**, we can check the backup catalog and get information about the existing backup sets. *Figure 7.25* demonstrates this operation:

```
postgres@repository:~$ pg_probackup-14 show -B /pg_probackup/db2
BACKUP INSTANCE 'db2'
=====
Instance Version ID      Recovery Time      Mode  WAL Mode TLI  Time   Data   WAL   Zratio Start LSN  S
top LSN  Status
=====
db2      14    RLIT51  2022-11-18 01:52:15+00  FULL  ARCHIVE 1/0  22s  486MB 16MB  1.00  0/27000028  0
/28001E48 OK
postgres@repository:~$
```

Figure 7.25: pg_probackup backup catalog

Finally, and as expected, we can also recover from a backup in case of a catastrophe, or if we want to create a new server with a copy of the data. See *Figure 7.26*, which illustrates a restore for the **db2** server:

```
postgres@repository:~$ pg_probackup-14 restore -B /pg_probackup/db2 --instance db2
INFO: Validating backup RLIT51
INFO: Backup RLIT51 data files are valid
INFO: Backup RLIT51 WAL segments are valid
INFO: Backup RLIT51 is valid.
INFO: Restoring the database from backup at 2022-11-18 01:51:01+00
INFO: Start restoring backup files. PGDATA size: 486MB
INFO: Backup files are restored. Transferred bytes: 486MB, time elapsed: 18s
INFO: Restore incremental ratio (less is better): 100% (486MB/486MB)
INFO: Syncing restored files to disk
INFO: Restored backup files are synced, time elapsed: 3s
INFO: Restore of backup RLIT51 completed.
postgres@repository:~$
```

Figure 7.26: pg_probackup restore from a backup.

There is no doubt that **pg_probackup** is another great tool we can consider when designing our backup strategies. It has some other features we did not cover in this chapter, take a look at the GitHub to know more on the same.

Conclusion

As we saw, PostgreSQL provides all the required features to protect our databases against losses. We can back up our databases in two ways, logically and physically. There are some tools and techniques already present in the community packages of `postgres`, but also other open-source projects bring extra features.

In the next chapter, we will learn about PostgreSQL replication.

Bibliography

- pg_basebackup: <https://www.postgresql.org/docs/15/app-pgbasebackup.html>
- Point In Time Recovery: <https://www.postgresql.org/docs/15/continuous-archiving.html>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 8

Replicating Data

Introduction

Replicating data is an important feature of any database system, whether structured (Structured Query Language, SQL) just as PostgreSQL, MySQL, or Oracle, or not (NoSQL) as MongoDB or Cassandra, for example. This feature opens the door for valuable solutions such as **High-Availability (HA)**, distributed databases, horizontal scaling, or **Disaster Recovery (DR)** models. As you can imagine, PostgreSQL supports it, and in this chapter, we will learn about the same.

Structure

In this chapter, we will learn about the two main types of data replication supported by PostgreSQL.

- Physical replication
- Logical replication

Objectives

In the content of this chapter, we will review the two main data replication types supported by PostgreSQL. How they work, their advantages and limitations, and

what is required to configure and use them. We will also study some practical examples to understand what we can do with data replication.

Physical replication

We could say that physical replication is the “original” replication method supported by PostgreSQL. It became possible because of a couple of mechanisms we already have reviewed: the **Write-Ahead Log (WAL)** and the continuous archiving; you can look at *Chapter 5, Architecture of PostgreSQL* if you want to refresh the concepts.

The WAL and the continuous archiving were designed and added to postgres to enable the possibility of performing *recovery*. If you remember, PostgreSQL works in memory, all the data changes happen in the shared buffers area, and they are not flushed to disk immediately but asynchronously. So, in the event of a power or hardware failure, these memory changes might get lost. However, the WAL keeps a record of all the transactions that have changed data, which are flushed to the disk on every commit. In the same failure event where all the memory data is lost, PostgreSQL can perform *recovery* and *redo* all the data changes from the records in the WAL. *Figure 8.1* illustrates the recovery process.

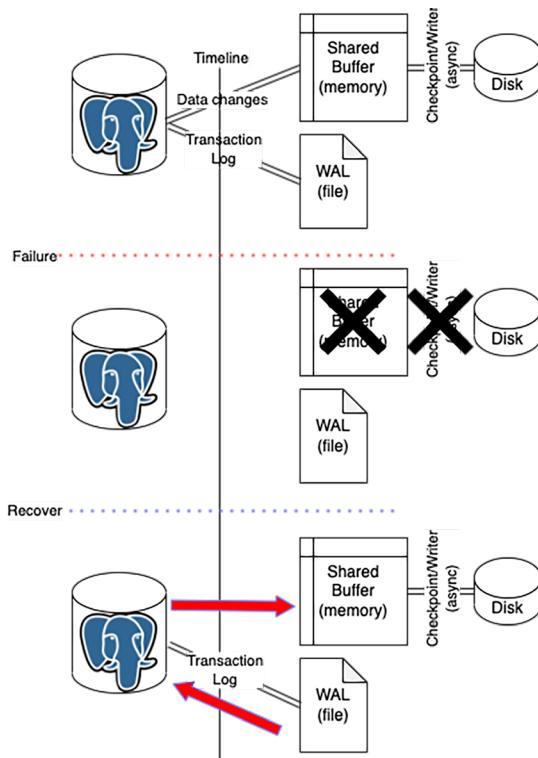


Figure 8.1: PostgreSQL recovery after a failure.

Continuous archiving also brings extra protection since the WAL files may get removed or reused over time, and archiving them to an external or dedicated device covers this situation, then postgres will still be able to recover.

These functionalities are great for getting our system up and running after the failure and, more importantly, with no data loss. But this same logic can be applied in a different server remotely, right? The answer is yes! The same way the data changes are replied to after a crash to *redo* the database state can be done continuously in a remote server on top of a restored physical backup. *Figure 8.2* shows in a very basic way how this looks.

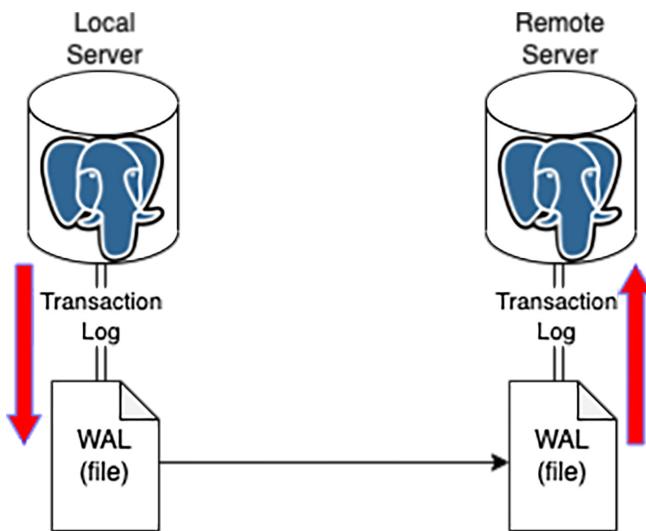


Figure 8.2: PostgreSQL basic physical replication diagram.

So, the physical replication can be done by re-applying the WAL changes from a read/write (primary) instance on top of a physical data copy (backup) in a remote (standby) instance.

As you can think, there are a few requirements to getting this working properly, and there are a couple of *variations* of the physical replication and some features to build different topologies. We will learn about all these details next.

Physical replication requirements:

As we saw in the previous *Chapter 7, Backup and Restore in PostgreSQL*, the physical backups are an exact binary copy of a running PostgreSQL, so they are inconsistent, and we need the WAL to make them consistent during a restore. And, as we stated before, the physical replication is an “extension” of this logic, so we can say the next

are the minimum requirements to build a replica server or standby in postgres argot. Consider the read/write instance or primary already should exist.

- A server with enough resources (CPU and memory). This depends on the replica's usage and whether this is supposed to handle any read workload.
- The server requires enough storage capacity according to the DB size.
- The same filesystem/directory layout as the primary is expected.
- A PostgreSQL server running in the same version as the primary.
- Even when it is not a “must,” using the same OS for the replica as the primary node, including the same kernel version, can prevent unsightly side effects.
- The replica node should be able to directly access the WAL archive repository or the primary server.
- If the WAL archive repository is the same as the backups, then the replica can be built from a backup, and there would not be a need to reach the primary server.

Hot standby

During a physical backup restoration, the data cannot be read. Postgres needs to apply all the required WAL and then “open” the database at a consistent point so it can be accessed for reads and writes.

Since physical replication follows essentially the same principles, just keeping the restore operation going, the same effect of the data not being available for reads is present. PostgreSQL added a feature to turn this “ongoing” recovery into a read-only access instance, so read queries can be executed even when the WAL apply is active. To activate this feature, the parameter **hot_standby** needs to be set to **on** in the **postgresql.conf** configuration file at the replica side.

Once the **hot_standby** parameter is on, the instance will accept read-only queries after reaching a consistent state. During a standby startup, there will be some time when the clients can not connect to the instance. In the database logs, you will see something similar to the next:

```
LOG: entering standby mode
```

```
... then some time later ...
```

```
LOG: consistent recovery state reached
```

```
LOG: database system is ready to accept read-only connections
```

Keep in mind that the operations in both, the primary and the standby, can run into some conflicts. This happens because the data changes on the primary should be re-applied in the standby, but the queries on both can be different. As we saw in *Chapter 6, PostgreSQL Internals*, the MVCC, and the VACUUM are mechanisms used in PostgreSQL to handle the concurrency and perform the cleaning from the *dead tuples*. These operations happen on the primary side and need to be re-applied in the standby, but at some point, a query in the standby might still be reading data that was already deleted from the primary; this situation will prevent the WAL application from continuing.

The WAL application can wait for a while to let the read query on the standby to complete before re-applying the data deletion, but cannot wait indefinitely. To handle this, PostgreSQL supports two parameters: **max_standby_archive_delay** and **max_standby_streaming_delay**. These can control how long the WAL application should wait to let the read query complete. If any query preventing the WAL application from continuing reaches these thresholds, it will be terminated, and the WAL application will resume. This way, the replication stays close to the changes from the primary.

Archive recovery

At this point, we have discussed that physical replication is possible by re-applying the data changes from the WAL from the primary into the instance running as a standby in a remote server. The initial way to do this was to physically copy the WAL files from the primary to the standby; then, the remote instance will take them and re-apply the changes.

In *Chapter 7, Backup & Restore in PostgreSQL*, we reviewed the **archive_command** parameter, which can contain an Operating System command or a script call to execute a routine to archive the WAL files before they get removed or recycled. This usually means copying the WAL from the `postgres` **data_directory** to a dedicated repository called *the archive*.

Well, we can build the standby to get the WAL from the same location, *the archive*, where the primary is archiving them. For this, when configuring the standby, we can set the **restore_command** parameter, which works analogous to the **archive_command**, so it can contain an Operating System command or a script call with the routine for getting the WAL from the archive. The next is an example of how this would look:

```

restore_command = 'cp /mnt/server/archivedir/%f "%p"'

restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"' #Windows

```

Figure 8.3 details these components for a physical replication by archive recovery.

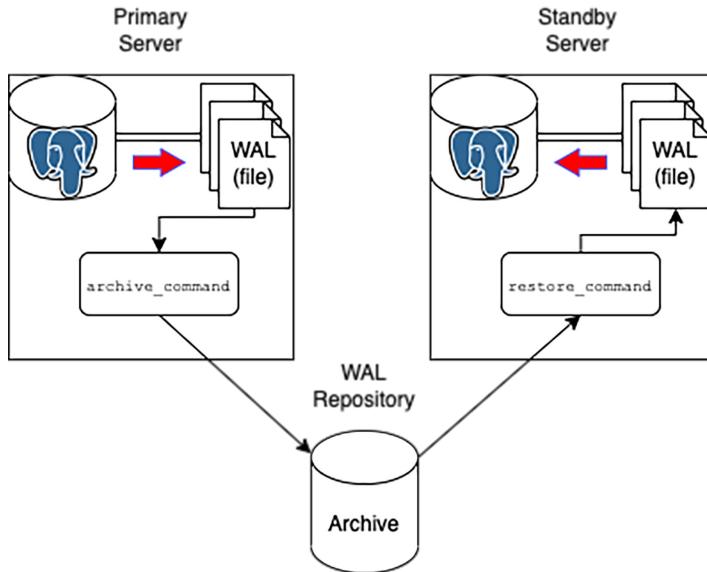


Figure 8.3: Archive recovery physical replication.

This physical replication setup has some advantages, such as it is very flexible, you can easily accommodate the primary and the standby server or servers in different physical locations, and they don't need to "see" each other, having access to the WAL repository is all that they need. But, also has the disadvantage the replication can get too far behind (lag) because of network latency or file transmission issues.

The next is an example of the required configuration to establish the archive recover physical replication.

```

# Primary
wal_level = 'replica'
archive_mode = 'on'
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/
server/archivedir/%f'

# Standby
hot_standby = 'on'

```

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
```

Streaming replication

In some way, the streaming replication is an improvement from the archive recovery setup. The foundations of the replication mechanism are the same; the WAL changes from the primary will be re-applied in the standby.

The big difference is now there is no need to copy the WAL files from one server to the other, but a couple of special processes will take care of streaming the WAL changes directly from the WAL buffer in the primary to the remote process in the standby so it can be directly re-applied. The processes are known as **WAL sender** in the primary and **WAL receiver** in the standby.

The configuration of this setup is very similar to the archive recovery; the difference is we need to set the **primary_conninfo** parameter with the details for the **WAL receiver** to connect to the primary server. The next is an example of how this parameter might look:

```
primary_conninfo = 'host=187.168.1.50 port=5432 user=someuser
password=passwd application_name=s1'
```

Also, the **restore_command** is now optional but still useful if the replication stops for a long period, and to resume, needs to get WAL files already archived from the primary. *Figure 8.4* shows this setup.

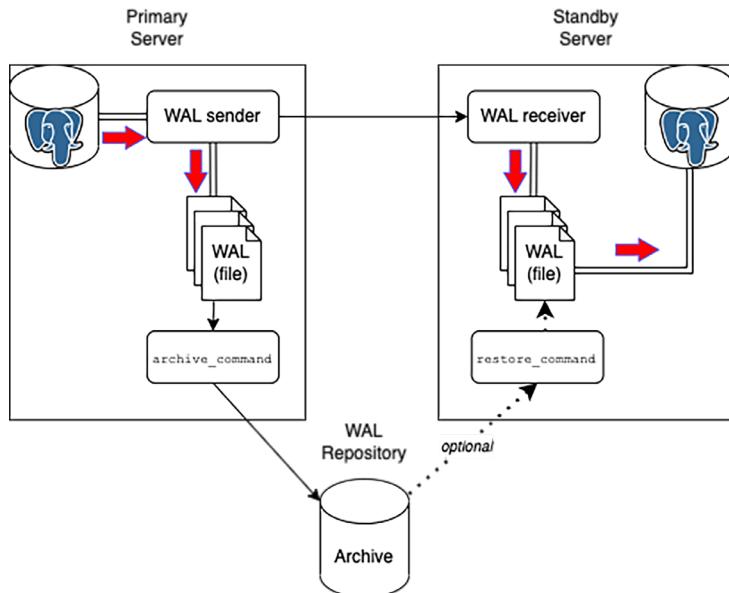


Figure 8.4: Streaming replication.

The streaming replication can keep a nearly real-time sync compared to the archive recovery replication. The direct communication between the **WAL sender** and the **WAL receiver** process saves a lot of time and processing resources. This setup requires proper network configuration to allow communication between the primary and the standby servers.

The next is an example of the configuration parameters to set the streaming replication.

```
# Primary

wal_level = 'replica'

archive_mode = 'on'

archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/
server/archivedir/%f' # optional


# Standby

hot_standby = 'on'

primary_conninfo = 'host=187.168.1.50 port=5432 user=someuser
password=passwd application_name=s1'

restore_command = 'cp /mnt/server/archivedir/%f "%p"' # optional
```

Cascading

Usually, when working with streaming replication, we can take advantage of the almost instant sync status between the primary and the standby servers. Our application design can consider driving the reading load directly to one or more standby; this way, the primary server can fully use its hardware resources to serve the read-write operations for the data changes and save the resources from the read-only operations.

However, in large environments, including multiple standby servers, the primary can suffer from the extra load of many WAL receiver processes. *Figure 8.5* illustrates this setup with multiple standby servers replicating from a single primary server.

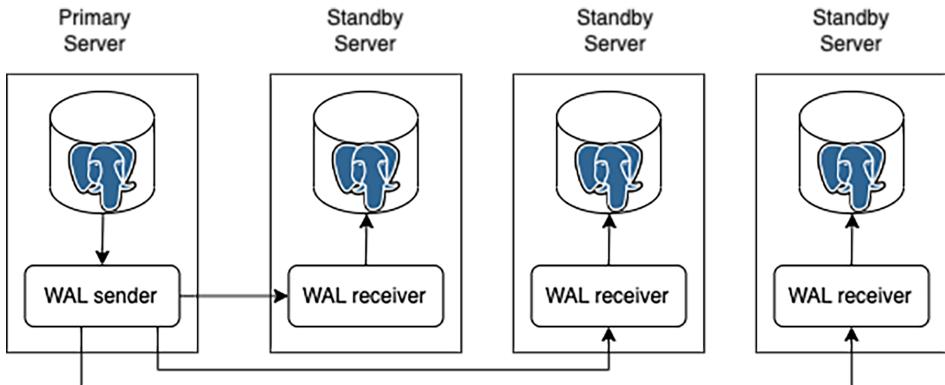


Figure 8.5: Multiple standby physical replication.

To prevent overwhelming the primary with the load from many standby nodes, we can use another PostgreSQL feature for the physical replication: cascading replication. When using the cascading replication setup, all the previous concepts remain valid; the difference is one or more standby nodes can serve as the upstream for another standby node, preventing the primary from getting this extra load. *Figure 8.6* shows how this appears:

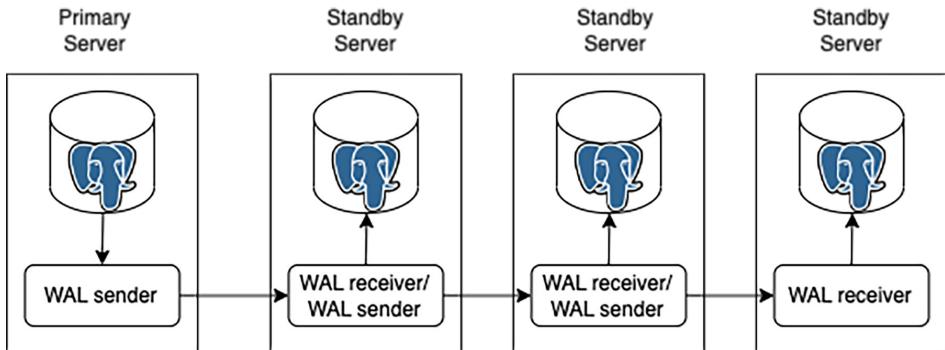


Figure 8.6: Cascading physical replication.

Configuring the cascading replication requires setting the `primary_conninfo` in the downstream server, pointing to another standby server that will serve as the upstream.

The following lines show an example of the configuration to achieve this replication type.

```
# Primary

wal_level = 'replica'
```

```
archive_mode = 'on'

archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/
server/archivedir/%f' # optional

# Standby

hot_standby = 'on'

primary_conninfo = 'host=<the primary or another standby as upstream>
port=5432 user=someuser password=passwd application_name=s1'

recovery_target_timeline = 'latest'

restore_command = 'cp /mnt/server/archivedir/%f "%p"' # optional
```

Note: For the cascading replication setup is advisable to set the parameter `recovery_target_timeline` to 'latest' in the downstream standby nodes, so if the upstream standby gets promoted (it turns into a read-write), the cascading won't get broken.

Delayed replica

Another interesting feature supported by PostgreSQL regarding physical replication is what we know as a **delayed replica**. This method relies on top regular streaming or archive recovery replication. The difference with the delayed replica is that the data changes will be *visible* after a defined time. For example, we plan to have a replication delayed by X time so that the data would be accessible in that replica as follows:

$$\text{delayed replica visibility time} = \text{data change time} + X$$

As an example, consider the next:

- We configured a replica delayed by 2 hours.
- There is a data change in the master node applied at 12:22 PM.

When will the data change be visible in the delayed replica?

$$\text{delayed replica visibility time} = 12:22 \text{ PM} + 2 \text{ hours}$$

$$\text{delayed replica visibility time} = 2:22 \text{ PM}$$

Figure 8.7 illustrates this functionality:

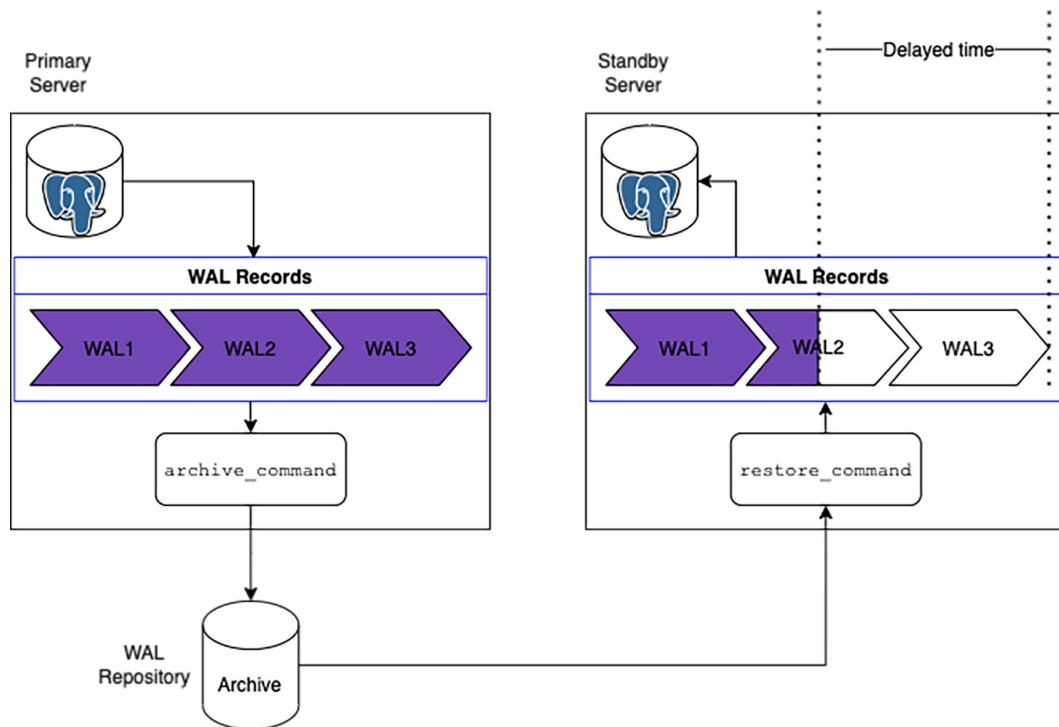


Figure 8.7: Delayed replica

We need to consider the time calculated regarding the local system time. So, for a practical configuration, we would need to ensure that both the primary and the standby servers are configured equally for their local timing.

As was stated before, the delayed replication is a variation of the *regular* streaming or archive recovery methods. In both cases, the changes are taken from the WAL and applied on the standby. But we can tell that the archive recovery is the more effortless/straightforward way to configure it. So, it will continuously retrieve the WAL from the archive and apply the changes just when the delayed time has passed.

To get this functionality working on a PostgreSQL standby instance, we need to configure the `recovery_min_apply_delay` parameter according to the time we want to keep it delayed from the master. The next is an example of the parameter configuration for the setup of the delayed replica.

```
# Primary
wal_level = 'replica'
```

```
archive_mode = 'on'

archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/
server/archivedir/%f' # optional

# Standby

hot_standby = 'on'

primary_conninfo = 'host=187.168.1.50 port=5432 user=someuser
password=passwd application_name=s1'

recovery_min_apply_delay = '2h'

restore_command = 'cp /mnt/server/archivedir/%f "%p"' # optional
```

We can use a delayed replica to provide a quick way to recover data or schema definition in the case of an erroneous deletion, table truncation, drop, or failed application deployment, which has changed a large piece of data making the undo complicated.

Configuration

We have reviewed the different features and topologies we can get for the physical replication in PostgreSQL. The next is a summary of the configuration parameters example we need to keep in mind to build the same.

When working with any physical replication setup, we need to “tell” the standby nodes they are supposed to be a replica, starting with PostgreSQL 12, and onwards this is done by adding an empty file in the **data_directory** (PGDATA) called **standby.signal**. With this file in place, once the postgres server is started, it will enter the replica mode, depending on the parameters we have configured.

Logical replication

Another type of replication is logical replication, where data is replicated logically instead of physically replicating block by block at the file system level. It is the same as taking logical backup, which we have seen in detail in *Chapter 7, Backup & Restore in PostgreSQL*.

Architecture

Logical replication uses a publisher and subscription model where the publisher pushes the data to one or more subscribers. The subscriber pulls the data from the publishers to which it is subscribed. One can configure as complex an architecture as possible using publisher and subscriber models, like one publisher and one subscriber, one publisher and multiple subscribers, cascading replication, and many more, as shown in the previous sections of this chapter.

Tables whose data needs to be replicated must have a replica identity, usually the Primary/Unique Key. It starts by initially taking a snapshot of the tables in the publisher node. The subscriber will initially pull those changes from the publisher, which takes time, depending on the table size.

Any transactional changes are replicated as and when the transaction occurs. The subscriber applies the changes in the same order in which they happen on the publisher node. The main thing to remember while setting up the logical replication is that it can only replicate **Data Manipulation Language (DML)** changes such as **INSERT**, **DELETE**, or **UPDATE**, not **Data Definition Language (DDL)** changes like **CREATE**, **ALTER**, or **DROP**.

The replication will break in case any changes to table structure in the form of altering the table are applied to the tables on which replication has been set. Also, the subscriber node may or may not be the read-only node. The only thing to be kept in mind is to avoid conflicts and refrain from performing write operations on the same set of tables on which logical replication has been configured.

Publication

The publication can be described same as the master node in the physical replication. The node on which the publication is configured is called the publisher node, which is responsible for sending data to the subscription node. The process which is used to send the data is WAL SENDER.

The publisher can be created using the **CREATE PUBLICATION** command and modified or dropped using **ALTER PUBLICATION** Command. The tables used for logical replication must have replication identity in the form of UNIQUE/PRIMARY KEY. Data is replicated in real-time once the initial snapshot of the tables is created. Publisher can choose to replicate the changes for either **INSERT**, **UPDATE**, **DELETE** or **TRUNCATE**, or a combination of any of them.

Syntax:

```
CREATE PUBLICATION name  
[ FOR TABLE [ ONLY ] table_name [ * ] [, ...]  
| FOR ALL TABLES ]  
[ WITH ( publication_parameter [= value] [, ... ] ) ]
```

Subscription

Subscription is like a standby node in the physical replication. The node which pulls data from the publisher is called the subscriber node. Subscription can be added on the child node using the **CREATE SUBSCRIPTION** command and modified or deleted using **ALTER SUBSCRIPTION** command.

Syntax:

```
CREATE SUBSCRIPTION subscription_name  
CONNECTION 'conninfo'  
PUBLICATION publication_name [, ...]  
[ WITH ( subscription_parameter [= value] [, ... ] ) ]
```

Configuring the Logical Replication:

Once tables have been identified to be set for logical replication, make the configuration changes as following:

Changes in **postgresql.conf** file:

Publisher Node:

wal_level = Logical

Publisher node as well as subscription node

Apart from this, in case replication slots have been used, then set **max_replication_slots** same as the number of subscriptions which will be connecting. Also, add some more as a buffer which will be used for table synchronization. **Max_wal_senders** should be set as **max_replication_slots**. Additionally **max_worker_processes** can be set as **(max_replication_slots + 1)**. Except **wal_level**, all other parameters will be same at the publisher and subscriber nodes.

- Changes in **pg_hba.conf** file:

Allow replication user in **pg_hba.conf** file for both the nodes:

```
host      all      replication_user      192.168.80.2/32      md5
```

- Create Publication on the Publisher node as per the following example:

```
CREATE PUBLICATION pub_test FOR TABLE emp, dept;
```

- Create Subscription on the Subscriber node as per the following example:

```
CREATE SUBSCRIPTION sub_test CONNECTION 'dbname=db1
host=192.168.80.1/32 user=replication_user' PUBLICATION pub_test;
```

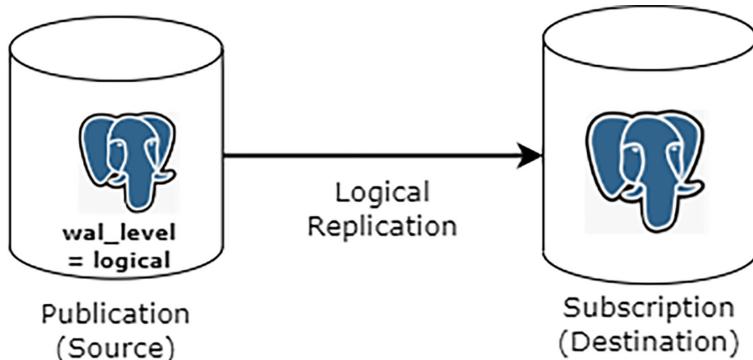


Figure 8.8: PostgreSQL Logical Replication

Please find the other use cases of logical replication as mentioned following:

- Logical replication can be used to replicate data between different versions of PostgreSQL. It is majorly used for performing data migration while upgrading the database.
- Since it is not managed at the OS or file system level, it works very nicely for performing replication with different platforms. For example, replicating data from Windows to Linux or replicating data amongst different flavors of Linux/Unix, and so on.
- Unlike physical replication, where one cannot replicate part of the PostgreSQL database cluster, logical replication can send incremental data changes to a single database or subset of the database. For example, BI & reporting team needs access to specific tables only and wants to perform some operations on its own schema tables; one can setup logical replication for a specific table from production, which will push the incremental changes in the same database where reporting database is hosted. BI Team does not need to have access to the entire production database just to access a few tables.

- Sharing incremental changes of a specific set of tables to multiple databases.

Amalgamate numerous databases into one database.

Please find the limitations of Logical Replication as following:

- Tables used in logical replication must have a Primary Key/Unique Key.
- Only DML can be replicated, and DDL cannot be replicated.
- TRUNCATE is also not replicated.
- SEQUENCES and views cannot be replicated.

Conclusion

In this chapter, we learned the different topologies we can build with physical replication and logical replication, along with different configuration setups. All these designs option bring flexibility to cover many different solutions. Also, we can take advantage of them to get systems that can stand in front of unexpected failures and bring service protection.

The next chapter will dive deep into Security and Access Control.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 9

Security and Access Control

Introduction

Security is one of the most important topics in any technological solution, and it gets more relevant if we are talking about data. So, when working with database systems is vital to know what are the strength and weaknesses in the security of our implementations.

Another essential concept closely tied to security is access control. Defining who can access the systems is crucial. PostgreSQL provides enough resources and interfaces to manage these elements efficiently and make our systems secure.

Structure

In this chapter, we will study the two main concepts involved in security and access control from the PostgreSQL point of view.

- Authentication
- Authorization

Objectives

During this chapter's study, you will learn about security and access control concepts and how they apply to PostgreSQL environments. We will review definitions and examples and walk through a few use cases. Once completed, you will be familiar with the concepts and able to relate the risks of lousy security designs and how to overcome them.

Authentication

When speaking about security in database systems, we can start looking at the components between the database and the access points, such as the Operating System itself, the Network configuration, applications, their users, and so on. However, some of these are out of the scope of this book since we are focusing exclusively on PostgreSQL.

Then the first barrier we can fortify to secure our database is authentication. We can define authentication as the mechanism to verify a user's identity trying to connect to the PostgreSQL system. Postgres offers different authentication methods, all controlled from a couple of configuration files called **postgresql.conf**, **pg_hba.conf** and **pg_ident.conf**. The following paragraphs will examine the files and the different authentication methods.

The pg_hba.conf

This file is generally the main configuration file for authentication. As we briefly saw in **Chapter 5, Architecture of PostgreSQL**. By default, this file is located in the PGDATA directory, the main path for the PostgreSQL instance; however, it can be placed elsewhere.

The HBA in the name of this configuration file stands for Host-Based Authentication. The authentication rules contain one per line (or row), each divided into columns:

- The type of connection.
- The name of the target database.
- The database user name.
- The source IP address or the network range, if applicable.
- The authentication method.

Commentary lines (starting with the # sign) are ignored, the same as the empty lines. Some columns accept wild cards, so writing a rule applicable to different criteria is possible. We need to consider the rules in the file are processed from top to bottom, and the first applicable rule is taken; if the authentication fails with the rule, no other one is considered.

Considering the PostgreSQL version 14, the following is an extract of the `pg_hba.conf` file you will get during the default installation:

```
# This file controls: which hosts are allowed to connect, how clients
# are authenticated, which PostgreSQL user names they can use, which
# databases they can access. Records take one of these forms:
#
# Local      DATABASE  USER  METHOD  [OPTIONS]
# host       DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostssl    DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostnossal DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostgssenc  DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostnogssenc DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
#
#
```

Let us take a look at what the meaning of these values is.

local

This type of connection means the client is connecting using the Unix-domain socket. No **Transmission Control Protocol/Internet Protocol (TCP/IP)** communication is involved.

host

This type of connection means the client connects through TCP/IP protocol; this rule would match independently if the communication uses or not the Secure Sockets Layer (SSL) for encryption.

hostssl

This rule will match any connection through TCP/IP, and SSL is enabled for encryption. The SSL details are out of the scope of this book, but you can look further for details at <https://www.postgresql.org/docs/14/ssl-tcp.html>.

hostnossal

As you can imagine, this rule matches any connection attempt made through TCP/IP that does not use SSL. This is the opposite of the previous one.

hostgssenc

Similar to the SSL connection rules, this one matches those connection attempts with TCP/IP that use **Generic Security Service Application Program Interface (GSSAPI)** for the encryption.

hostnogssenc

The opposite of the previous, it only matches when the connection is made through TCP/IP and does not use GSSAPI for encryption.

Database

The value used in this column will match the requested database in the connection so that it can contain the database name or any of the next values: **all**, **sameuser**, **samerole**, or **replication**. *Table 9.1* shows the meaning of these.

Value	Means
all	The rule will match all the databases.
sameuser	The rule will match if the database is named after the user name used in the connection.
samerole	The rule matches if the database is named after a role the requested user is part of.
replication	The rule matches if the requested connection is a physical replication connection.

Table 9.1: Special values for the DATABASE column in the pg_hba.conf file.

Also, it is possible to specify multiple databases in the same column for a single rule. In such cases, each is separated by a comma (,). As an extension of this last option, you can add the list of databases in a separate file and specify it by preceding the file name with the @ sign.

User

The value for this column will match the requested user for the connection. Like the previous column, you can use a few extra options. *Table 9.2* describe them.

Value	Means
all	The rule will match all the user names.
+<role>	The rule will match all the users* who are directly or indirectly members of the specified role.

Table 9.2: Special values for the USER column in the pg_hba.conf file.

*As we saw in *Chapter 4, Global Objects in PostgreSQL*, the difference between user and role is that the user is a role with the LOGIN privilege so that it can connect to the database. The role is used to logically group users and handle their privileges more efficiently.

Address

The value set for this column matches the rule for the client system IP address. You can specify an IPv4 or IPv6 address and include the corresponding **Classless Inter-Domain Routing (CIDR)** mask length after the slash sign (/). This way, you can specify a specific host IP, a small network range, or a larger network range. *Table 9.3* shows a few examples:

Range	IPv4	IPv6
Single host	172.58.10.2/32	fe80::7a31:c1ff:0370:7334/128
Small network	172.58.10.0/24	fe80::7a31:c1ff:0000:0000/96
Larger network	172.58.0.0/16	fe80::0000:0000:0000:0000/64
All the network	0.0.0.0/0	::/0

Table 9.3: Examples of IP addresses for the ADDRESS column in the pg_hba.conf file.

Like the previous columns, the **ADDRESS** column also admits some particular values. *Table 9.4* lists them:

Value	Means
all	The rule matches any IP address.
samehost	The rule matches any IP address from the database server.
samenet	The rule matches any IP address from the network to which the database server is connected.

Table 9.4: Special values for the ADDRESS column in the pg_hba.conf file.

In addition to specifying the IP address or any special words, this column can also admit network host names. Any other string than an IP notation or the special keywords is treated as a host name. A very good **Domain Name System (DNS)** is essential to use host names to execute the name resolution quickly.

We might say using host names is the most “complicated” method because the name verification happens twice, with one reverse name resolution and one forward name resolution. If during the process, an error occurs or the returned IP/name pair doesn’t match, then the rule won’t match either.

Any entry starting with a dot (.) will be considered a domain and will match the suffix of the host name. So, for example, the value `.example.com` will match `test.example.com` but not just `example.com`.

Method

This column defines the authentication method when all the other columns match a connection request. PostgreSQL supports a variety of authentication methods. We will see the available methods in the **Authentication methods** subsection.

[Options]

The last column is optional. Some authentication methods accept these extra options. The values for this column can be a list of parameters in the form `name=value`. The following authentication methods can accept the extra options: **ldap**, **pam**, **radius**, **sspi**, **ident**, **cert**, and **peer**.

Finally, any change made to the `pg_hba.conf` file will not become active immediately. This file is read during the PostgreSQL server startup or after the SIGHUP signal.

This can be done with the **pg_ctl reload** command or by using the database function **pg_reload_conf()**. Figure 9.1 illustrates both methods.

```
postgres@ubuntu-focal:~$ /usr/lib/postgresql/14/bin/pg_ctl reload \
> -D /var/lib/postgresql/14/main
server signaled
postgres@ubuntu-focal:~$ _

postgres=# SELECT pg_reload_conf();
 pg_reload_conf
-----
 t
(1 row)
```

Figure 9.1: SIGHUP signal methods to reload pg_hba.conf file changes

Authentication methods

As we saw before, PostgreSQL supports different authentication methods, and *Table 9.5* describe them:

Auth method	Description
trust	Allow the connection unrestrictedly. Using this, there is no need for a password or other authentication.
reject	Reject the connection unconditionally. Using it, there won't be any authentication requirement, and the connection request is immediately rejected. It is helpful to filter out a host or an IP address range.
scram-sha-256	This performs a SCRAM-SHA-256 challenge-response authentication that prevents password sniffing and supports storing the password on the server in a cryptographically hashed form that is considered very secure. Currently, it is the most secure method, but some old client libraries don't support it.
md5	It uses a challenge-response mechanism that prevents password sniffing and doesn't store the passwords on the server in plain text. However, it doesn't protect if an attacker steals the password hash. If md5 is set, but the password was encrypted for SCRAM, then SCRAM-based authentication is used.
password	This method sends the password in clear text, so it is vulnerable to sniffing attacks. It should be avoided unless the connection is protected using SSL.

Auth method	Description
gss	Uses GSSAPI to authenticate the user; it is only available when using TCP/IP connections.
sspi	It uses SSPI, the full form is Security Support Provider Interface, to perform the user authentication. This option only applies to systems running Windows.
ident	Operates by obtaining the operating system user name for the client from an external ident server and using it as the database user. A user name mapping (see next section <code>pg_ident.conf</code>) can be used in conjunction optionally. This method only is available in TCP/IP connections.
peer	This one gets the operating system user name for the client from the kernel and verifies if matches with the database user. This is only available for local connections.
pam	This method performs the user authentication via the Pluggable Authentication Modules (PAM) service from the operating system.
ldap	Authenticates by using a Lightweight Directory Access Protocol or LDAP external server.
radius	Authenticates by using a Remote Authentication Dial-In User Service or RADIUS external server.
cert	Authenticates by using SSL client certificates, the server and the client have to interchange valid certificates. Only available when SSL connections type is used.
bsd	This method uses the BSD service from the operating system to perform the authentication.

Table 9.5: Authentication methods supported by PostgreSQL

The `pg_ident.conf`

The second file that has a role for some authentication methods is the **`pg_ident.conf`**. The same as the **`pg_hba.conf`**, this file is in the **PGDATA** directory by default but can be placed elsewhere.

This file is used with some authentication methods that rely on the operating system user names, such as **gss** or **ident**; it is helpful in the situation the client operating system user name is different from the database user. A user name map is required to map the operating system user with the database user.

When defining the user name mapping is necessary to specify **map=<name>** in the **OPTIONS** column of the **`pg_hba.conf`** file for the given rule. Then the user map,

identified by the `<name>` value, is defined in the `pg_ident.conf` file. A mapping definition looks like the following:

```
name_of_map operating-system_user database_user
```

Commentaries, blank lines, and whitespaces are processed the same as the `pg_hba.conf` file. The `map-name` can be an arbitrary value used to identify it in the `pg_hba.conf` file, the other two values are the operating system user and a matching database user. It is possible to set the same `<name>` multiple times to define multiple user mappings with a single map name.

When configuring the user mappings it is advisable to consider the operating system user and the database user are not equivalent, but the operating system user is permitted to connect as the database user. A connection will be successful if a user mapping matches the user name obtained from the external authentication system and the requested database user.

It is possible to save some lines when configuring the same map for multiple operating system users using regular expressions (*Reference: The PostgreSQL Global Development Group 2022*). In this case, the `system-username` value should start with a slash (/) and then the expression. For example, the next line will allow any system user name that ends with `@somedomain.com` to login into the database as `guest`.

```
mymap /^.*@somedomain\.com$ guest
```

It is possible to capture (parenthesized subexpression) a portion of the expression and then reference it in the `database-username` with the `\1` expression. See the following example; here, the domain part for any system user name ending with `@mydomain.com` will be removed and allow login into the database as the same user name.

```
mymap /^(.*)@mydomain\.com$ \1
```

PostgreSQL reads the `pg_ident.conf` file during the startup or when the `SIGHUP` signal is sent to the server. So, to make any change active, you can do the same as with the `pg_hba.conf` file, see *Figure 9.1*.

Examples

Now we have studied the two main files for the authentication process and the available authentication methods, we can examine a couple of examples to make this clearer. Consider the following `pg_hba.conf` and `pg_ident.conf` files.

`pg_hba.conf`

```
# TYPE  DATABASE      USER          ADDRESS           METHOD  
  
# 1.  
host    all           all           172.58.0.0/16     ident map=sales  
  
# 2.  
host    pgbench       all           192.168.12.10/32   scram-sha-256  
  
# 3.  
host    all           fred          .testdom.org      md5  
host    all           all            .testdom.org      scram-sha-256  
  
# 4.  
local  sameuser      all           md5  
local  all           @managers     md5  
local  all           +dba          md5  
  
pg_ident.conf  
# NAME      OS  USER      PG  USER  
  
# 5.  
sales    fred         fred  
sales    karen        karen  
sales    robert       bob  
sales    fred         tester
```

The following are the explanation of the lines from the previous example files.

1. It allows connections from **172.58.0.0** hosts to any database if the ident is passed. If ident says the user is “**fred**” and the PostgreSQL connection was requested as user “**tester**,” it will succeed since there is an entry in **pg_ident.conf** for map “**sales**” that allows “**fred**” to connect as “**tester**.”

2. It lets users from host **172.58.12.10** connect to the database “**pgbench**” if the password is correct.
3. Any user from the **testdom.org** domain can connect to any database if the password is supplied. It requires **SCRAM** authentication except for user “**fred**” since it uses an old application that doesn’t support **SCRAM**.
4. These lines let local users connect only to databases with the same name as their database user name, except for the users listed in the **\$PGDATA/managers** file and the members of role “**dba**,” who can connect to all databases. The password is asked in all cases.
5. From the “**sales**” user mapping definition, we can see the operating system “**robert**” can connect to the database as “**bob**” no “**robert**,” “**karen**” as the same name, and “**fred**” can connect as “**fred**” or “**tester**” user.

Authorization

The authorization is the next step in securing our database. We can say this is the process of defining *who* can access *what*. So, once a user gets connected to the database after the authentication, the authorization definition will handle what he/she can “see.”

If you remember from *Chapter 4, Global Objects in PostgreSQL*, we studied users/roles and how the privileges can be granted to them to enable access. This section will review some more advanced features and how we can use them to define a robust and secure authorization.

Speaking in PostgreSQL argot, the access or capabilities a role has, are defined in three different ways:

- Role attributes
- Object ownership
- Object privilege

Role attributes

As we saw in previous chapters, there is a special attribute the initial role (**postgres**) created just after the PostgreSQL installation gets complete, called **superuser**. This attribute contains all the possible permissions in the database cluster, so it is used to initialize the database model, create new roles and users, grant them attributes or privileges, and so on.

Alongside the **superuser** attribute exists a list of attributes with a wide cluster effect, which means their effect applies to the whole cluster and not just to specific databases. Table 9.6 describe all the existing roles up to PostgreSQL version 14.

Role attribute	Description
SUPERUSER NOSUPERUSER	As described before, this attribute bypasses any check except the login and has full capabilities in the database cluster.
LOGIN NOLOGIN	This attribute enables a role to connect to a database cluster. We can say the roles with this attribute are database users.
CREATEDB NOCREATEDB	This attribute gives a role the capability to create and drop databases if the same is the database owner.
CREATEROLE NOCREATEROLE	This attribute lets a role to be able to create other roles. Also, a role with this can alter, drop or grant membership to others.
INHERIT NOINHERIT	The INHERIT attribute is granted by default to all the roles, meaning the role can inherit the privileges from any other role granted. NOINHERIT should be used explicitly to prevent this.
REPLICATION NOREPLICATION	This attribute enables a role to start a streaming replication session. The role should have the LOGIN attribute as well.
BYPASSRLS NOBYPASSRLS	A role with this attribute gets the ability to bypass every row-level security (RLS) policy. We will study these rules in a future chapter.
CONNECTION LIMIT	This attribute can limit the number of simultaneous connections a role can have.
PASSWORD	This attribute enables a role to have a password and only has meaning for those authentication methods which require a password.

Table 9.6: Role attributes.

As you can see, all the attributes, except for **PASSWORD** and **CONNECTION LIMIT**, have a *negative* counterpart, so we can explicitly set the attribute or deny it. The attributes can be established during role creation or after with the **ALTER ROLE** command. Figure 9.2 illustrates this:

```
postgres=# CREATE ROLE sample SUPERUSER;
CREATE ROLE
postgres=# ALTER ROLE sample NOSUPERUSER;
ALTER ROLE
postgres=#

```

Figure 9.2: Setting role attributes

Object ownership

This is another aspect of the roles to take into account. The roles own any object created by themselves and, being the owner, grant full access to these objects. So modifying them, deleting them, or granting specific access, such as **SELECT**, to other roles is possible.

Being the owner of a table or other objects doesn't grant the capability to edit objects owned by a different role. Only the superuser roles can modify or delete objects owned by a different role.

Remember that any database object can only have one owner, so if multiple users are required to access the object with the owner privileges level, the owner should be a role and then grant this role to the users. *Figure 9.3* illustrates this.

```

postgres=# CREATE TABLE t1 (
  id int,
  name varchar(10)
);
CREATE TABLE
postgres=# CREATE USER fred;
CREATE ROLE
postgres=# CREATE USER john;
CREATE ROLE
postgres=# CREATE ROLE owners;
CREATE ROLE
postgres=# ALTER TABLE t1 OWNER TO owners ;
ALTER TABLE
postgres=# GRANT owners TO john, fred ;
GRANT ROLE
postgres=# \dg fred|john|owners
      List of roles
   Role name | Attributes | Member of
   -----+-----+-----+
     fred    |           | {owners}
     john    |           | {owners}
   owners   | Cannot login | {}

postgres=# \dt t1
      List of relations
 Schema | Name | Type | Owner
 -----+-----+-----+
  public | t1   | table | owners
 (1 row)

postgres=#

```

Figure 9.3: Roles object ownership

Objects privileges

Finally, the last way a role can get authorization to operate with the database objects are the granted object privileges. This would apply to all those roles that are not owners of the objects. Usually, these users/roles are used by the applications or clients that connect to the database, so they can hold only the minimum required access privileges.

The object privileges are controlled with the commands **GRANT** and **REVOKE**. In context object privileges, the **GRANT** command is used to add privileges on specific database objects, and the opposite **REVOKE** is used to remove them. *Figure 9.4* shows an example.

```
postgres=# CREATE USER ann;
CREATE ROLE
postgres=# GRANT SELECT ON t1 TO ann;
GRANT
postgres=# REVOKE SELECT ON t1 FROM ann;
REVOKE
postgres=#
```

Figure 9.4: GRANT and REVOKE commands.

There are different object privileges available. *Table 9.7* describe them:

Privileges	Description
SELECT	Allows reading from any column or set of columns of a table, view, materialized views, or any other table-like object.
INSERT	Allows writing new rows into a table, view, etc. It can be granted on specific columns. In such cases, only these columns can be assigned in the INSERT command.
UPDATE	Allows modifying existing rows in a table, view, etc. Consider the UPDATE command also requires SELECT privilege since, in practice, it would reference a table column to filter which rows to affect.
DELETE	Allows removing existing rows from a table, view, etc. Like UPDATE , any DELETE would require the SELECT privilege due to the same reason, it should reference table columns to define the rows to be deleted.
TRUNCATE	Allows removing ALL the rows on a table.
REFERENCES	Allows creation of a foreign key constraint referencing a table, or specific column(s).
TRIGGER	Allows the creation of a trigger on a table, view, and the like.

Privileges	Description
EXECUTE	Allows invoking a function or procedure. Only applicable to these object types.
USAGE	This privilege can apply to different object types. For procedural languages, it allows the use of language to create functions or procedures. For schemas, it allows access to the objects within the schema. Assuming other specific privileges on the objects were granted. For sequences, it allows calling the <code>currval</code> and <code>nextval</code> functions to see the current sequence value or ask for the next one.

Table 9.7: PostgreSQL object privileges

Conclusion

Making our database system secure is crucial to ensure data protection and prevent leaks. There are multiple options to enforce security around PostgreSQL, like operating systems or networking hardening. However, PostgreSQL by itself supports different authentication methods so the identity of the users can be verified when trying to connect. Also, you can design the access level a given user can have with the existing tool, such as the role's attributes, object ownership, and object privileges.

In the next chapter, we will learn about some of the most used tools and extensions the open-source world has brought into PostgreSQL.

Bibliography

- The PostgreSQL Global Development Group: <https://www.postgresql.org/docs/14/functions-matching.html#POSIX-SYNTAX-DETAILS>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 10

Most used Extensions/Tools

Introduction

Being an open-source project, PostgreSQL has caught the attention of a large community. The same has been involved in fixes and new features added to the core code. However, some other open-source projects are built around postgres to extend its capabilities or make some tasks easier.

This chapter will review some of the extensions and extra tools available for PostgreSQL and what we can achieve with them. We will learn how to get and integrate them with postgres and how to configure them for work.

Structure

We will study this chapter in two sections. Each one will explain a set of extensions and tools we consider the most helpful and widely used. We will not review backup tools since we already saw some in *Chapter 7, Backup and Restore in PostgreSQL*.

- Extensions
- Tools

Objectives

You will learn how to integrate an extension to PostgreSQL and how to use it. Also, you will become familiar with the most popular external tools and what we can do with them. Once complete this chapter, you will have a good knowledge of how to configure and use them.

Extensions

As we have seen in the past chapters, PostgreSQL is a very robust database software. The base distribution from the community packages already contains unique functionalities that cover many operations out of the box. However, the design of PostgreSQL makes it able to extend its capabilities by integrating what we know as **extensions**.

An extension is a set of the control file, SQL files, and loadable libraries. The libraries are usually written in C language; once they are loaded, PostgreSQL can use the functionalities like any other core function.

Adding an extension to PostgreSQL generally requires two or three steps, depending on whether the extension routines need to perform operations at the server start.

- Install or get the extension libraries at the operating system level. Usually, installing packages accordingly to the Operating System distribution we are using.
- Some extensions require to be added to the **shared_preloaded_libraries** parameter in the **postgresql.conf** file. Any change to this parameter requires a PostgreSQL restart to become active. During the startup routine, the added libraries will also be loaded and perform their tasks, such as allocating memory or starting background processes.
- Create the extension within the PostgreSQL database where we will use it. Once the libraries are placed at the operating system level, we can execute the **CREATE EXTENSION** command to load the library and create all the required database objects.

As we saw in *Chapter 2, Getting PostgreSQL to work*, *Figure 2.3*, when installing PostgreSQL from the source code, we get a folder called **contrib/**, which is shipped with the source and contains several extensions. The PostgreSQL core community maintains all these extensions; however, there are others developed externally.

In the following subsections, we will review some extensions widely used and helpful for various projects. We will study the steps to add them to PostgreSQL version 14 running on Ubuntu and the basics for their usage.

pg_cron

This extension lets to schedule database jobs at specific times; it uses the same time and job conventions as the standard **cron** utility for Linux operating systems. So you can easily schedule recurrent database tasks or maintenance events. The following are some examples.

```
-- Execute cleaning data custom function on Sunday at 3:00 AM GMT
SELECT cron.schedule('0 3 * * 7', $$SELECT f_clean_data()$$);
schedule
-----
8

-- Runs vacuum daily at 5:30 AM GMT
SELECT cron.schedule('daily-vacuum', '30 5 * * *', 'VACUUM');
schedule
-----
9

-- Stop scheduling daily-vacuum job
SELECT cron.unschedule('daily-vacuum' );
unschedule
-----
t
```

The Citus Data company currently supports this extension as an open-source project (Reference: *pg_cron*).

Consider running a PostgreSQL version 14 on Ubuntu operating system. The following are the steps to integrate and use this extension.

1. Install the required package for the PostgreSQL version on Ubuntu.
`sudo apt-get -y install postgresql-14-cron`
2. The **pg_cron** extension requires to be pre-loaded at server startup so that it can allocate background workers. So, we need to add the next to the **postgresql.conf** file and restart postgres.

```
# edit postgresql.conf
shared_preload_libraries = 'pg_cron'
```

3. By default, the extension expects to be added to the **postgres** database and to schedule the jobs based on the GMT timezone. You can modify these defaults to use a different database and another timezone, for example, database *prod1* and the timezone CST.

```
# edit postgresql.conf
cron.database_name = 'prod1'
cron.timezone = 'CST'
```

4. Finally, once you have added the previous configuration and restarted PostgreSQL, you need to create the extension in the desired database. The following should be executed as a superuser.

```
CREATE EXTENSION pg_cron;
```

Figure 10.1 illustrates the creation of the extension within the **postgres** database and shows the created schema called **cron** and its tables.

```
postgres=# CREATE EXTENSION pg_cron;
CREATE EXTENSION
postgres=# \dx
      List of installed extensions
   Name   | Version | Schema | Description
-----+-----+-----+
pg_cron | 1.4-1  | public | Job scheduler for PostgreSQL
plpgsql | 1.0    | pg_catalog | PL/pgSQL procedural language
(2 rows)

postgres=# \dn
      List of schemas
   Name   | Owner
-----+-----
cron   | postgres
public | postgres
(2 rows)

postgres=# \dt cron./*
      List of relations
 Schema |        Name        | Type | Owner
-----+-----+-----+
cron   | job             | table | postgres
cron   | job_run_details | table | postgres
(2 rows)

postgres=#
```

Figure 10.1: pg_cron extension

pg_stat_statements

This is a really interesting and valuable extension. Once loaded and added to a database, this extension will collect statistics about the executed queries, such as the query text, the query identifier, the user and database names, the number of times the query executes, the total time spent in the planning phase and the time spent in the execution phase, the number of rows retrieved, and much more.

All this information comes in handy when tracking performance issues or verifying what queries are causing the most load in the system so you can plan improvements. You can access the collected data through a database view named the same as the extension: **pg_stat_statements**. The following are some examples:

```
-- Find the query with the highest total execution time
SELECT max(total_exec_time) AS max_exec_time, query
FROM pg_stat_statements
GROUP BY query
ORDER BY max_exec_time DESC LIMIT 1 ;
max_exec_time | query
-----+-----
216302.15497300148 | UPDATE pgbench_branches SET bbalance = bbalance
+ $1 WHERE bid = $2
(1 row)

-- Find the query executed the most times
SELECT max(calls) AS max_calls, query
FROM pg_stat_statements
GROUP BY query
ORDER BY max_calls DESC LIMIT 1 ;
max_calls | query
-----+-----

```

```
72981 | UPDATE pgbench_accounts SET abalance = abalance + $1 WHERE
      aid = $2

(1 row)

-- Find the query which retrieved the most rows

SELECT max(rows) AS max_rows, query
FROM pg_stat_statements
GROUP BY query
ORDER BY max_rows DESC LIMIT 1 ;

max_rows |           query
-----+-----
100000 | copy pgbench_accounts from stdin

(1 row)

-- Find the query with the worst cache hit ratio (most reads from disk)

SELECT min(100.0 * shared_blk_hit/nullif(shared_blk_hit + shared_
blk_read, 0)) hitratio,
       substr(query,0,50) query
FROM pg_stat_statements
GROUP BY query
ORDER BY hitratio ASC NULLS LAST LIMIT 1 ;

hitratio |           query
-----+-----
84.6153846153846154 | SELECT e.extname AS "Name", e.extversion AS
                      "Vers

(1 row)
```

This extension is currently delivered as part of the **contrib/ package** from the base community distribution (*Reference: pg_stat_statements*).

The following are the steps to add the extension to PostgreSQL.

1. Considering this example is running on Ubuntu, when installing the postgresql server package, the **postgresql-contrib** package also gets installed. This last one contains multiple extensions, as we saw above. So the extension libraries are already present.
2. Due to the fact this extension requires allocating some extra shared memory at the postgres startup, you need to add it to the **shared_preloaded_libraries** parameter.

```
# edit postgresql.conf
shared_preload_libraries = 'pg_stat_statements'
```

3. The extension supports a few configuration parameters, so optionally, you can adjust them in the **postgresql.conf** file. For example, you can change the number of queries to track from the default 5000 to 10000. Check the bibliography entry cited above for extra details about the configuration parameters.

```
# edit postgresql.conf
pg_stat_statements.max = 10000
```

4. After adding the configuration and restarting postgres, you can create the extension in the database.

```
CREATE EXTENSION pg_stat_statements;
```

In *Figure 10.2*, you can see the result of creating the extension; it will add new views:

```
postgres=# CREATE EXTENSION pg_stat_statements;
CREATE EXTENSION
postgres=# \dx
              List of installed extensions
   Name    | Version | Schema | Description
-----+-----+-----+
pg_stat_statements | 1.9    | public  | track planning and execution statistics of all SQL statements executed
plpgsql          | 1.0    | pg_catalog | PL/pgSQL procedural language
(2 rows)

postgres=# \dv pg_stat_statements*
              List of relations
 Schema |      Name      | Type | Owner
-----+-----+-----+
public | pg_stat_statements | view | postgres
public | pg_stat_statements_info | view | postgres
(2 rows)
```

Figure 10.2: pg_stat_statements extension

pg_repack

We will not lie if we say this is one of the most potent and helpful extensions for routine DBA tasks. As we saw in previous chapters, PostgreSQL uses **MultiVersion Concurrency Control (MVCC)** to handle multiple concurrent user sessions and maintain data integrity and consistency.

Over time, the **vacuum** removes the dead tuples generated because of the different row versions, but it does not return this space to the operating system. In certain circumstances, this *unused* space within the tables or indexes pages gets large, and even when the database might use it to accommodate new rows, sometimes it stays unused; this is what we know as bloat, negatively impacting data access performance.

The only way to eliminate bloat is to rebuild the tables or indexes. We can do that with the VACUUM FULL, but it comes with a high cost; the database object gets locked as long as the task remains. Here is where **pg_repack** comes in.

This extension and its **Command-Line Interface (CLI)** counterpart help rebuild database objects with minimal locking so you can securely use them on a running system.

pg_repack works by creating a *shadow* object and copying the data from the source. The original object remains operating with no lock, and all the rebuild happens in a different table or index. In the end, the original object is dropped, and the new one is renamed like the original. The next is an example of bloat in a given table and how it looks after the repack.

Doing a rows count first:

```
SELECT count(1) FROM pgbench_accounts;
```

```
count
-----
3000000
(1 row)
```

Calling a SQL script to check the bloated size and percentage (*Reference: Table bloat*).

```
\i bloat.sql
```

```

schemaname |      tblname      | real_size.pretty | bloat_size_byte_
pretty   | bloat_percentage

-----+-----+-----+-----+
-----+-----+-----+-----+
public    | pgbench_accounts | 384 MB           | 5592 kB          |
1.4

(1 row)

```

Deleting 1 million rows:

```

DELETE FROM pgbench_accounts WHERE aid IN (SELECT aid FROM pgbench_
accounts LIMIT 1000000) ;

DELETE 1000000

```

Refreshing table statistics:

```

ANALYZE pgbench_accounts ;

ANALYZE

```

Checking the bloat again:

```

\i bloat.sql

schemaname |      tblname      | real_size.pretty | bloat_size_byte_
pretty   | bloat_percentage

-----+-----+-----+-----+
-----+-----+-----+-----+
public    | pgbench_accounts | 384 MB           | 132 MB          |
34.4

(1 row)

```

Now we can see 132MB from the total of 384MB is bloat, representing 34.4%

From the operating system prompt, execute the pg_repack CLI for the given table:

```

postgres@ubuntu-focal:~$ pg_repack --table=pgbench_accounts postgres
INFO: repacking table "public.pgbench_accounts"
postgres@ubuntu-focal:~$ 

```

After refreshing the statistics again, we can check the bloat after the repack:

```
postgres=# \i bloat.sql
      schemaname |      tblname      | real_size.pretty | bloat_size_byte_
      pretty | bloat_percentage
-----+-----+-----+-----+
-----+-----+
      public    | pgbench_accounts | 256 MB           | 3720 kB
      1.4
(1 row)
```

The total table size is now 256MB, and all the unused space got removed. The new bloat size is 3720KB representing 1.4%.

Finally, verify the table has the expected rows after the DELETE we did before:

```
postgres=# SELECT count(1) FROM pgbench_accounts;
      count
-----
2000000
(1 row)
```

The **pg_repack** project started as a fork of another called **pg_reorg**, but it has gained its reputation. It is maintained and distributed as an open-source development (*Reference: pg_repack*).

The following are the required steps to add this extension to PostgreSQL.

1. Install the package at the operating system level. In this case, Ubuntu.

```
apt install postgresql-14-repack
```

2. Create the extension in the desired database.

```
CREATE EXTENSION pg_repack;
```

As a result of the two previous commands, you will end with the operating system CLI utility, and the PostgreSQL extension added. *Figure 10.3* illustrates the CLI utility:

```
postgres@ubuntu-focal:~$ pg_repack --version
pg_repack 1.4.8
postgres@ubuntu-focal:~$
```

Figure 10.3: pg_repack operating system CLI

Figure 10.4 shows the extension created in the database and the repack schema added; this will contain some temporary log tables used to track the repack operations.

```
postgres=# CREATE EXTENSION pg_repack;
CREATE EXTENSION
postgres=# \dn
   List of schemas
 Name | Owner
-----+
 public | postgres
 repack | postgres
(2 rows)
```

Figure 10.4: pg_repack extension

Tools

Additional to the extension projects, some other tools, and utilities do not intend to be added to the database directly, but they can help with various tasks.

We just need the right package, depending on the operating system we are working on. Some might rely on other libraries, so we need to install them beforehand, but this is specified in the tool's instructions if applicable.

In the following subsections, we will review useful tools with proven usability and results for our PostgreSQL projects.

pgbadger

Logging is an essential piece for any modern application or solution. When well-defined and configured, the log files can save the day when we need to review the activity of our service or troubleshoot an issue. PostgreSQL has multiple options to configure the logging so we can get details about the user connections and disconnections, the duration of the queries and transactions, the maintenance operations such as vacuum, and much more.

However, getting a quick insight into what has happened in the database by just *reading* the log file can be complicated. These files can be considerable, and the number of logged events huge. The **pgbadger** tool comes in handy in these situations.

pgbadger is a log parser and analyzer for PostgreSQL logs. We can get a detailed and well-built report from the database logs using **pgbadger**. It will show result tables and graphs for the most executed queries, the queries that have taken the longest to complete, queries that have waited for the most, queries that have generated the

most temporary files, the most frequent errors, and more details. *Figure 10.5* shows an example of a **pgbadger** report.

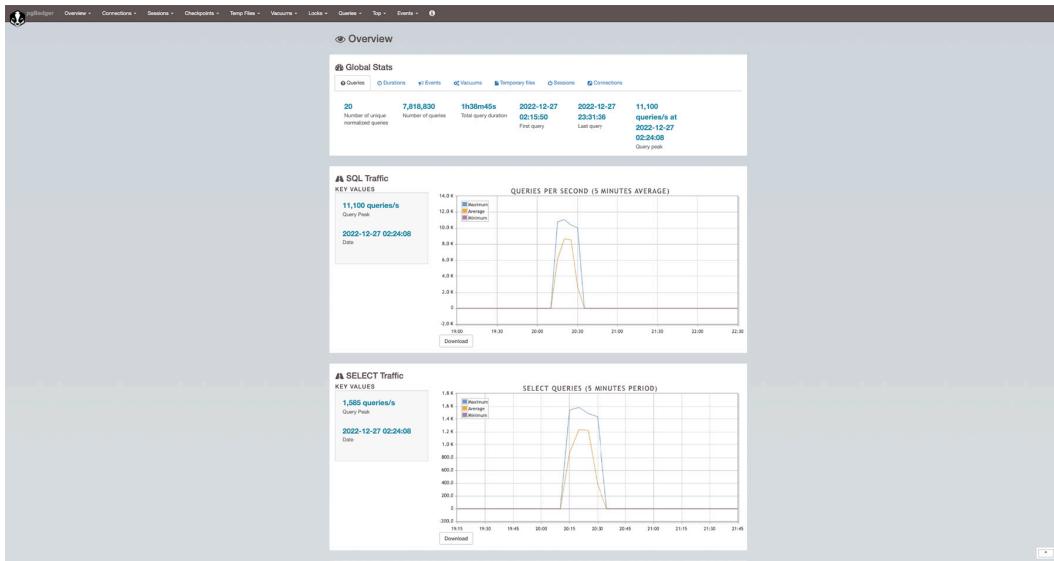


Figure 10.5: pgbadger report.

pgbadger support multiple options to configure the reports. You can source all the data from a single PostgreSQL log file or multiple; you can even execute **pgbadger** in a remote server and parse logs from another via ssh. For a more robust solution, you can configure **pgbadger** in a *central* server and source the data incrementally, say hourly, from remote database servers. *Figure 10.6* illustrates this concept.

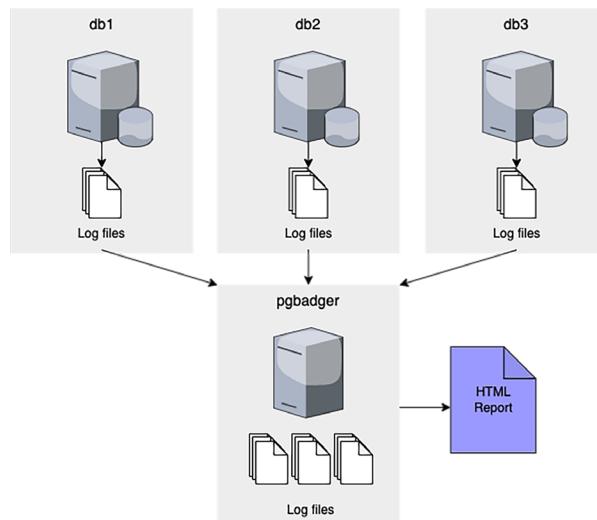


Figure 10.6: Example of central pgbadger serving multiple database servers.

The pgbadger tool is supported by Gilles Darold (*Reference: pgbadger*) and distributed under the PostgreSQL license. The following are the steps to get this tool.

1. Install the package for your operating system distro, in this example, Ubuntu.

```
apt install pgbadger
```

Once the package is installed, you can start using it. Usually, you install this in a separate server from the database to avoid consuming compute resources while the report is being created.

The available options to parse the database logs and get precisely what you need are significant. It is advisable to look at the tool's online documentation and try it by yourself.

pgbench

One fundamental concept while working as a **Database Administrator (DBA)** is benchmarking the systems. This is extensively helpful when planning a system migration, evaluating a new hardware option, or testing configuration changes, for example. With the benchmark, you can get an insight into the performance of your system so that you can compare the results and decide based on data.

In the open-source world, you might find different options to perform benchmarks of database systems. Some tools can work with various database systems, and others are oriented to a specific one. Also, the complexity of its operation might change. Hopefully, there is one that fits perfectly when working with PostgreSQL and shines because it is easy to use; this is the **pgbench** tool.

With **pgbench**, you can easily and quickly get a few predefined tables, load them with test data as big as you desire, and perform a test workload while getting statistics about the system's performance. The following is an example of a test tables initialization and benchmarking.

Execute **pgbench** to initialize the test tables:

```
postgres@ubuntu-focal:~$ pgbench -i postgres
dropping old tables...
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.09 s, remaining 0.00 s)
vacuuming...
```

```
creating primary keys...
```

```
done in 0.35 s (drop tables 0.06 s, create tables 0.02 s, client-side
generate 0.15 s, vacuum 0.07 s, primary keys 0.05 s).
```

Verify within the PostgreSQL database the tables got created using the meta-command `\dt`:

```
postgres=# \dt
              List of relations
 Schema |        Name         | Type  | Owner
-----+-----+-----+-----+
 public | pgbench_accounts | table | postgres
 public | pgbench_branches | table | postgres
 public | pgbench_history | table | postgres
 public | pgbench_tellers | table | postgres
(4 rows)
```

Execute **pgbench** against the database to test the load for ten seconds with two client sessions:

```
postgres@ubuntu-focal:~$ pgbench -c2 -T10 postgres
pgbench (14.6 (Ubuntu 14.6-1.pgdg20.04+1))
starting vacuum...end.

transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 2
number of threads: 1
duration: 10 s
number of transactions actually processed: 10706
latency average = 1.867 ms
initial connection time = 8.549 ms
```

```
tps = 1071.369350 (without initial connection time)
```

Here, the system was able to attend **1071** transactions per second (tps).

By default, **pgbench** executes a set of **INSERT**, **SELECT**, and **UPDATE** statements on the test tables, but it also supports some options to restrict the kind of operations to test, and it even is possible to execute custom scripts. You can also limit the time of the benchmark or the number of transactions to test, define the number of clients and parallel jobs, skip the vacuum operations, and so on.

pgbench is supported and maintained by the PostgreSQL community (*Reference: pgbench*) and is included in the **contrib** package from the base distribution.

To get this tool, you need to install the **postgresql-contrib** package for your operating system distribution. As we saw before, with the **pg_stat_statement** extension, this package is installed together with the PostgreSQL server package when working in Ubuntu.

pgbouncer

If you remember from previous chapters, PostgreSQL is a process-oriented software that relies on operating system processes for every executable component of its architecture. So, when you start a new PostgreSQL service, you will see many new process-ID (PIDs) running in the operating system, one for the postmaster, another for the checkpointer, another for the autovacuum, and so on. This also applies to the client or user connections; every new user session that connects to the `postgres` will create a new PID.

This design can work perfectly fine and perform as wonder. But as the number of concurrent sessions rises because of application growth, new application releases, or simply because of an increment in your user base, performance issues due to the high number of running processes in the operating system can show up.

Reducing the number of concurrent processes in the database is the best way to deal with resource saturation at the operating system level, but definitely, we will not want to limit or reduce the number of users that want to be attended by our system. So, to reduce the number of processes in the database end without sacrificing the number of clients we can serve, we can use **pgbouncer**.

The **pgbouncer** tool is a pooler application that can help to keep a large number of client requests being attended to through a reduced number of database connections. *Figure 10.7* illustrates this.

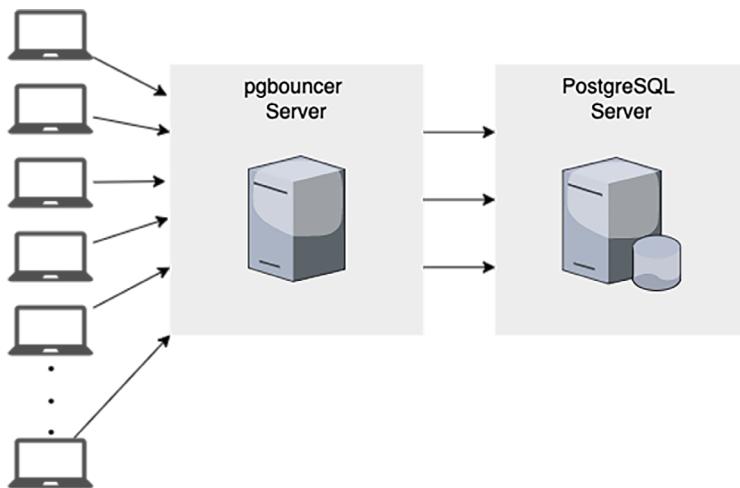


Figure 10.7: pgBouncer setup.

pgbouncer allows you to configure different pools; depending on the databases you need to connect and the users that will connect, you can define the size of the pool (backend sessions) and the number of client connections to accept. The size of the pools depends on the resource the database server has, mainly CPUs, and the number of connections you need to attend; however, the goal is to set the client connections significantly higher than the number of database connections.

Once installed, **pgbouncer** can be configured by editing the **pgbouncer.ini** file. The following is an example of a basic configuration.

```
[databases]
pg1 = host=localhost dbname=pg1 auth_user=appuser pool_size = 20

[pgbouncer]
pool_mode = session
listen_port = 6432
listen_addr = localhost
auth_type = md5
auth_file = users.txt
logfile = pgbouncer.log
pidfile = pgbouncer.pid
admin_users = adminuser
```

```
stats_users = stat_collector
max_client_conn = 200
```

In the above configuration file example, the next definitions are taking place:

- The database connection called **pg1** will create a pool of 20 backend sessions.
- The pool will attend the client sessions in **session** mode, which means a backend session from the pool will stick to a client session until it finishes; after that, the backend session can be assigned to a different client session.
- The maximum number of client connections this pgbouncer will attend is 200.

The **pgbouncer** tool is maintained by its own community (Reference: pgbouncer) and is available as an open-source project. The next are the steps to get this tool.

1. You just need to install the proper package for your operating system; as with other open-source software, you may compile it by yourself. In the case of Ubuntu, you can get it with the next.

```
apt install pgbouncer
```

Conclusion

One of the beautiful things about working with an open-source system, such as PostgreSQL, is all the options you have from the community to add new features and extend the capabilities of the base software. In this chapter, we learned about a few extensions and tools we can add to our postgres solutions.

In the next chapter, we will learn about the Basic Database Objects in PostgreSQL.

Bibliography

- pg_cron: https://github.com/citusdata/pg_cron
- pg_stat_statements: <https://www.postgresql.org/docs/14/pgstatstatements.html>
- pg_repack: [https://reorg.github.io/pg_repack/# details](https://reorg.github.io/pg_repack/#details)
- pgbadger: <https://github.com/darold/pgbadger>
- pgbench: <https://www.postgresql.org/docs/14/pgbench.html>
- pgbouncer: <https://www.pgbouncer.org/>
- Table bloat: https://github.com/ioguix/pgsql-bloat-estimation/blob/master/table/table_bloat.sql

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 11

Basic Database Objects

Introduction

In the previous chapters, we mainly learned about configurations and setting up PostgreSQL. In this chapter, we will majorly focus on basic database objects, which are essential from a PostgreSQL development perspective.

Structure

In this chapter, we will cover the following topics:

- Managing Schema
- Managing DB Objects using DDL Commands
- Enforcing Data Integrity using Constraints
- Manipulating data using DML Queries

Objectives

In this chapter, we will learn about managing schemas in PostgreSQL and how to work with **Data Definition Language (DDL)** commands like **CREATE**, **ALTER**, and the like., along with **Data Manipulation Language (DML)** commands like **SELECT**,

INSERT, **UPDATE**, and so on. We will also have some insights into the constraints in **Relational Database Management Systems (RDBMS)**.

Managing schemas

In the previous chapters, we have seen how to deal with global database objects like Users, Roles, Tablespaces, and the like:

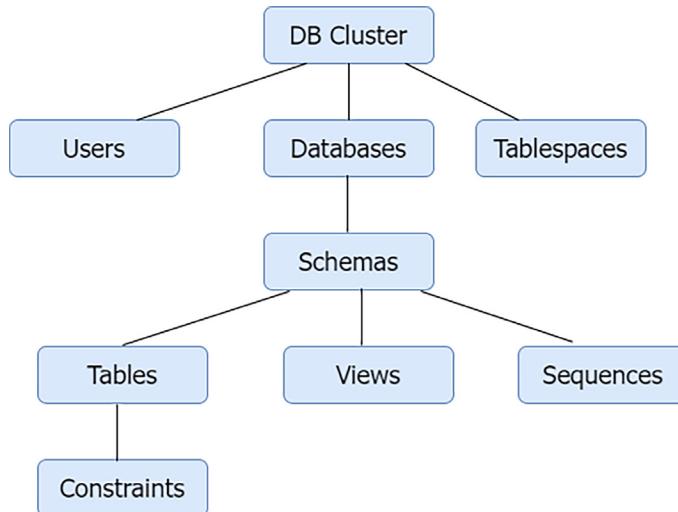


Figure 11.1: Hierarchy of database objects

DB cluster

In *Figure 11.1*, we can see the hierarchy of database objects, where the Database Cluster comes at the top, and all other objects come inside the database cluster, unlike Oracle. A separate data directory, a port, and a postmaster service identify Database Cluster. Also, one machine/DB Server can contain multiple data directories with separate ports that is a database server with high-end configuration can contain multiple database clusters.

When we install postgres, by default database cluster gets created with port 5432 for community postgresql. If we want to create one more database cluster, it can be created using the **initdb** command, which is very simple and easy to use, as shown in *Figure 11.2*:

```

initdb initializes a PostgreSQL database cluster.

Usage:
  initdb [OPTION]... [DATADIR]

Options:
  -A, --auth=METHOD      default authentication method for local connections
    --auth-host=METHOD   default authentication method for local TCP/IP connections
    --auth-local=METHOD   default authentication method for local-socket connections
  [-D, --pgdata=]DATADIR  location for this database cluster
  -E, --encoding=ENCODING set default encoding for new databases
  -g, --allow-group-access allow group read/execute on data directory
  -k, --data-checksums   use data page checksums
    --locale=LOCALE       set default locale for new databases
    --lc-collate=, --lc-ctype=, --lc-messages=LOCALE
    --lc-monetary=, --lc-numeric=, --lc-time=LOCALE
                           set default locale in the respective category for
                           new databases (default taken from environment)
    --no-locale           equivalent to --locale=C
    --pwfile=FILE          read password for the new superuser from file
  -T, --text-search-config=CFG
                           default text search configuration
  -U, --username=NAME     database superuser name
  -W, --pwprompt          prompt for a password for the new superuser
  -X, --waldir=WALDIR     location for the write-ahead log directory
    --wal-segsize=SIZE    size of WAL segments, in megabytes

Less commonly used options:
  -d, --debug            generate lots of debugging output
    --discard-caches     set debug_discard_caches=1
  -L DIRECTORY           where to find the input files
  -n, --no-clean         do not clean up after errors
  -N, --no-sync          do not wait for changes to be written safely to disk
    --no-instructions    do not print instructions for next steps
  -s, --show              show internal settings
  -S, --sync-only        only sync data directory

Other options:
  -V, --version          output version information, then exit
  -?, --help              show this help, then exit

If the data directory is not specified, the environment variable PGDATA
is used.

Report bugs to <pgsql-bugs@lists.postgresql.org>.
PostgreSQL home page: <https://www.postgresql.org/>

```

Figure 11.2: initdb command options

The second layer in this hierarchy has been discussed in detail in *Chapter 4, Global Objects in PostgreSQL*. A database contains at least one tablespace and is accessed by users. Users, Databases, and Tablespace are at the same level, unlike Oracle, where users are created inside the database. Let's summarize the significance of these global objects in short.

Users/roles

Users or roles are the same in PostgreSQL with only one difference users have access to the database, while roles cannot log into the database by default.

Databases

One can create multiple databases inside the database cluster. A database is a collection of multiple schemas and is accessed by users.

Tablespaces

The tablespaces are the locations where the data of the database resides. One database in PostgreSQL can contain multiple tablespaces. However, at the time of database creation, either default or only one tablespace can be selected where all objects will reside by default.

Schemas

A database contains one or more named schemas with tables, functions, and other DB Objects. Schemas are the same as directories/folders of Operating Systems, which help in organizing the files. In the same way, schemas help in managing the tables and other DB Objects. For example, in an application, one can segregate the schemas for Portal or Back Office separately. One can have different schemas for different modules of the projects.

Syntax:

```
CREATE SCHEMA schemaname [ AUTHORIZATION username ];
```

Examples:

```
CREATE SCHEMA test;
```

Default - Public Schema

The **public** schema is the default schema that is created in all the new databases on its own. All the objects get created by default in the **public** schema in case no schema is mentioned at the time of object creation, or the **search_path** is not set.

SEARCH_PATH in Schema

SEARCH_PATH attribute can be used to change the default behavior of creating objects in the **public** schema. If any specific schema name is set in the **search_path**, then objects will be created in that schema instead of the **public** schema. If multiple schemas are given in **search_path**, then objects will be created in the first schema if a schema is not mentioned.

Also, in the case of any SQL statements or DB Objects like functions - objects will be searched in the same order in which schema names are specified in **search_path**. Let's see a few useful commands for accessing and setting **search_path**.

Default **search_path**:

```
postgres=# show search_path;
          search_path
-----
 "$user", public
(1 row)
```

Let's say if we have same user name and schema name say 'test'. When we login using test user , since search path is set as **\$user**, whatever objects we create, will be created in the test schema as shown in the example below:

```
postgres=# create user test;
CREATE ROLE
postgres=# create schema test authorization test;
CREATE SCHEMA
postgres=# \q
```

– log in again using test user

```
$ psql -d postgres -U test
```

```
psql (14.7)
```

Type "help" for help.

```
postgres=> create table test1(n1 int);  
CREATE TABLE  
postgres=> \dt test1
```

List of relations

Schema	Name	Type	Owner
test	test1	table	test

(1 row)

When retrieving data using a **SELECT** statement, it will search the schema in the order in which schema names are mentioned in the **search_path**. Let's take a look at the below:

```
$ psql ----- by default postgres user & postgres DB will  
be considered  
  
psql (14.7)  
Type "help" for help.
```

```
postgres=# insert into test1 values(2);  
ERROR: relation "test1" does not exist  
LINE 1: insert into test1 values(2);  
          ^  
  
postgres=# insert into test.test1 values(2);  
INSERT 0 1  
postgres=# set search_path to "$user", public, test;  
SET  
postgres=# show search_path;  
search_path  
-----
```

```
“$user”, public, test  
(1 row)
```

```
postgres=# insert into test1 values(3);  
INSERT 0 1  
postgres=# select * from test1;  
n1  
----  
1  
2  
3  
(3 rows)
```

In the above example, since **search_path** was not set, PostgreSQL tried to search the table in the **public** schema where it was not found, so an error was given. Once the **search_path** was set, there was no need to mention the schema name while accessing the table.

search_path can be set in **postgresql.conf** file permanently for all users, or it can be altered using **ALTER ROLE** to set it at the global level.

```
postgres=# alter role test set search_path to “$user”, public, test;  
ALTER ROLE  
postgres=# show search_path;  
search_path  
-----  
“$user”, public  
(1 row)  
postgres=> \c postgres test    #connecting with test user  
psql (14.5, server 13.3)  
You are now connected to database “postgres” as user “test”.
```

```
postgres=> show search_path;  
          search_path  
-----  
      "$user", public, test  
(1 row)  
  
postgres=>
```

Managing DB Objects using DDL commands

The acronym **DDL** stands for **Data Definition Language**. As the name suggests, DDL Commands deal with creating DB Objects, majorly table creation. This chapter will demonstrate **CREATE**, **ALTER**, and **DROP** commands.

Data types

Before going deep into tables, let us understand PostgreSQL's most frequently used data types. Considering PostgreSQL as one of the most advanced Open Source **Relational Database Management Systems (RDBMS)**, it has a rich set of multiple data types, and it is complicated to cover all the data types in this book.

Number

- **INTEGER** – occupies 4 bytes - Range: -2,147,483,648 to +2,147,483,647.
- **NUMERIC** - stores up to 131072 digits before the decimal point.
- **NUMERIC(p [, s])** - Stores exact numeric of maximum precision, p, and optional scale, s.

Character

- **CHAR (n)** - Stores fixed-length character string of n characters, blank padded.
- **CHARACTER VARYING(n)** - Stores variable-length character string with a maximum length of n characters.
- **TEXT** - Stores variable length characters having an unlimited length.

Date

- **DATE** - stores the date values only.
- **TIME** - stores the time of day values.
- **TIMESTAMP** - stores both date and time values.
- **INTERVAL** - stores periods of time.

Arrays

- Store an array of strings, an array of integers, and so on., in array columns.

JSON

- PostgreSQL provides two JSON data types: **JSON** (Plain JSON) and **JSONB** (Binary JSON) for storing JSON data.

Boolean

- A Boolean data type can hold one of three possible values: true, false, or null.

Apart from the above most frequent data types, it supports several special data types related to geometric, network, and so on.

Table

A table is one of the fundamental objects of a database that stores data in a structured manner, that is, in rows (records) and columns (fields).

Create table

Creates the table - one of the vital database components which contains data. Table stores the data in a systematic manner in such a way that retrieval of data becomes more accessible.

Syntax:

```
CREATE TABLE table_name
(
    column_name1 data_type,
    column_name2 data_type,
    column_name3 data_type,
    ....
```

```
)  
TABLESPACE tablespace_name;
```

Alter table

Altering a table means changing the definition of a table after its creation.

Syntax:

```
ALTER TABLE table_name    RENAME COLUMN column TO new_column ;  
ALTER TABLE table_name    RENAME TO new_name ;  
ALTER TABLE table_name ALTER COLUMN column_name datatype  
ALTER TABLE table_name    action [, ...] ;
```

Where action is one of:

- ADD column type [column_constraint [...]]
- DROP COLUMN column
- ADD table_constraint
- DROP CONSTRAINT constraint_name

Drop table

DROP Table removes a table from the database. Only the table owner, the schema owner, and the superuser can drop a table. DROP TABLE always removes any indexes, rules, triggers, and constraints for the target table. However, to drop a table referenced by a view or a foreign-key constraint of another table, CASCADE must be specified. CASCADE will remove a dependent view entirely, but in the foreign-key case, it will only remove the foreign-key constraint, not the referencing table entirely.

Syntax :

```
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Truncate

The **TRUNCATE** command empties a table or set of tables. It is faster than **DELETE** as it does not scan the table, unlike **DELETE**. It reclaims the disk space immediately.

Syntax:

```
TRUNCATE <table_list>;
```

View

Indirectly, a view can be used to save a query, which can be used directly as if data is being populated from the table(s) and not using the vast query. Any query which is used frequently, instead, the query is run every time the view is referenced in a query. **CREATE VIEW** defines a view of a query, while the **DROP VIEW** statement is used to drop views like any other DB Object. (Reference: PostgreSQL Community View)

Syntax :

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name [, ...] ) ] [ WITH ( view_option_name [= view_option_value] [, ...] ) ] AS query [ WITH [ CASCDED | LOCAL ] CHECK OPTION ]
```

CREATE MATERIALIZED VIEW defines a materialized view of a query. The query is executed to populate the view when the command is issued (unless **WITH NO DATA** is used) and may be refreshed later using **REFRESH MATERIALIZED VIEW**. **CREATE MATERIALIZED VIEW** is similar to **CREATE TABLE AS**, except that it also remembers the query used to initialize the view so that it can be refreshed later upon demand. A materialized view has many of the same properties as a table, but there is no support for temporary materialized views.

DROP VIEW is used to drop the view the same way **DROP TABLE** is used to drop the table. Also, **ALTER VIEW** is used to alter the view like **ALTER TABLE**.

Sequences

Sequences are used to generate the numbers automatically based on the attributes set at the time of its creation. Typically, they are used for auto-incrementing numbers and are attached to the table's Primary/Unique key columns.

Syntax :

```
CREATE SEQUENCE name [INCREMENT [ BY ] increment][ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue| NO MAXVALUE ][ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE]
```

Sequence Functions:

- **nextval(regclass) -> bigint**
 - Increments sequence value set as per increment value set at the time of creating the sequence and return next incremented value in the sequence.
 - By default, it increments the value by 1.
 - It needs **USAGE** and **UPDATE** privileges on the sequence.
 - Syntax: **SELECT NEXTVAL('SCHEMA_NAME.SEQ_NAME')** where **SCHEMA_NAME** is the schema name and **SEQ_NAME** is the sequence name.
 - Eg: **SELECT NEXTVAL('test_schema.test_seq');**
- **currval(regclass) -> bigint**
 - It gives the latest used value for a specific sequence.
 - It gives session level value, and in case **nextval** is not initiated, then it will give an error.
 - It also needs **USAGE** and **UPDATE** privileges on the sequence.
- **setval (regclass, bigint [, boolean]) → bigint**
 - This function can set a custom value for the sequence.
 - It needs the **UPDATE** privilege on the sequence.
 - For example:
 - **select setval('test_schema.test_seq',99) #Next nextval will return 100**
 - **select setval('test_schema.test_seq',99,true) #Next nextval will return 100**
 - **select setval('test_schema.test_seq',99, false) #Next nextval will return 99**
- **lastval() → bigint**
 - Determines the last value used in the particular session.
 - It gives session level value, and in case **nextval** is not initiated, then it will give an error.
 - This function is identical to **currval**, except that instead of taking the sequence name as an argument, it refers to whichever sequence **nextval** was most recently applied to in the current session. (Reference: PostgreSQL Community Sequences).

Enforcing data integrity using constraints

Let us try to understand the concept of constraints by taking the example of the *dept* and *emp* table as shown in the Entity Relationship Diagram in *Figure 11.3*:

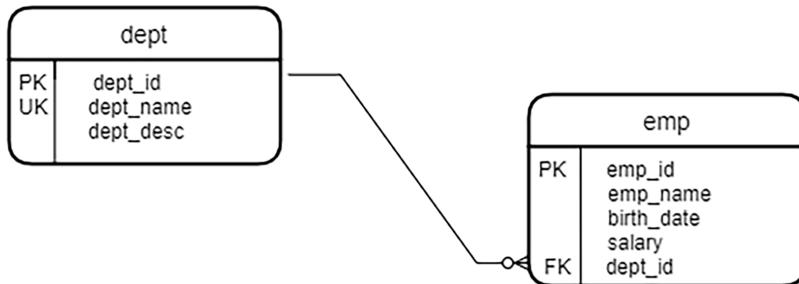


Figure 11.3: Entity Relationship Diagram 1:N relationship of dept and emp

One department in an organization can have zero or more employees, and one employee belongs to one department. With this, there exists a 1:N (one to many) relationships between both tables.

Please find the DDL script and description of each table below, and let us try to understand the constraints in this example.

- dept table:

```

postgres=> create table dept
postgres-> (
postgres(>     dept_id numeric primary key,
postgres(>     dept_name character varying(50) UNIQUE,
postgres(>     dept_desc character varying(100)
postgres(> );
CREATE TABLE

postgres=> \d+ dept
                                         Table "test.dept"
   Column   |          Type          | Collation | Nullable | Default
   | Storage  | Stats target | Description
-----+-----+-----+-----+-----+
dept_id | numeric           |          | not null |
main      |                  |          |
  
```

dept_name	character varying(50)				
extended					
dept_desc	character varying(100)				
extended					

Indexes:

```
"dept_pk1" PRIMARY KEY, btree (dept_id)
"dept_dept_name_key" UNIQUE CONSTRAINT, btree (dept_name)
```

Access method: heap

- emp table:

```
postgres=> CREATE TABLE emp
postgres-> (
postgres(>     emp_id numeric primary key,
postgres(>     emp_name character varying(100) NOT NULL,
postgres(>     birth_date date check (birth_date>'01-01-1950'::date),
postgres(>     salary numeric(9,2),
postgres(>     dept_id numeric references dept(dept_id)
postgres(> );
CREATE TABLE
```

```
postgres=> \d+ emp
```

Table “test.emp9”

Column	Type	Collation	Nullable	Default
Storage	Stats target	Description		
emp_id	numeric		not null	
main				
emp_name	character varying(100)		not null	
extended				
birth_date	date			
plain				

salary	numeric(9,2)			
main				
dept_id	numeric			
main				

Indexes:

“emp_pkey” PRIMARY KEY, btree (emp_id)

Check constraints:

“emp_birth_date_check” CHECK (birth_date > ‘1950-01-01’::date)

Foreign-key constraints:

“emp_dept_id_fkey” FOREIGN KEY (dept_id) REFERENCES dept(dept_id)

Access method: heap

- **Primary Key:** It indicates that a column, or group of columns, can be used as a unique and not null identifier for rows in the table. In our example, **emp_id** should contain values that are unique and not null, so it has been created as the primary key of the table. In the same way, the *dept* table has the **dept_id** as **Primary Key (PK)** for the table. Primary Key creates an index on the primary key column automatically. It is advisable that all tables in the database must have at least one primary key column.
- **Unique Key:** Unique Key ensures that the data contained in a column, or a group of columns, is unique among all the rows in the table. It allows NULL values in the column having UNIQUE constraint as all Nullable columns internally have a unique number in the memory. Whenever a unique constraint is created, an index will also be created for that column. In our example, each department in the organization should have a unique name. Notice that it has created the **dept_dept_name_key** Index automatically.
- **Foreign Key:** A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. This maintains the referential integrity between two related tables. In our example, notice **dept_id** in the *emp* table is referenced from the *dept* table and how the description of the table looks.
- **Check Constraints:** It allows specifying that the value in a particular column must satisfy a Boolean (truth-value) expression. In our example, the **birth_date** column in the *emp* table should have values greater than 01st Jan 1950 and can be checked using the **CHECK** constraint as mentioned in the example:

```
CREATE TABLE emp
(
    emp_id numeric primary key,
    emp_name character varying(100) NOT NULL,
    birth_date date check (birth_date>'01-01-1950'::date),
    salary numeric(9,2)
);
```

- **Not-Null Constraints:** A not-null constraint specifies that a column must not assume the null value. In our example, `emp_name` is not NULL.

```
CREATE TABLE emp
(
    emp_id numeric primary key,
    emp_name character varying(100) NOT NULL,
    birth_date date check (birth_date>'01-01-1950'::date),
    salary numeric(9,2)
);
```

- **Default:** It assigns a default data value for the column whose column definition it appears within. The default expression's data type must match the column's data type. The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null. In our example, `salary` should have a default value of 50000.

```
CREATE TABLE emp
(
    emp_id numeric,
    emp_name character varying(100) NOT NULL,
    birth_date date check (birth_date>'01-01-1950'::date),
    salary numeric(9,2) default 50000
);
```

Manipulating data using DML Queries

Once a table has been created, it is time to insert, update and delete the data. One of the most important operations is to retrieve the data in an organized manner. This section will explore how to manipulate the data using **Data Manipulation Language (DML)** operations.

Let's dive deep into each query used to perform DML.

Inserting data

Creating a table using DDL commands will create an empty table with no data. INSERT statement is used to input the data into the table. Horizontal rows inserted into a table is known as row or record in database terminology.

Different Syntax for INSERT query are:

```
INSERT INTO table_name VALUES (value1, value2, value3,...);
```

OR

```
INSERT INTO table_name (column1, column2, column3,...)VALUES (value1,  
value2, value3,...);
```

OR

```
INSERT INTO table_name (column1, column2, column3,...) select_query
```

Let's insert some records using different types of insert syntaxes:

Dept Table:

```
insert into dept values (1,'HR','Human Resource Dept');  
  
insert into dept (dept_id, dept_name, dept_desc) values  
(2,'IT','Information Technology Dept');  
  
insert into dept (dept_id, dept_name, dept_desc) select 3,'Sales',dept_  
desc from dept where dept_id=1;
```

Emp Table:

```
insert into emp values(1,'Carolina','01-15-1988','50000','1');  
  
insert into emp (emp_id,emp_name,birth_date,salary,dept_id) values  
(2,'John','22-10-1994','71000',2);
```

```
insert into emp values (3,'Peter','09-02-1996','65000',null);  
insert into emp values (3,'Audrey','02-29-1996','68000',2);  
ERROR: duplicate key value violates unique constraint "emp_pkey"  
DETAIL: Key (emp_id)=(3) already exists.
```

Notice how it gives an error if the same primary key value is inserted a second time.

Let's insert a few records.

```
insert into emp values (4,'Audrey','02-29-1996','68000',2);  
insert into emp values(5,'Mollie','07-08-1949','50000',1);  
ERROR: new row for relation "emp" violates check constraint "emp_birth_date_check"  
DETAIL: Failing row contains (5, Mollie, 1949-07-08, 50000.00, 1).
```

It threw an error since the date of birth check constraint is violated as the birth year is 1949, less than 1950. Let's correct the values and insert with new values.

```
insert into emp values(5,'Mollie','07-08-1959','50000',11);  
ERROR: insert or update on table "emp" violates foreign key constraint "emp9_dept_id_fkey"  
DETAIL: Key (dept_id)=(11) is not present in table "dept".
```

We rectified the birth date, but there is a typo this time, and the **dept_id** is entered as 11, which is not present in the *dept* table. This violates referential integrity between the primary key of the *dept* table with the foreign key of the *emp* table. Let's insert emp 5 with dept as null, which is very allowed as the foreign key accepts **NULL** records.

```
insert into emp values(5,'Mollie','07-08-1959','50000',null);
```

Updating data

The **UPDATE** statement is used to update the rows in a table. **UPDATE** changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need to be mentioned in the **SET** clause. **UPDATE** query will only give an error if any constraint is violated with the updated records or there are syntactical errors. It will give the number of rows updated. If no rows are updated, it will say '0' (ZERO) rows are updated.

Syntax:

```
UPDATE [ ONLY ] table
SET column = {expression} [, ...]
[ WHERE condition ]
```

In the same example, let's update the **dept_desc** of **dept_id=3** as it is not correct.

```
update deptset dept_desc = 'Sales & Marketing'
where dept_id =3;
```

Deleting data

DELETE deletes rows that satisfy the **WHERE** clause from the specified table. If the **WHERE** clause is absent, the effect is to delete all rows in the table. The result is a valid but empty table.

Syntax:

```
DELETE from table_name where <condition>;
```

Select (Retrieve) data

The **SELECT** query is used to retrieve the data from the table.

Syntax:

```
SELECT column_1, column_2 , ... column_n
FROM table
WHERE condition
GROUP BY column_list
ORDER BY column_list
```

Let's retrieve data using a simple select query as mentioned below:

```
postgres=> select * from dept;
      dept_id | dept_name | dept_desc
-----+-----+
      1 | HR       | Human Resource Dept
```

```
2 | IT          | Information Technology Dept  
3 | Sales       | Sales & Marketing  
(3 rows)
```

```
postgres=> select * from emp;  
  
emp_id | emp_name | birth_date | salary | dept_id  
-----+-----+-----+-----+  
1 | Carolina | 1988-01-15 | 50000.00 | 1  
2 | John     | 1994-10-10 | 71000.00 | 2  
3 | Peter    | 1996-09-02 | 65000.00 |  
4 | Audrey   | 1996-02-29 | 68000.00 | 2  
5 | Mollie   | 1959-07-08 | 50000.00 |  
(5 rows)
```

Joins used in data retrieval

PostgreSQL JOINS are used to retrieve data from multiple tables. A PostgreSQL **JOIN** is performed whenever two or more tables are joined in a SQL statement.

There are different types of PostgreSQL joins:

- **INNER JOIN** (or simple join)
- **LEFT OUTER JOIN** (or LEFT JOIN)
- **RIGHT OUTER JOIN** (or RIGHT JOIN)
- **FULL OUTER JOIN** (or FULL JOIN)

Inner join

INNER JOINS return all rows from multiple tables where the join condition is met as shown in *Figure 11.4*.

Syntax:

```
SELECT columns  
FROM table1
```

```
INNER JOIN table2
```

```
ON table1.column = table2.column;
```

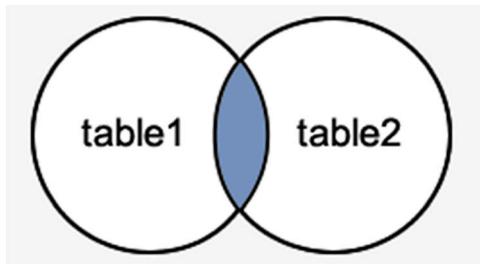


Figure 11.4: INNER join

Let's check the details of all employees who are assigned to a department using inner join:

```
postgres=> select emp_id, emp_name, to_char(birth_date,'DD-Mon-YYYY')
  "Birth Date", salary,d.dept_id, dept_name,dept_desc
  from emp e inner join dept d on e.dept_id = d.dept_id;
      emp_id | emp_name | Birth Date |    salary | dept_id | dept_name | dept_desc
-----+-----+-----+-----+-----+-----+-----+
-----+
      1 | Carolina | 15-Jan-1988 | 50000.00 |        1 | HR       |
Human Resource Dept
      2 | John     | 10-Oct-1994 | 71000.00 |        2 | IT       |
Information Technology Dept
      4 | Audrey   | 29-Feb-1996 | 68000.00 |        2 | IT       |
Information Technology Dept
(3 rows)
```

Notice how **birth_date** has been formatted using **to_char** in the **SELECT** clause. Also, the alias given to the *emp* and *dept* tables along with the “*Birth Date*” column. We can avoid using the alias in case the column name is unique in all the referenced tables. In case the column name is the same (in this case, **dept_id**), then we need to give an alias mandatorily.

Left outer join

This type of join returns all rows from the LEFT-hand table specified in the ON condition and only those rows from the other table where the joined fields are equal (join condition is met), as shown in *Figure 11.5*:

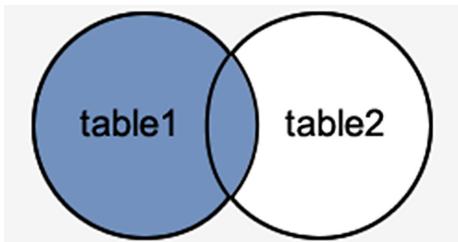


Figure 11.5: LEFT OUTER join

Syntax:

```
SELECT columns
FROM table1
LEFT OUTER JOIN table2
ON table1.column = table2.column;
```

In our example, *emp* is the left table, so all the rows which are in the *emp* table will be shown along with the matching rows on the right side of the table, that is., the *dept* table. Peter and Mollie are not assigned to any *dept*, still, they are shown in the output because of the left outer join.

```
postgres=> select emp_id, emp_name, to_char(birth_date,'DD-Mon-YYYY')
  "Birth Date", salary,d.dept_id, dept_name,dept_desc
  from emp e left outer join dept d on e.dept_id = d.dept_id;
    emp_id | emp_name | Birth Date |   salary | dept_id | dept_name | dept_desc
-----+-----+-----+-----+-----+-----+-----+
-----+
      1 | Carolina | 15-Jan-1988 | 50000.00 |        1 | HR          |
Human Resource Dept
      2 | John      | 10-Oct-1994 | 71000.00 |        2 | IT          |
Information Technology Dept
```

3	Peter	02-Sep-1996	65000.00			
4	Audrey	29-Feb-1996	68000.00	2	IT	
Information Technology Dept						
5	Mollie	08-Jul-1959	50000.00			

(5 rows)

Right outer join

This type of join returns all rows from the RIGHT-hand table specified in the ON condition and only those rows from the other table where the joined fields are equal (join condition is met) as shown in *Figure 11.6*:

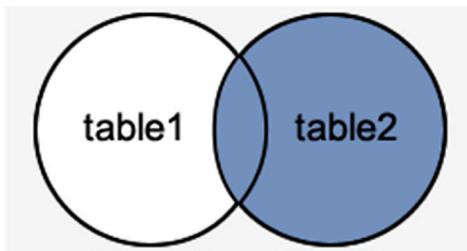


Figure 11.6: RIGHT OUTER join

Syntax:

```
SELECT columns
FROM table1
RIGHT OUTER JOIN table2
ON table1.column = table2.column;
```

In our example, *dept* is the right table, so all the rows which are in the *dept* table will be shown along with the matching rows on the left side of the table, that is, *emp* table. The Sales dept does not have any employees assigned, but still, they are shown in the output because of the right outer join.

```
postgres=> select emp_id, emp_name, to_char(birth_date,'DD-Mon-YYYY')  
"Birth Date", salary,d.dept_id, dept_name,dept_desc from emp e right  
outer join dept d on e.dept_id = d.dept_id;  
  
emp_id | emp_name | Birth Date | salary | dept_id | dept_name |  
dept_desc
```

```

-----+-----+-----+-----+-----+-----+
-----+
      1 | Carolina | 15-Jan-1988 | 50000.00 |           1 | HR        |
      Human Resource Dept
      2 | John      | 10-Oct-1994 | 71000.00 |           2 | IT        |
      Information Technology Dept
      4 | Audrey    | 29-Feb-1996 | 68000.00 |           2 | IT        |
      Information Technology Dept
      |           |           |           |           3 | Sales     |
      Sales & Marketing
(4 rows)

```

Full outer join

This type of join returns all the rows from the LEFT -hand table and RIGHT - hand tables with nulls in place where the join condition is not met as shown in *Figure 11.7*:

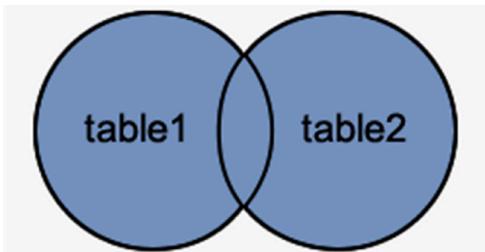


Figure 11.7: FULL OUTER join

Syntax:

```

SELECT columns
FROM table1
FULL OUTER JOIN table2
ON table1.column = table2.column;

```

Both right and left side data will be shown in case of **FULL OUTER JOIN**. In this example, Peter and Mollie are shown without any dept data. Also, the Sales dept is shown with NULL employees.

```

postgres=> select emp_id, emp_name, to_char(birth_date,'DD-Mon-YYYY')
  "Birth Date", salary,d.dept_id, dept_name,dept_desc
postgres-> from emp e full outer join dept d on e.dept_id = d.dept_id;

emp_id | emp_name | Birth Date | salary | dept_id | dept_name | dept_desc
-----+-----+-----+-----+-----+-----+
-----+
1 | Carolina | 15-Jan-1988 | 50000.00 | 1 | HR | Human Resource Dept
2 | John | 10-Oct-1994 | 71000.00 | 2 | IT | Information Technology Dept
3 | Peter | 02-Sep-1996 | 65000.00 | | |
4 | Audrey | 29-Feb-1996 | 68000.00 | 2 | IT | Information Technology Dept
5 | Mollie | 08-Jul-1959 | 50000.00 | | |
| | | | | 3 | Sales | |
Sales & Marketing
(6 rows)

```

Aggregate functions

The Aggregate Functions are used to club the multiple rows into single rows by grouping the rows based on columns mentioned. Common aggregate functions are **SUM, MIN, MAX, AVG, STRING_AGG**, and many more.

SELECT Syntax:

```

SELECT <Column_List>, <aggregate_function>
FROM table1
GROUP BY <Column_List>
ORDER BY <Column_List>

```

Count the number of employees in each dept. Also, give minimum salary and maximum salary in that department.

```
postgres=> select d.dept_id,d.dept_name,count(1),min(e.salary),max(e.
salary) from emp e right outer join dept d on e.dept_id = d.dept_id
group by d.dept_id,d.dept_name order by 1;
```

dept_id	dept_name	count	min	max
1	HR	1	50000.00	50000.00
2	IT	2	68000.00	71000.00
3	Sales	1		

(3 rows)

As we can see, we have used **GROUP BY** clause to retrieve such data. It also shows **NULL** rows whose **dept_id** is null.

Conclusion

In this chapter we have tried to cover the basics of different database objects like tables, views, sequences, and so on, from a PostgreSQL developer perspective. These are SQL queries / commands which are helpful for PostgreSQL DBAs too.

In the next chapter, we will touch on the topics from PostgreSQL PL/pgSQL perspective, which again will be very much useful in case anyone wants to build a career in PostgreSQL Development.

Bibliography

- PostgreSQL Community View: <https://www.postgresql.org/docs/current/sql-createview.html>
- PostgreSQL Community Sequences: <https://www.postgresql.org/docs/current/functions-sequence.html>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 12

Advance Database Objects

Introduction

In the previous chapter, we learned the basics of database objects, like tables, views, sequences, and the like. Also, we learned how to manipulate data using **SELECT**, **INSERT**, **UPDATE** and **DELETE** queries. We also touched on one of the most important SQL concepts of **JOINS**. In this chapter, we will focus on working with PL/pgSQL concepts like **FUNCTIONS**, **TRIGGERS**, and much more.

Structure

In this chapter, we will cover the following topics:

- Managing procedures / functions
- Managing triggers
- Managing rules
- Custom Data types

Objectives

In this chapter, we will learn about managing advance database objects in PostgreSQL, like Procedures, Functions, Triggers, and the like., and how to work with PL/PgSQL. We will also have some insights into the custom data types used in **Relational Database Management Systems (RDBMS)**.

Managing procedures/functions

Let us understand how to work with procedures and functions in PostgreSQL. Let us begin with functions first:

Function

Functions are the DB Objects which can execute multiple SQL queries in a sequence and returns only one output. A function cannot use COMMIT or ROLLBACK in its code. One must have usage privileges on the arguments and return type of the function to be able to use them. The input and out parameters are considered as the signature of the function, and since PostgreSQL supports OOPs concepts, one can overload its signature.

Syntax as per PostgreSQL documentation: (*Reference: PostgreSQL Community Function*):

```
CREATE [ OR REPLACE ] FUNCTION  
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_  
        expr ] [ , ... ] ] )  
    [ RETURNS rettype  
    | RETURNS TABLE ( column_name column_type [ , ... ] ) ]  
    { LANGUAGE lang_name  
    | TRANSFORM { FOR TYPE type_name } [ , ... ]  
    | WINDOW  
    | { IMMUTABLE | STABLE | VOLATILE }  
    | [ NOT ] LEAKPROOF
```

```
| { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT }  
| { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER  
}  
| PARALLEL { UNSAFE | RESTRICTED | SAFE }  
| COST execution_cost  
| ROWS result_rows  
| SUPPORT support_function  
| SET configuration_parameter { TO value | = value | FROM CURRENT  
}  
| AS 'definition'  
| AS 'obj_file', 'link_symbol'  
| sql_body  
} ...
```

If the schema name is mentioned while creating the function, then that function will be created in that schema like any other DB Object. Else it will take the schema name from the **search_path**.

Generic Syntax, which is commonly used, is as follows:

```
CREATE OR REPLACE FUNCTION func_name (<input params>) RETURN rettype  
IS  
    declaration_statements  
BEGIN  
    statements;  
EXCEPTION  
    WHEN exception THEN  
        statement;  
END function_name
```

Function execution syntax

Functions can be used in the DML queries, and their output can be integrated along with the query results. Due to this, it can be called using the SELECT clause.

```
select schema_name.func_name (<input_params>);
```

Let us see an example of the function.

We will use the same tables we created in *Chapter 11, Basic Database Objects*, as shown in *Figure 12.1*, where the **dept** table is the master table and the **emp** table is the child table containing a foreign key of the **dept_id** of the **dept** table. One department can have multiple employees that is there exists a **one-to-many (1:M)** relationship between **dept** and **emp** tables.

```
postgres=# \d+ dept
                                         Table "test.dept"
  Column   |      Type       | Collation | Nullable | Default | Storage | Stats target | Description
  dept_id  | numeric        |           | not null |          | main    |              |
  dept_name | character varying(50) |           |           |          | extended |              |
  dept_desc | character varying(100) |           |           |          | extended |              |
Indexes:
  "dept_pk1" PRIMARY KEY, btree (dept_id)
  "dept_dept_name_key" UNIQUE CONSTRAINT, btree (dept_name)
Referenced by:
  TABLE "emp" CONSTRAINT "emp_dept_id_fkey" FOREIGN KEY (dept_id) REFERENCES dept(dept_id)
Access method: heap

postgres=# \d+ emp
                                         Table "test.emp"
  Column   |      Type       | Collation | Nullable | Default | Storage | Stats target | Description
  emp_id   | numeric        |           | not null |          | main    |              |
  emp_name | character varying(100) |           | not null |          | extended |              |
  birth_date | date          |           |           |          | plain   |              |
  salary    | numeric(9,2)   |           |           |          | main    |              |
  dept_id   | numeric        |           |           |          | main    |              |
Indexes:
  "emp_pkey" PRIMARY KEY, btree (emp_id)
Check constraints:
  "emp_birth_date_check" CHECK (birth_date > '1950-01-01'::date)
Foreign-key constraints:
  "emp_dept_id_fkey" FOREIGN KEY (dept_id) REFERENCES dept(dept_id)
Access method: heap
```

Figure 12.1 Definition of dept and emp tables

Below function will return the count of the employees. It takes **dept_id** as an input parameter. In case there are no employees in any department then it will return 0 as the inbuild function count will return 0 in case no data is available:

```
CREATE OR REPLACE FUNCTION GET_DEPT_CNT(P_DEPT_ID NUMERIC)
RETURNS CHARACTER VARYING AS
$BODY$
```

```
DECLARE  
  
    V_DEPT_CNT CHARACTER VARYING;  
  
BEGIN  
  
    SELECT COUNT(E.DEPT_ID)  
    INTO V_DEPT_CNT  
    FROM DEPT D,  
         EMP E  
    WHERE D.DEPT_ID = E.DEPT_ID  
    AND D.DEPT_ID = P_DEPT_ID;  
  
    RETURN V_DEPT_CNT;  
  
EXCEPTION  
    WHEN OTHERS THEN  
        RAISE NOTICE 'Some error occurred';  
END$BODY$  
  
LANGUAGE plpgsql VOLATILE;
```

Function execution example

Now let us see how to execute the function with and without query:

```
#Calling function without query:  
  
postgres=# SELECT GET_DEPT_CNT(8);  
get_dept_cnt  
-----  
0  
(1 row)
```

#Calling function with query:

```
postgres=# SELECT *,GET_DEPT_CNT(DEPT_ID) FROM DEPT;
dept_id | dept_name | dept_desc | get_dept_cnt
-----+-----+-----+-----+
 1 | HR | Human Resource Dept | 1
 2 | IT | Information Technology Dept | 2
 3 | Sales | Sales & Marketing | 0
(3 rows)
```

Procedure

Procedures are the DB Objects which can execute multiple SQL queries in a sequence. In PostgreSQL, one can code in multiple languages like PL/pgSQL, PL/Pearl, and PL/Python - to name a few. Procedures can accept multiple inputs and outputs as parameters mentioned at the time of the creation of the procedure. Also, we can commit or rollback the DML statements depending on the procedure's logic. A procedure is called using **EXEC** or **CALL** statements. One must have usage privileges on the arguments to create and execute procedures.

Syntax as per PostgreSQL documentation: (*Reference: PostgreSQL Community Procedure*):

```
CREATE [ OR REPLACE ] PROCEDURE
  name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_
expr ] [ , ... ] )
{ LANGUAGE lang_name
  | TRANSFORM { FOR TYPE type_name } [ , ... ]
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  | SET configuration_parameter { TO value | = value | FROM CURRENT
}
  | AS 'definition'
  | AS 'obj_file', 'link_symbol'
  | sql_body
```

```
    } ...
```

Generic Syntax which is commonly used:

```
CREATE OR REPLACE PROCEDURE procedure_name ([ argname [ IN | IN OUT
| OUT ] argtype [ DEFAULT value ] )
IS
declaration_statements
BEGIN
statements;
EXCEPTION
WHEN exception THEN
statement;
END procedure_name
```

Like any other programming language, procedures contains a declaration section and transaction with the **BEGIN... END** block, which contains the main code of the procedure. Also, before the **END** clause, one can mention the exception (if any) depending upon the logic of the code.

The body of the procedure, which is inside **BEGIN...END** block can contain code in multiple languages like SQL, C, Python, and many more, as mentioned earlier.

Procedure execution syntax

Let us see how we execute or run the code written in a procedure:

Normal procedure execution (without cursor):

```
call schemaname.procedure_name(input_parameters_seperated_by_commas);
```

Execution of procedure returning cursor with the “**FETCH**” statement:

```
BEGIN; -- Start the transaction
call schemaname.procedure_name(input_parameters_seperated_by_commas);
--Returns : "<unnamed portal n>" i.e. cursor >
FETCH ALL IN "<unnamed portal n>";
END;
```

Let us see an example of the procedure which will call the function which we had created in the previous section.

```
CREATE OR REPLACE PROCEDURE P_EMP_DTLS
(
    P_EMP_ID IN NUMERIC,
    P_DEPT_ID IN NUMERIC,
    INOUT P_OUT_CURSOR REFCURSOR,
    INOUT P_SUCCESS CHARACTER VARYING
)
LANGUAGE 'plpgsql'

AS
$BODY$
DECLARE
    V_DEPT_EMP_CNT NUMERIC;

BEGIN
    P_SUCCESS = 'Please enter EMP_ID or DEPT_ID';
    IF P_EMP_ID IS NOT NULL THEN
        OPEN P_OUT_CURSOR FOR
            SELECT *
            FROM EMP E
            WHERE E.EMP_ID = P_EMP_ID;
        P_SUCCESS = 'EMP DETAILS';
    END IF;

    IF P_DEPT_ID IS NOT NULL THEN
```

```

OPEN P_OUT_CURSOR FOR

    SELECT *,GET_DEPT_CNT(P_DEPT_ID)

    FROM EMP E

    WHERE E.DEPT_ID = P_DEPT_ID;

    P_SUCCESS = 'EMP DETAILS WITH DEPT COUNT';

    END IF;

EXCEPTION

WHEN OTHERS THEN

    P_SUCCESS = 'SOME ERROR OCCURRED';

    RAISE NOTICE '%',SQLERRM;

END;

$BODY$;

```

The above procedure takes 2 input parameters, **emp_id**, and **dept_id**. If **emp_id** is passed as an input parameter, it will give details of that employee in the cursor variable **p_out**. It will show the “Emp Details” message in the output parameter **P_SUCCESS**.

If **emp_id** is null and **dept_id** is entered, then it will call the function in the query, which returns the count of employees in that department along with respective employee details. It will show the “Dept Details” message in the output parameter **P_SUCCESS**.

In case both **emp_id** and **dept_id** are not passed, then it shows “**Please enter EMP_ID or DEPT_ID**” in the **P_SUCCESS** parameter, and the cursor will be NULL.

Procedure execution

Let us see how we execute or run the code written in the procedure in the above example with all the scenarios mentioned.

Execution of procedure returning cursor:

```
postgres=# select * from emp;
   emp_id | emp_name | birth_date | salary | dept_id
-----+-----+-----+-----+
      1 | Carolina | 1988-01-15 | 50000.00 |      1
      2 | John     | 1994-10-10 | 71000.00 |      2
      3 | Peter    | 1996-09-02 | 65000.00 |
      4 | Audrey   | 1996-02-29 | 68000.00 |      2
      5 | Mollie   | 1959-07-08 | 50000.00 |

(5 rows)
```

```
postgres=# BEGIN;
BEGIN
postgres=# call P_EMP_DTLS (1,NULL,NULL,NULL);
      p_out_cursor | p_success
-----+-----
<unnamed portal 1> | EMP DETAILS
(1 row)
```

```
postgres=# fetch all in "<unnamed portal 1>";
   emp_id | emp_name | birth_date | salary | dept_id
-----+-----+-----+-----+
      1 | Carolina | 1988-01-15 | 50000.00 |      1
(1 row)
```

```
postgres=# END;
```

```
COMMIT
```

```
postgres=# BEGIN;
```

```
BEGIN
```

```
postgres=# call P_EMP_DTLS (NULL,1,NULL,NULL);
```

p_out_cursor		p_success
-----+-----		

```
<unnamed portal 2> | EMP DETAILS WITH DEPT COUNT
```

```
(1 row)
```

```
postgres=# fetch all in "<unnamed portal 2>";
```

emp_id		emp_name		birth_date		salary		dept_id		get_dept_cnt
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----										

```
1 | Carolina | 1988-01-15 | 50000.00 | 1 | 1
```

```
(1 row)
```

```
postgres=# END;
```

```
COMMIT
```

```
postgres=*
```

```
postgres=*
```

```
postgres=# BEGIN;
```

```
BEGIN
```

```
postgres=# call P_EMP_DTLS (NULL,NULL,NULL,NULL);
```

p_out_cursor		p_success
-----+-----		

```
| Please enter EMP_ID or DEPT_ID
```

```
(1 row)
```

```
postgres=# END;  
COMMIT  
postgres=#
```

Managing triggers

Triggers are created when any code needs to be executed **BEFORE**, **AFTER**, or **INSTEAD** of the Data Manipulations operations like **DELETE**, **UPDATE**, and **INSERT**. With the help of triggers, one can perform one more additional check while manipulating the data in the Database. Triggers in PostgreSQL are part of the table and are created for a specific table. As the name suggests, it is executed when some event is triggered.

To create a new trigger in PostgreSQL, one needs to

1. Create a trigger function using **CREATE FUNCTION** statement.
2. Bind this trigger function to a table using **CREATE TRIGGER** statement.

A trigger function is similar to a normal function, except that it does not take any arguments and has a return value type trigger:

Syntax:

```
CREATE FUNCTION trigger_function() RETURN trigger AS
```

Trigger function

When a PL/pgSQL function is called a trigger, several special attributes are created by default in PostgreSQL. They are as follows:

- **NEW**: The data type of this attribute is **RECORD**. It holds the the new database row for **INSERT/UPDATE** operations in row-level triggers. In statement-level triggers, it does not contain any value in case of **DELETE** operations.
- **OLD**: : The data type of this attribute is **RECORD**. It holds the old database row for **UPDATE/DELETE** operations in row-level triggers. In statement-level triggers, it does not contain any value in case of **INSERT** operations.
- **TG_NAME**: The data type of this attribute is name; It contains the trigger's name that is currently fired.
- **TG_WHEN**: The data type of this attribute is text. It returns a string of **BEFORE**, **AFTER**, or **INSTEAD OF**, depending on the trigger's definition.

- **TG_OP:** The data type of this attribute is text. It returns a string of **INSERT**, **UPDATE**, **DELETE**, or **TRUNCATE** telling for which operation the trigger was fired.
- **TG_TABLE_NAME:** The data type of this attribute is name. It returns the name of the table on which trigger was invoked.
- **TG_TABLE_SCHEMA:** The data type of this attribute is name. It returns the name of the table schema on which trigger was invoked.
- A trigger function must return either **NULL** or a record/row value having exactly the structure of the table for which the trigger was fired for.

Let us try to understand triggers with the help of an example. One of the major uses of triggers is for auditing purposes. We can trace at what time the data was updated/deleted, who deleted the same, and other such information required for auditing. (*Reference: PostgreSQL Community Trigger*)

Let us create a table that contains auditing columns like what operation is being performed - **INSERT**, **UPDATE**, etc., what is the date and time of operation, which user did the same, etc.

```
CREATE TABLE test.emp_hist(
    operation      char(1)    NOT NULL,
    date_time     timestamp NOT NULL,
    userid        text       NOT NULL,
    emp_id        numeric    NOT NULL,
    emp_name      varchar(100),
    birth_date    date,
    salary        numeric(9,2),
    dept_no       numeric
);
```

Let us now create the trigger function, which will insert data in the **emp_hist** table at the time of DML operations.

```
CREATE OR REPLACE FUNCTION test.emp_trg() RETURNS trigger AS $emp_trg$
BEGIN
    IF (TG_OP = 'DELETE') THEN
```

```
        INSERT INTO emp_hist SELECT 'D', now(), user, OLD.*;

        RETURN OLD;

ELSIF (TG_OP = 'UPDATE') THEN

    IF NEW.salary IS NULL or NEW.salary <= 0 THEN

        RAISE EXCEPTION 'Please enter valid salary for %', NEW.
            emp_name;

    END IF;

    INSERT INTO emp_hist SELECT 'U', now(), user, OLD.*;

    RETURN NEW;

ELSIF (TG_OP = 'INSERT') THEN

    IF NEW.salary IS NULL or NEW.salary <= 0 THEN

        RAISE EXCEPTION 'Please enter valid salary for %', NEW.
            emp_name;

    END IF;

    INSERT INTO emp_hist SELECT 'I', now(), user, NEW.*;

    RETURN NEW;

END IF;

RETURN NULL; -- result is ignored since this is an AFTER
trigger

END;

$emp_trg$ LANGUAGE plpgsql;
```

Finally, let us create the trigger on the **emp** table:

```
CREATE TRIGGER emp_auditing
AFTER INSERT OR UPDATE OR DELETE ON test.emp
FOR EACH ROW EXECUTE PROCEDURE test.emp_trg();
```

Let us test the code of trigger by performing **INSERT**, **UPDATE** and **DELETE** operations

on the **emp** table:

Data in the **emp** table is as below:

```
postgres=# select * from emp order by 1;
emp_id | emp_name | birth_date | salary | dept_id
-----+-----+-----+-----+
1 | Carolina | 1988-01-15 | 50000.00 | 1
2 | John     | 1994-10-10 | 71009.00 | 2
3 | Peter    | 1996-09-02 | 65000.00 |
4 | Audrey   | 1996-02-29 | 68000.00 | 2
5 | Mollie   | 1959-07-08 | 50000.00 |

(5 rows)
```

However, there are no records in the **emp_hist** table:

```
postgres=# select * from emp_hist;
operation | date_time | userid | emp_id | emp_name | birth_date | salary | dept_no
-----+-----+-----+-----+-----+-----+-----+
-----+-----+
(0 rows)
```

Let us insert some values in the **emp** table and check how they are reflected in the **emp_hist** table, as the trigger has been written on the **emp** table.

```
postgres=# insert into emp values (6,'Trigger Test Emp','1980-01-09',69000,3);
INSERT 0 1

postgres=# select * from emp_hist;
operation |          date_time          | userid | emp_id |      emp_
name      | birth_date | salary | dept_no
-----+-----+-----+-----+-----+
-----+-----+-----+-----+
I          | 2023-01-24 20:27:38.074769 | postgres |       6 | Trigger
```

```
Test Emp | 1980-01-09 | 69000.00 |          3  
(1 row)
```

Let us update the newly inserted record and check its effect on the audit table **emp_hist**.

```
postgres=# update emp set salary = 96000 where emp_id = 6;  
UPDATE 1  
postgres=# select * from emp_hist;  
operation |           date_time           | userid | emp_id |      emp_  
name     | birth_date | salary | dept_no  
-----+-----+-----+-----+  
I       | 2023-01-24 20:27:38.074769 | postgres |      6 | Trigger  
Test Emp | 1980-01-09 | 69000.00 |          3  
U       | 2023-01-24 20:29:12.715331 | postgres |      6 | Trigger  
Test Emp | 1980-01-09 | 69000.00 |          3  
(2 rows)
```

```
postgres=# select * from emp where emp_id=6;  
emp_id |      emp_name      | birth_date | salary | dept_id  
-----+-----+-----+-----+  
6 | Trigger Test Emp | 1980-01-09 | 96000.00 |          3  
(1 row)
```

As shown above, the **emp_hist** table contains the old values before the update occurred.

Let us delete the same row and check whether we get the actual values before deleting the data:

```
postgres=# delete from emp where emp_id = 6;  
DELETE 1  
postgres=# select * from emp_hist;
```

```

operation |          date_time          | userid | emp_id |      emp_
name     | birth_date | salary | dept_no
-----+-----+-----+-----+-----+
-----+-----+-----+-----+
I       | 2023-01-24 20:27:38.074769 | postgres |      6 | Trigger
Test Emp | 1980-01-09 | 69000.00 |      3
U       | 2023-01-24 20:29:12.715331 | postgres |      6 | Trigger
Test Emp | 1980-01-09 | 69000.00 |      3
D       | 2023-01-24 20:31:45.24357  | postgres |      6 | Trigger
Test Emp | 1980-01-09 | 96000.00 |      3
(3 rows)

```

```

postgres=# select * from emp where emp_id=6;
emp_id | emp_name | birth_date | salary | dept_id
-----+-----+-----+-----+-----+
(0 rows)

```

Event trigger

Event Triggers are called when DDL statements like **CREATE**, **ALTER**, and **DROP** statements are executed. Just like Trigger Functions which are invoked on DML statements, event triggers also have trigger function with no argument and it returns the event trigger. It also contains special variable as below:

- **TG_EVENT**: The data type of this variable is text. It returns a string representing the event the trigger on which it was fired.
- **TG_TAG**: The data type of this variable is text. It contain the variable that contains the command tag for which the trigger is fired.

Managing rules

Rules in PostgreSQL creates additional commands to be executed whenever any table data is modified/inserted on the table where rule has been created. At the same time, it also can execute the code mentioned in **INSTEAD** block. The majorly to create

additional security on specific table data. This will be more clear with the example mentioned later in this section. (*Reference: PostgreSQL Community Rules*)

Syntax as per PostgreSQL Documentation:

```
CREATE [ OR REPLACE ] RULE name AS ON event  
    TO table_name [ WHERE condition ]  
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command  
... ) }
```

where event can be one of:

```
SELECT | INSERT | UPDATE | DELETE
```

One of the use cases of using Rules is to protect data. For example, if we delete the **ADMIN** entry from the **USER_MST** table of an application, then no other user will be created as **ADMIN** is the one who creates the other users and assign rights/privileges. The **ADMIN** entry should be deleted accidentally too. To protect **ADMIN** entry in **USER_MST** rule can be created as below:

```
CREATE OR REPLACE RULE user_data_protect1 AS
```

```
ON UPDATE TO user_mst
```

```
WHERE OLD.user_name = 'ADMIN'
```

```
DO NOTHING INSTEAD
```

```
CREATE OR REPLACE RULE user_data_protect2 AS
```

```
ON DELETE TO user_mst
```

```
WHERE OLD.user_name = 'ADMIN'
```

```
DO NOTHING INSTEAD
```

Let us try to update and delete records:

```
postgres=# SELECT * FROM USER_MST;
```

```
user_id | user_name
```

```
-----+-----
```

```
1 | ADMIN
2 | SomeUser
3 | OneMoreUser
(3 rows)
```

```
postgres=# DELETE FROM USER_MST WHERE USER_NAME = 'ADMIN';
```

```
DELETE 0
```

```
postgres=# SELECT * FROM USER_MST;
```

```
user_id | user_name
-----+-----
1 | ADMIN
2 | SomeUser
3 | OneMoreUser
(3 rows)
```

```
postgres=# UPDATE USER_MST SET USER_NAME = 'ADMIN-UPDATED' WHERE USER_ID = 1;
```

```
UPDATE 0
```

```
postgres=# SELECT * FROM USER_MST;
```

```
user_id | user_name
-----+-----
1 | ADMIN
2 | SomeUser
3 | OneMoreUser
(3 rows)
```

Trigger versus rules

Let us try to understand the difference between Trigger and Rules as mentioned below:

- A trigger will be used on DML Operations as a whole DML Operation and not on specific data.
- A rule can be defined for the specific data of the table.
- Rules are faster than triggers.

Custom data type

Along with inbuild data types, PostgreSQL also provides functionality to create custom data types. **CREATE TYPE** is used to create user-defined data types. One can create five different types as below: (*Reference: PostgreSQL Community Type*)

- composite type,
- enum type,
- range type,
- base type,
- shell type

Each of these types has its own significance. Discussion on each of these type is out of the scope of this book; however, let us take one of them to understand how custom types works

Let us try to understand the **ENUM** type. In the below example, **ENUM** type named **GENDER** has been created with the values 'Male', 'Female', and 'Others'. These are case-sensitive values, as shown in the example below:

```
postgres=# CREATE TYPE GENDER AS ENUM ('Male','Female','Others');

CREATE TYPE

postgres=# CREATE TABLE STUDENT (
postgres(# ROLL_NO SERIAL,
postgres(# STUDENT_NAME TEXT,
postgres(# STUDENT_GENDER GENDER
postgres(# );

CREATE TABLE
```

```
postgres=# INSERT INTO STUDENT VALUES(1,'TEST','Male');

INSERT 0 1

postgres=# INSERT INTO STUDENT VALUES(2,'TEST2','FeMale');

ERROR: invalid input value for enum gender: "FeMale"

LINE 1: INSERT INTO STUDENT VALUES(2,'TEST2','FeMale');

^

postgres=# INSERT INTO STUDENT VALUES(2,'TEST2','Female');

INSERT 0 1
```

Conclusion

This chapter covered advanced database objects like procedures, functions, triggers, and rules from a PostgreSQL PL/PgSQL perspective.

In the next chapter, we will touch on more advanced concepts essential from the performance tuning perspective, like Indexes, Statistics, and Explain Plans, along with best practices for the postgresql parameters. These topics are good to know for PostgreSQL Developer as well as PostgreSQL DBA.

Bibliography

- PostgreSQL Community Procedure: <https://www.postgresql.org/docs/14/sql-createprocedure.html>
- PostgreSQL Community Function: <https://www.postgresql.org/docs/14/sql-createfunction.html>
- PostgreSQL Community Trigger: <https://www.postgresql.org/docs/current/plpgsql-trigger.html>
- PostgreSQL Community Rules: <https://www.postgresql.org/docs/14/sql-createrule.html>
- PostgreSQL Community Type: <https://www.postgresql.org/docs/14/sql-createtype.html>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 13

Performance Tuning

Introduction

Any **Relational Database Management System (RDBMS)** intends to resolve user queries as quickly as possible, and PostgreSQL is no exception. So, keeping postgres well performant is one of the main goals for **Database Administrators (DBAs)**.

In previous chapters, we have studied many of the capabilities of PostgreSQL and how to configure and use them. Now, we will dive into the performance tuning topics. We will learn about the basic concepts and components involved in database tuning and some best practices we can use as an initial guide.

Structure

During this chapter, we will study the next sections:

- Indexes
- Statistics
- Explain plan
- Best practices for the `postgresql.conf` parameters

Objectives

Once you complete this chapter, you will be familiar with the basic components for tuning in PostgreSQL and understand their relationship and impact. Also, you will relate the standard best practices when installing or tuning a postgres system to deliver the best performance.

Indexes

Have you ever noticed the book indexes that help locate the chapters by mentioning the page number of the chapter? One can jump to that page directly rather than going through each chapter sequentially which takes a lot of time. In the same way, indexes also create a similar kind of metadata like page numbers which helps to jump to the requested data directly instead of scanning the whole table sequentially.

This benefit comes with an overhead since adding an index involves it being created / modified as data gets inserted, updated, or deleted from the table. This is one of the reasons one should create wisely through proper analysis. If the table is used for read-only operations more than the write operations, then it could be beneficial to create the indexes; else, it might become an overhead to update the index on every write operation.

Syntax as per PostgreSQL Documentation: (*Reference - PostgreSQL Community Index*)

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON
[ ONLY ] table_name [ USING method ]

( { column_name | ( expression ) } [ COLLATE collation ] [ opclass
[ ( opclass_parameter = value [, ...] ) ] [ ASC | DESC ] [ NULLS
{ FIRST | LAST } ] [, ...] )

[ INCLUDE ( column_name [, ...] ) ]

[ NULLS [ NOT ] DISTINCT ]

[ WITH ( storage_parameter [= value] [, ...] ) ]

[ TABLESPACE tablespace_name ]

[ WHERE predicate ]
```

Generic Syntax normally used:

```
CREATE INDEX <schema_name>.<index_name> on <table_name> (<column/
column-list>)
```

CREATEing and DROPPing the index is not an online activity by default. The table is locked for any modification. However, read operations can be performed while the index is being created/modified by including the **CONCURRENTLY** keyword in the index creation statement. It will help to modify indexes concurrently online, as shown in the syntax as per PostgreSQL documentation above.

Indexes on multiple columns can also be created, which are called Multi Column Indexes. While creating such indexes, make sure that the order of columns is the same as used in the query's **WHERE** clause.

Reindex

Reindexing an index is one of the maintenance activities in PostgreSQL, which helps to rebuild the corrupted/bloated/invalid index concurrently. An index can also be rebuilt when storage is changed for the specific index.

Syntax as per PostgreSQL Documentation: (*Reference - PostgreSQL Community Re Index*):

```
REINDEX [ ( option [, ...] ) ] { INDEX | TABLE | SCHEMA | DATABASE |  
SYSTEM } [ CONCURRENTLY ] name
```

where **option** can be one of:

CONCURRENTLY [**boolean**] - Reindex can be created concurrently online

TABLESPACE **new_tablespace** - *Reindexing could be done in new tablespace*

VERBOSE [**boolean**] - Logs will be printed while reindexing

Whenever a Primary Key or Unique Key constraints are created, by default the system creates indexes on such columns as commonly those columns will be extensively used for fetching the data in the queries. Generally, the indexes could be created on columns used to filter data, that is, WHERE clause columns. It is not a thumb rule; however, it is a good practice to create indexes on those columns that are used as the child table's foreign key. Typically such columns will be used in join conditions to fetch the data.

Indexes might not be needed for small databases initially, as default indexes created on Primary Key or Unique are enough. However, as time goes on, data also increases, and it takes more time than it usually takes. One can use **EXPLAIN PLAN** to find out

whether indexes are used by the query to fetch the data. **EXPLAIN PLAN** is discussed in detail later in this chapter.

Indexes are one of the building blocks to improve the performance of the queries. These queries could be the reporting queries that fetch data or any **SELECT** queries that need improvement in the query execution time.

Indexes might also make the performance of **INSERT**, **UPDATE**, and **DELETE** queries in which a **WHERE** clause is used for search conditions. The **ANALYZE** command is recommended so that statistics of the indexes can be kept up to date, and whenever it is used, it will contain updated metadata which will help to enhance performance better.

Index types

Each query is different and may not give the best results with the default BTREE index. To overcome this, PostgreSQL has different types of indexes, which help increase the performance of various kinds of queries. If one wants to create a specific index type apart from the default BTREE index, it can be done with the **USING** keyword in the index creation statement. Let us now understand the different types of indexes in PostgreSQL.

Btree index

As discussed earlier, by default BTREE index is created when no index type is mentioned in the **CREATE INDEX** statement. The b-tree index sorts the data and keeps it sequentially. It works well when the *equal-to* operator is used in the comparison in the query's **WHERE** clause. Along with *equal-to* BTREE works well with the below operators as well:

- <
- >
- <=
- >=
- IS NULL
- IS NOT NULL
- BETWEEN
- IN

B-tree indexes can also be used to retrieve data in sorted order. This is not always faster than a simple scan and sort, but it is often helpful.

Hash index

Hash Index generates a 32-bit hash code on the column where the index is created. It works best when the *equal* operator is used in the where clause of the query.

GiST and SP-GiST index

These indexes are used when two-dimensional geometrical data types are used.

Gin index

This index is used when one-dimensional data types like ARRAYS are used in the data types.

Brin index

BRIN indexes (a shorthand for **Block Range INdexes**) store summaries about the values stored in consecutive physical block ranges of a table. Thus, they are most effective for columns whose values are well-correlated with the physical order of the table rows. (*Reference: PostgreSQL Community Index Types*).

Indexes and expressions

An index column need not be just a column of the underlying table but can be a function or scalar expression computed from one or more columns of the table.

For example,

```
--WHERE clause with expression

SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John
Smith';

--INDEX creation with expression same as WHERE clause

CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

The system sees the query as just *WHERE indexed-column = 'constant'* and so the speed of the search is equivalent to any other simple index query. Thus, indexes on expressions are useful when retrieval speed is more important than insertion and update speed.

The metadata views - **pg_index** and **pg_indexes** give information about the details of the indexes like the schema name, index name, and the like.

We will see more examples of INDEX later in this chapter.

Statistics

Statistics are what we usually call metadata, meaning *the data about the data*. As we saw in the *Query Processing* section of *Chapter 6*, the planner is the stage in charge of determining the fastest and cheapest path to get the data required by a query. To accomplish that, it relies on the collected statistics.

There are two main types of statistics in PostgreSQL, each maintained by different components.

- The statistics about the server activity. These are collected and maintained by the stats collector background process; it feeds a set of catalog views with information about the usage of the system objects, such as the count of accesses to the tables and indexes or the number and state of the user sessions. Usually, this information is used by the Database Administrator to verify the system's state.
- The statistics about the data distribution within the tables and indexes. This information is collected when the **ANALYZE**, or **VACUUM ANALYZE** commands are executed manually or by the autovacuum background process. This is the information the planner uses, and we will focus on this section.

The statistical information about the tables and indexes data is stored in three different system catalogs, depending on their kind:

- **pg_class**: In this catalog, the system stores statistics about the number of tuples (rows) in the tables or indexes and the occupied blocks in the disk, alongside some objects' identity details.
- **pg_statistics**: Here postgres stores information about the *selectivity* of the data. It has one or two rows per table column, depending if the table has child tables (inheritance). This information might be hard to read since it is intended to be used by the system itself; however, postgres has a view called **pg_stats** which shows the same information in a more easy-to-understand way.
- **pg_statistics_ext_data**: Here additional information is stored in the case we add an *extended statistics object* for a specific set of tables columns.

In the following paragraphs, we will study each of these in detail and see a few examples.

To illustrate the different statistics types, we will use an example database called **pagila**, a port from the MySQL **sakila** example database. (Reference: *Devrim Gündüz pagila*).

Statistics in **pg_class**

Considering the tables and data from the example database **pagila** we can review the statistics in the **pg_class** catalog. For example, consulting the information for the address table and its index:

```
pagila=# SELECT
    relname AS "relation_name",
    relkind AS "relation_kind",
    reltuples AS "relation_tuples",
    relpages AS "relation_pages",
    pg_size.pretty(pg_relation_size(oid)) AS "relation_size"
FROM pg_class
WHERE relname LIKE 'address%'
AND relkind IN ('r', 'i');

-[ RECORD 1 ]-----
relation_name | address
relation_kind | r
relation_tuples | 603
relation_pages | 8
relation_size | 64 kB

-[ RECORD 2 ]-----
relation_name | address_pkey
relation_kind | i
```

```
relation_tuples | 603
relation_pages   | 4
relation_size    | 32 kB
```

This catalog shows that the table and the index have 603 rows. The table uses 8 pages, each of 8 KB occupying 64 KB in total in the disk. And, how we could guess the index is smaller and uses only 4 pages with a total of 32 KB. The planner uses this information to know how much work it would take to read from these objects.

The number of tuples is not updated in real time. As we saw before, the statistics are updated by the **ANALYZE** or **VACUUM ANALYZE** commands, but the planner uses the information about the used pages to escalate the estimation of the total number of rows, so the approximation is closer to the actual number.

Statistics in pg_statistics

The information in the **pg_class** is OK when the queries aim to read the complete table data, so the planner can know the total number of rows it will retrieve. However, the most common operations are executed to retrieve just a subset of rows from the table.

To improve these filtering operations, using the **WHERE** clause, postgres keeps statistics about the selectivity of the values contained per table column. This information is stored in the **pg_statistics** catalog, and as we saw above, there is a more human-readable view called **pg_stats**. Considering the same sample database, we could verify the information of the **postal_code** column from the **address** table:

```
pagila=# SELECT
  attname AS "column_name",
  n_distinct AS "distinct_rate",
  array_to_string(most_common_vals, E'\n') AS "most_common_values",
  array_to_string(most_common_freqs,   E'\n')   AS   "most_common_
  frequencies"
FROM pg_stats
WHERE tablename = 'address'
AND attname = 'postal_code';
```

```
- [ RECORD 1 ]-----+-----+
column_name          | postal_code
distinct_rate        | -0.9900498
most_common_values   |           +
                      | 22474      +
                      | 52137      +
                      | 9668
most_common_frequencies | 0.006633499 +
                         | 0.0033167496+
                         | 0.0033167496+
                         | 0.0033167496
```

The **pg_stats** view has some other details, but in this example, we can see the rate of distinct values, the most common values, and the frequencies of the most common values in the **postal_code** column. (Reference: PostgreSQL Community pg_stats)

Analyzing more details on this information:

- The **distinct_rate** shows the proportion of distinct values versus the total rows. In the case of a unique value column, such as the Primary Key, this rate is 100%, and it is represented with the value -1. The table column from the example is near -1, meaning almost all the values are distinct.
- The **most_common_values** gives insight into the **postal_code** values that repeat the most. In this case, four values: **null**, **22474**, **9668**, and **52137**.
- Finally, the **most_common_frequencies** shows the frequency of each of the most common values, the nulls are 0.66% of the total values, and each of the other values represents 0.33% of the total values from the sample.

We can verify the information the next way:

```
pagila=# WITH total AS (SELECT count(*)::numeric cnt FROM address)
SELECT
    postal_code, count(address.*), round(count(address.*)::numeric * 100 / total.cnt, 2) AS "percentage"
FROM address, total
```

```
GROUP BY address.postal_code, total.cnt  
HAVING count(address.*) > 1  
ORDER BY 2 DESC;  
  
postal_code | count | percentage  
-----+-----+  
        | 4 | 0.66  
22474    | 2 | 0.33  
52137    | 2 | 0.33  
9668     | 2 | 0.33  
(4 rows)
```

We can see only the **null** and the other three **postal_code** values appear more than once in the whole table, appearing four times or twice, so excepting these, all the other values are distinct. Also, by doing some math (**count x 100 / total_rows**), we get the percentage the number of appearances represents versus the total number of values, that is the frequency.

The planner uses these and the other details from the **pg_statistics** catalog to resolve queries using the **WHERE** clause, so it can define the best path to retrieve just a fraction of the rows when required.

Statistics in pg_statistics_ext_data

So far, we have seen PostgreSQL keeps general statistics about the total number of rows per table, the space they occupy on the disk, and information about the selectivity per table column. However, there are cases where the queries correlate multiple different columns, and the single-column orientation of the regular statistics could be improved.

For such cases, PostgreSQL has the ability to compute *extended statistics*. These are a particular type of statistics and are not gathered by default, so the user or database administrator has to define them. The goal is to define the correlation between different columns, so the planner can improve its calculation when resolving queries. The way the data from different columns is related depends on the design of the database model and the tables, which is why human intervention is required.

We have to use the **CREATE STATISTICS** command to define the *extended statistics*. *Figure 13.1* shows the options for this command:

```

pagila=# \h CREATE STATISTICS
Command:      CREATE STATISTICS
Description: define extended statistics
Syntax:
CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
    ON ( expression )
    FROM table_name

CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
    [ ( statistics_kind [, ...] ) ]
    ON { column_name | ( expression ) }, { column_name | ( expression ) } [, ...]
    FROM table_name

URL: https://www.postgresql.org/docs/14/sql-createstatistics.html

```

Figure 13.1: CREATE STATISTICS command options.

There are three types of extended statistics, each covering a distinct kind of column values correlation:

- Functional dependencies.
- Number of distinct values counts.
- Most common values list.

Functional dependencies

This kind of correlation is possible when the knowledge about one column is sufficient to determine another. This information lets the planner make accurate estimations about the total number of rows when the two columns are involved, for example, when using AND. Considering the same sample database as before, we can try this on the **city** table.

```

pagila=# CREATE STATISTICS extsts (dependencies) ON city_id, country_
id FROM city;

pagila=# ANALYZE city;

pagila=# SELECT
        stxname AS "ext_stats_name",
        stxkeys AS "ext_stats_columns",
        stxddependencies AS "stats_dependencies"
        FROM pg_statistic_ext JOIN pg_statistic_ext_data ON (oid = stxoid)

```

```
WHERE stxname = 'extsts';
```

```
- [ RECORD 1 ]-----+
ext_stats_name      | extsts
ext_stats_columns   | 1 3
stats_dependencies  | {"1 => 3": 1.000000, "3 => 1": 0.070000}
```

The **stats_dependencies** value shows that the column **city_id** (number 1) can 100% identify the **country_id** (number 3), whereas the other way, only 7%.

These estimations are beneficial, but they have some limitations. The planner assumes the dependencies on the columns are compatible and redundant, so it will do wrong estimations if they are incompatible. *Figure 13.2* shows how the planner will estimate 1 row even when the correlation is wrong (**NOTE: We will study EXPLAIN command later in this chapter**).

```
pagila=# EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM city WHERE country_id = 50 AND city_id = 172;
                                     QUERY PLAN
-----
Index Scan using city_pkey on city  (cost=0.28..8.29 rows=1 width=25) (actual rows=1 loops=1)
  Index Cond: (city_id = 172)
  Filter: (country_id = 50)
Planning Time: 0.152 ms
Execution Time: 0.102 ms
(5 rows)

pagila=# EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM city WHERE country_id = 60 AND city_id = 172;
                                     QUERY PLAN
-----
Index Scan using city_pkey on city  (cost=0.28..8.29 rows=1 width=25) (actual rows=0 loops=1)
  Index Cond: (city_id = 172)
  Filter: (country_id = 60)
  Rows Removed by Filter: 1
Planning Time: 0.386 ms
Execution Time: 0.139 ms
(6 rows)
```

Figure 13.2: Dependencies extended statistics, wrong estimation.

Number of distinct values counts

The standard statistics rely on statical data for individual columns, which could lead to wrong estimations about the number of distinct values from a multiple columns combination.

The Number of Distinct values count (or N-Distinct) extended statistics can help when the queries request distinct values from combining different columns, for example, when using the **GROUP BY** clause.

We can try this type of statistic on the **pagila** example database. Let's use the **address** table. Imagine a set of queries selecting distinct values from the **district**, **city_id**, and **postal_code** columns; we might define the n-distinct extended statistics as follows.

```
pagila=# CREATE STATISTICS extsts2 (ndistinct) ON district, city_id,
postal_code FROM address;
pagila=# ANALYZE address;
pagila=# SELECT
    stxname AS "ext_stats_name",
    stxkeys AS "ext_stats_columns",
    stxdndistinct AS "stats_ndistinct"
FROM pg_statistic_ext JOIN pg_statistic_ext_data ON (oid = stxoid)
WHERE stxname = 'extsts2';

-[ RECORD 1 ]-----+
-----
ext_stats_name | extsts2
ext_stats_columns | 4 5 6
stats_ndistinct | {"4, 5": 601, "4, 6": 601, "5, 6": 601, "4, 5, 6": 601}
```

Consulting the information stored in the **pg_statistic_ext_data** catalog, we can verify the number of distinct values through the different **district** (number 4), **city_id** (number 5), and **postal_code** (number 6) column combinations. All of them produce 601 different values with the current data. *Figure 13.3* illustrates the planer does a correct estimation.

```
pagila=# EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM address GROUP BY district, city_id, postal_code;
QUERY PLAN
-----
HashAggregate  (cost=20.06..26.07 rows=601 width=26) (actual rows=601 loops=1)
  Group Key: district, city_id, postal_code
  Batches: 1  Memory Usage: 169kB
    ->  Seq Scan on address  (cost=0.00..14.03 rows=603 width=18) (actual rows=603 loops=1)
Planning Time: 0.373 ms
Execution Time: 1.057 ms
(6 rows)
```

Figure 13.3: N-Distinct extended statistics.

Most common values list

As we saw at the beginning of this subsection, the standard statistics stored in the `pg_statistics` catalog contain information about the most common values per column and their frequency. This information is good when filtering data from a table based on a single column, but this can lead to lousy planning when more columns are involved.

The Most common values list (MVC List) extended statistics can improve the planner's decisions when working with queries doing filtering on multiple column conditions. The following shows how to create this statistic on the `address` table.

```
pagila=# CREATE STATISTICS extsts3 (mcv) ON district, postal_code FROM address;
pagila=# ANALYZE address;
pagila=# SELECT itms.*
  FROM pg_statistic_ext JOIN pg_statistic_ext_data ON (oid = stxoid),
  pg_mcv_list_items(stxdmcv) itms
 WHERE stxname = 'extsts3';

-[ RECORD 1 ]-----
index      | 0
values     | {QLD,""}
nulls      | {f,f}
frequency   | 0.003316749585406302
base_frequency | 2.200165562458575e-05

-[ RECORD 2 ]-----
index      | 1
values     | {Alberta,""}
nulls      | {f,f}
```

```

frequency      | 0.003316749585406302
base_frequency | 2.200165562458575e-05
-[ RECORD 3 ]-----
index          | 2
values          | {"",65952}
nulls           | {f,f}
frequency       | 0.001658374792703151
base_frequency | 8.250620859219656e-06
-[ RECORD 4 ]-----
index          | 3
values          | {"Abu Dhabi",41136}
nulls           | {f,f}
frequency       | 0.001658374792703151
base_frequency | 5.500413906146438e-06
...

```

Consulting the information stored for this extended statistics object, we can see the combination of **values**, their **frequency** as combined, and the **base_frequency**, which is the result computed as per-column frequency. For example, record number 4 shows the combination of **district** and **postal_code** {"Abu Dhabi",41136} has a frequency of 0.16% and if the are computed per column, the frequency is only 0.0005%, which is a remarkable difference.

Explain plan

Before understanding explain plan, let's review what the **ANALYZE** command is. As we saw in the previous section, the **ANALYZE** command gathers statistics about the contents of tables in the database and stores this metadata in the **pg_class**, **pg_statistic**, or **pg_statistics_ext_data** system catalogs. Eventually, the query planner uses this metadata which helps in determining the effective execution plans for queries.

EXPLAIN PLAN as the name suggests gives details about the queries' execution plan. It gives appropriate results when the statistics are updated. That is why it is recommended use **EXPLAIN PLAN** with **ANALYZE** command so that all metadata is up-to-date when the query plan is created.

Syntax as per PostgreSQL Documentation (*Reference: PostgreSQL Community Explain Plan*).

```
EXPLAIN [ ( option [, ...] ) ] statement  
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

where option can be one of:

```
ANALYZE [ boolean ]  
VERBOSE [ boolean ]  
COSTS [ boolean ]  
SETTINGS [ boolean ]  
BUFFERS [ boolean ]  
WAL [ boolean ]  
TIMING [ boolean ]  
SUMMARY [ boolean ]  
FORMAT { TEXT | XML | JSON | YAML }
```

The explain plan gives the details about whether it will use index scan or sequential scan, what plans are used to fetch the data if multiple tables are used, how many rows will come in the output of the query and many such details which can help in understanding the time taken by each step. This will help to tune to query in case there are performance issues.

Let's see the same example of one Department (**dept**) having multiple Employees in the **emp** table we used in Chapter 11 and Chapter 12. *Figure 13.4* shows the example of the join query for the **emp** and **dept** tables to fetch details of employees in **dept_id**

=2.

```
postgres=# EXPLAIN (ANALYZE,BUFFERS)
postgres-# SELECT *
postgres-# FROM EMP E,
postgres-#      DEPT D
postgres-# WHERE E.DEPT_ID = D.DEPT_ID
postgres-# AND E.DEPT_ID = 2;
                                         QUERY PLAN
-----
Nested Loop  (cost=10000000000.13..10000000000.22 rows=1 width=668) (actual time=0.068..0.077 rows=2 loops=1)
  Buffers: shared hit=5
    -> Seq Scan on emp e  (cost=10000000000.00..10000000001.06 rows=1 width=300) (actual time=0.034..0.037 rows=2 loops=1)
        Filter: (dept_id = '2'::numeric)
        Rows Removed by Filter: 3
        Buffers: shared hit=1
    -> Index Scan using dept_pk1 on dept d  (cost=0.13..8.15 rows=1 width=368) (actual time=0.017..0.017 rows=1 loops=2)
        Index Cond: (dept_id = '2'::numeric)
        Buffers: shared hit=4
Planning Time: 0.371 ms
Execution Time: 0.149 ms
(11 rows)
```

Figure 13.4: Example of EXPLAIN PLAN in a join query with ANALYZE and BUFFERS

Notice that Sequential Scan is being used for the **WHERE** clause condition **e.dept_id = 2**. There is a scope of tuning here where we can create an index on the Foreign Key column **dept_id** of the **emp** table. Let's create an index on this column and check the explain plan again.

```
CREATE INDEX INDX_EMP_DEPT_ID ON EMP(DEPT_ID);
```

Figure 13.5 shows the simple Explain Plan with no attributes. Be aware of the difference in the output of the explain plan in *Figures 13.4* and *13.5*.

Notice that Index Scan is used this time with the newly created index instead of Sequential Scan. This makes the performance of the query faster. We might not be able to see the difference when there are fewer records in the query output, but the increase in performance can be observed when the data increases.

```
postgres=# EXPLAIN
postgres-# SELECT *
postgres-# FROM EMP E,
postgres-#      DEPT D
postgres-# WHERE E.DEPT_ID = D.DEPT_ID
postgres-# AND E.DEPT_ID = 2;
                                         QUERY PLAN
-----
Nested Loop  (cost=0.26..16.31 rows=1 width=668)
  -> Index Scan using idx_emp_dept_id on emp e  (cost=0.13..8.15 rows=1 width=300)
      Index Cond: (dept_id = '2'::numeric)
  -> Index Scan using dept_pk1 on dept d  (cost=0.13..8.15 rows=1 width=368)
      Index Cond: (dept_id = '2'::numeric)
(5 rows)
```

Figure 13.5: Example of EXPLAIN PLAN in a join query without any attributes

Below gives details of the useful parameters which should be kept in mind while tuning the queries and analyzing them using the explain plan. There are more configurations from the ones mentioned below, which could be checked using the **SHOW** command or in the **postgresql.conf** file, as we will see in the next section.

- **enable_indexscan (boolean)**
- **enable_nestloop (boolean)**
- **enable_seqscan (boolean)**
- **enable_sort (boolean)**

These are self-explanatory from the name of the parameter. By default, most of such parameters are on, and one can disable them depending upon the situation.

Best practices for the **postgresql.conf** parameters

When configuring a new PostgreSQL cluster or working with an existing one to improve performance, we can consider some standard parameter configurations widely advised as best practices for almost any system. These parameters are defined in the **postgresql.conf** file, generally located in the **\$PGDATA** directory.

We need to remember that these best practices or baselines are generally effective and produce the desired results; however, there is always room for improvements, especially if the database model design or user workloads have some specifics. So, we can start with the best practices and develop a regular habit of monitoring our systems and tuning what is required over time.

The following is a list of the main configuration parameters we can tune based on best practices to bring good performance.

- **shared_buffers**
- **work_mem**
- **autovacuum**
- **effective_cache_size**
- **maintenance_work_mem**
- **max_connections**

In the previous chapters, we have seen a few of the above; now, we will review them and learn about their suggested initial values.

shared_buffers

This parameter defines the size of the memory area named the same, which we also might know as the “database cache.” This memory area will keep the most accessed rows so new reads can retrieve them from here rather than going to disk; also, all the data changes are written in this area instead of immediately on disk.

The default value of this parameter is very conservative, just 128 MB. The best practice for this parameter is to set it between 15% and 25% of the total RAM. So, for example, in a 64 GB RAM system, the **shared_buffers** should be set to 16 GB. Changing this parameter demands a server restart.

work_mem

This parameter determines the size of the memory area named the same. Differently from the **shared_buffers**, which exists just one for the whole database cluster (shared by all the existing databases), a **work_mem** is allocated per user session. This area is used for all the sort operations, such as **ORDER BY**, **DISTINCT**, and **MERGE JOINS**.

If the operation using the **work_mem** requires more space to complete, then PostgreSQL will use temporary files written on disk. Tuning this parameter to avoid the IO disk operations can improve performance.

The default value of this parameter is just 4 MB. We must consider the available resources and the number of concurrent user sessions to adjust it.

To be on the safe side, we can pick the **max_sessions** value, which also should be sized right. So, the calculation for this parameter can be expressed the following way: **work_mem = 25% of total RAM / max_connections**.

We can change this parameter without a server restart and even set it at the role (user) level using the **ALTER USER** command. (*Reference: PostgreSQL Community alter user*).

autovacuum

This parameter controls if the autovacuum background process is enabled or not. As we have seen in previous chapters, vacuuming the tables and indexes is vital in a PostgreSQL system. By default, this parameter is enabled, so the recommendation is to keep it this way.

If, for any reason, you need to disable the autovacuum, the recommendation is to disable it by table rather than for the whole cluster, changing it at the cluster level

in the `postgresql.conf` file requires a server restart. You can use the `ALTER TABLE` command to disable the autovacuum at the table level. (*Reference: PostgreSQL Community alter table*).

effective_cache_size

This parameter lets the query planner know how much memory is expected to be available in the system for disk caching within the database. It does not represent an allocated memory area, but the planner uses its value to decide whether the operations to resolve specific queries will fit the RAM.

If the value is too low, the planner might disregard the index scans and prefer sequential table scans, which usually perform slower to access specific data subsets.

The default for this parameter is 4 GB, and the best practice is setting it between 50% and 75% of the total RAM. Change it doesn't require a server restart.

maintenance_work_mem

This parameter defines the size for the memory area named the same, which is used for the maintenance tasks `VACUUM`, `CREATE INDEX`, and `ALTER TABLE ADD FOREIGN KEY`. Since only one of these tasks can be running simultaneously per connection and are usually executed in a lower proportion than the `work_mem` operations, its value can be significantly larger than `work_mem`.

Its default value is 64 MB, which is very conservative. The advisable value for this parameter is between 5% and 10% of the total RAM.

max_connections

This parameter limits the maximum number of simultaneous user database connections. As we saw before, each of these connections can hold a `work_mem` size memory area at least.

Remember, PostgreSQL has a process-based architecture, so each user session represents a new process at the database server. There is some relation between the number of CPUs available in the system and the maximum number of processes it can handle efficiently.

The default of this parameter is 100, and usually, it is recommended to set this value lower than 10 per CPU and consider 20 per CPU as the limit. If you need to allow more user sessions than these, consider a pooler such as `pgbouncer`.

Summary

The above parameters are the main ones we advise you to verify when working to improve the performance of your system. Next is a summary table with the default and suggested values.

Parameter	Default values	Best practice value
<code>shared_buffers</code>	128 MB	Between 15 - 25% of total RAM
<code>work_mem</code>	4 MB	25% of total RAM / max_connexions
<code>autovacuum</code>	ON	ON
<code>effective_cache_size</code>	4 GB	Between 50% - 75% of total RAM
<code>maintenance_work_mem</code>	64 MB	Between 5% - 10% of total RAM
<code>max_connections</code>	100	Up to 20 per CPU

Conclusion

In this chapter, we have seen the concepts which can help to increase the performance of the database and helps in tuning the queries. The topics like Index creation and its maintenance, statistics, and explain plan are important from the DBA perspective. We also discussed the best practices to be considered in one of the most important `postgresql.conf` file.

In the next chapter, we will discuss how-to, tips, and tricks to troubleshoot the database.

Bibliography

- Devrim Gündüz pagila: <https://github.com/devrimgunduz/pagila>
- PostgreSQL Community pg_stats: <https://www.postgresql.org/docs/14/view-pg-stats.html>
- PostgreSQL Community Index: <https://www.postgresql.org/docs/current/sql-createindex.html>
- PostgreSQL Community Re Index: <https://www.postgresql.org/docs/current/sql-reindex.html>
- PostgreSQL Community Index Types: <https://www.postgresql.org/docs/current/indexes-types.html>

- PostgreSQL Community Explain Plan: <https://www.postgresql.org/docs/14/sql-explain.html>
- PostgreSQL Community alter user: <https://www.postgresql.org/docs/14/sql-alteruser.html>
- PostgreSQL Community alter table: <https://www.postgresql.org/docs/14/sql-altertable.html>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 14

Troubleshooting

Introduction

All technological systems are susceptible to unexpected errors and failures; even in the most stable workloads, the possibility of hitting a hidden bug or an unknown situation is present. While working with PostgreSQL, we can face this kind of situation, and it is essential to know the tools and methods we can use to troubleshoot and debug the errors to avoid or fix them.

In previous chapters, we reviewed the features of PostgreSQL and how to configure them and tune our systems. In this chapter, we will learn about the principal methods and tools we can use when the time to debug an error comes in. We will study these concepts from the PostgreSQL and Operating System points of view.

Structure

This chapter includes the following sections.

- Debugging using log files.
- Debugging using PostgreSQL tools and commands.
- Debugging using Operating System tools and commands.

Objectives

Completing this chapter, you will gain knowledge of the main techniques and tools to troubleshoot an unexpected error. Also, you will be familiar with the relationship between the Operating System and the PostgreSQL processes.

Debugging using log files

Like any other system, PostgreSQL can log helpful information in its log files. The logged data can help understand the sequence of events and their relationship. Also, some information is not visible from the DB but from the log files.

What is logged and where is logged depend on a set of configuration parameters in the **postgresql.conf** file. Depending on the system workload and the type of operations, we might need some details to be logged, and others do not. However, we can consider some standard and usually recommended logging parameters. The following subsections describe the main parameters we should set for logging valuable information.

Where to log

It is important to define where the log files will be stored; this way, we will ensure their availability when needed and control how to recycle them to avoid running out of space. Also, under high workloads, avoid causing an impact on the disk for the database operations by moving the log files to a different disk. The following are the principal parameter we can set to define these aspects for the log files.

logging_collector

When enabling this parameter, a background process is started, and it captures all the messages sent to the standard error (**stderr**) and redirects them into log files. The default value for this parameter is **off**; however, it is highly recommended to set it to **on**.

Even when it is possible to log the PostgreSQL messages without enabling the **logging_collector** parameter, the only way to ensure all the messages will be captured, and there will not be any loss is by enabling this parameter. Also, this process adds support for the rotation capabilities.

log_destination

PostgreSQL supports a variety of destinations for the log messages, which adds flexibility to decide what is the best depending on the infrastructure and solution design. The possible values for this parameter are: **stderr**, **csvlog**, and **syslog**. When running PostgreSQL in Windows, it is possible to set this parameter to **eventlog**.

The **stderr** is the default value, and it will generate the output in a text format. The **csvlog** will format the output to be separated by commas, which is especially useful when we use the log messages to feed other different applications or analyzer systems. To use the **syslog** (Linux/Unix-alike systems) or the **eventlog** options, it is necessary to configure the operating system accordingly so the messages are processed as expected. Configuring these operating system utilities is beyond the scope of this book, so reviewing them, is worth it.

log_directory

As you might already deduce, this parameter controls the directory path where the log files will be created and stored. This parameter only becomes valid when the **logging_collector** is enabled.

You can set the value for this parameter as an absolute path or relative to the cluster data directory, a.k.a **\$PGDATA**. It is default set to **log**, which means the log files will be created in the **\$PGDATA/log** directory. It is highly recommended, especially if your system handles a high workload, to set this directory in separate storage from the one where the database data is. *Figure 14.1* shows a system with this parameter configured in a different storage from the database data.

```
postgres=# show data_directory;
          data_directory
-----
/var/lib/postgresql/14/main
(1 row)

postgres=# show log_directory;
          log_directory
-----
/pg/logs
(1 row)
```

Figure 14.1: log_directory in a separate path from the database data

log_filename

This parameter can be used when the **logging_collector** is enabled, and it sets the file name convention to be used when creating the log files. This parameter supports **strftime** patterns to produce time-varying filenames. (Reference: *strftime*).

The default value is **postgresql-%Y-%m-%d_%H%M%S.log**, so considering the file creation date as **Feb 15, 2023, 17:22:45**, the filename will be **postgresql-2023-02-15_172245.log**. We can define the log file names in a way it we do not need an external scheduled job to remove the old files; for example, if we set the file names to be set to the day of the week or month, and have the **log_truncate_on_rotation** enabled, every time the *same* logfile gets created a new one will take the place of the previous overwriting it. This will consistently keep the same number of files in place.

log_rotation_age

This parameter controls when to rotate the log file to a new one based on time; this is possible only when the **logging_collector** is enabled. The value is expressed in minutes; the default value is 24 hours (1440 minutes), so a new log file is added at the beginning of a new day. It is possible to set this parameter to 0 (zero) to deactivate the time-based rotation mechanism.

log_rotation_size

Similarly to the last parameter, this one defines when to create a new log file based on the file size when the **logging_collector** is enabled. If the value has no units specified, it is calculated as kilobytes. The default value is 10MB, and setting it to 0 (zero) disables the size-based rotation.

log_truncate_on_rotation

When the **logging_collector** is enabled, this parameter controls if the messages sent to an existing log file will be appended (**off**) or the file will be truncated / overwritten (**on**). As we saw before, this can be combined with the **log_filename** to control the number of files to store.

For example, if the **log_filename** is configured as **postgresql-%H.log**, the **log_rotation_age** is on its default of **1440** minutes (1 day), and the **log_truncate_on_rotation** is **on**, then every hour, a new empty file will be created, and PostgreSQL will keep only 24 files.

What to log

Setting the PostgreSQL logger correctly will provide valuable information we can use when we need to debug system operations and events. The following are the main options we should consider configuring, but they depend on the type of workload, so we need to evaluate them to avoid extra logging that might not add value.

log_line_prefix

This is one of the most useful configuration options; whenever we aim to analyze the database logs manually or use a parser such as **pgbadger**, we should consider configuring this parameter with all the relevant information. This parameter defines a **printf-style** string that will be added at the beginning of each log line. The default value is `%m [%p]`. (Reference: PostgreSQL Community **Log_line_prefix**)

Among others, the information we can include by setting this parameter might contain:

- The specific timestamp (`%t`).
- The process ID (`%p`).
- The PostgreSQL database (`%d`).
- The client database user (`%u`), application name (`%a`), and/or source host (`%h`).

These details will help to identify the process, user, and source for a specific log event, making the analysis and troubleshooting easier. Let us say you have the following configuration:

```
postgres=# SHOW log_line_prefix;
          log_line_prefix
-----
%t [%p]: db=%d user=%u,app=%a,client=%h
(1 row)
```

And there is some error when calling an unexisting function:

```
postgres=# SELECT * FROM new();
ERROR:  function new() does not exist
```

You will see the following message as illustrated in *Figure 14.2*:

2023-02-20 18:23:40 CST [19059]: db=postgres user=postgres,app=psql,client=[local]	ERROR: function newO does not exist at character 15
2023-02-20 18:23:40 CST [19059]: db=postgres user=postgres,app=psql,client=[local]	HINT: No function matches the given name and argument types.
2023-02-20 18:23:40 CST [19059]: db=postgres user=postgres,app=psql,client=[local]	STATEMENT: SELECT * FROM newO ;

↑ ↑ ↑ ↑ ↑ ↑ ↑

Time PID Database User App Host Log message

Figure 14.2: Example of output as per log_line_prefix setting

log_connections/log_disconnections

As the name suggests, these parameters will add a log event for every user connection and/or disconnection, depending on how we configure them. The default value is **off** for both of them.

log_min_duration_statement

This parameter controls when the completion of the queries' statements will be logged; a new log message will be added when the execution of a given statement is longer than the time defined in this parameter. The used unit is milliseconds, and the default value is **-1**, meaning this option is disabled. A 0 (zero) value will cause all the statements to be logged.

This is one of the most useful logging options.

log_lock_waits

When enabled, a log message will be added if a session waits longer than the defined time in the **deadlock_timeout** parameter (default 1 second) to acquire a lock. The default value is **off**. The log messages produced by this setting help to identify if the locking waits are causing issues.

log_autovacuum_min_duration

This parameter defines to log of any autovacuum events running at least this amount of time. The units are milliseconds, and the default value is **-1**, which disables this option. Setting a 0 (zero) value causes all the autovacuum operations to be logged. By defining a meaningful amount of time, we can get insight if the autovacuum is causing performance or blocking issues.

A well-configured logger in PostgreSQL will produce a great tool for debugging errors and failure situations. Even when it is possible to read and gather details from the postgres logs manually, it is highly recommended to parse and analyze the log files with some other tool, for example, **pgbadger**, which we saw in *Chapter 10*.

Parameters summary

As we saw, multiple configuration parameters can help to log relevant information we can use in case of troubleshooting. *Table 14.1* summarize the reviewed parameter and the suggested configuration.

Parameter	Default value	Recommended value	Units
<code>logging_collector</code>	<code>off</code>	<code>on</code>	
<code>log_destination</code>	<code>stderr</code>	<code>stderr, csvlog</code>	
<code>log_directory</code>	<code>log</code>	A different storage path from the DB data.	
<code>log_filename</code>	<code>postgresql-%Y-%m-%d_%H%M%S.log</code>	<code>postgresql-%a.log</code>	
<code>log_rotation_age</code>	<code>1440</code>	<code>1440</code>	minutes
<code>log_rotation_size</code>	‘10MB’	<code>0</code>	KB, MB, GB
<code>log_truncate_on_rotation</code>	<code>off</code>	<code>on</code>	
<code>log_line_prefix</code>	<code>%m [%p]</code>	<code>%t [%p]: db=%d user=%u, app=%a, client=%h</code>	
<code>log_connections</code>	<code>off</code>	<code>on</code>	
<code>log_disconnections</code>	<code>off</code>	<code>on</code>	
<code>log_min_duration_statement</code>	<code>-1</code>	<code>500</code>	milliseconds
<code>log_lock_waits</code>	<code>off</code>	<code>on</code>	
<code>log_autovacuum_min_duration</code>	<code>-1</code>	<code>1000</code>	milliseconds

Table 14.1: Main configuration parameters for PostgreSQL logging.

Debugging using PostgreSQL tools and commands

As we have studied before in this book, PostgreSQL is composed of a set of background processes responsible for different tasks, and there are a few native commands we can use to instruct them. Also, the system maintains a variety of catalogs that contain

information we can use when we need to troubleshoot. So, while accessing postgres with the right privileges, we can execute the following commands to gather details to solve some issues.

In the following paragraphs, we will study the main and basic commands or queries we can use to gather details about *what is happening* in the database system. Then a few other commands to instruct the system to execute certain actions.

Gather information

Before trying to fix something, we must know what is happening, so when troubleshooting PostgreSQL, we can use the following to get insights.

Check the PostgreSQL version

It might look basic, but when we start troubleshooting an issue, the initial step we can do is verify the specific version the system is running. With this information, we can identify if the version is affected by an identified bug and if the case, if there is a fix already, if the version supports some feature that helps to solve the issue, and if it is still under support.

```
postgres=# SELECT version();
```

Check database and objects size

There are situations where the issue we are facing is related to the disk running out of free space; in such cases, knowing the size of the database and the tables and indexes comes in handy.

We can get the database size in a human-readable way with the following:

```
postgres=# SELECT pg_size_pretty(pg_database_size('database_name'));
```

Then, to be more specific, we can verify the total size a table is consuming in the disk, including the data and the indexes:

```
postgres=# SELECT  
pg_size_pretty(pg_total_relation_size('table_name'));
```

If we want to know the size of the table data and the indexes separately, we can use the following:

```
postgres=# SELECT
```

```
pg_size.pretty(pg_table_size('table_name')) AS "Table Size",
pg_size.pretty(pg_indexes_size('table_name')) AS "Indexes Size";
```

Check database connections

We can also know the number of database connections and their state, giving us insight into what the database servers.

```
postgres=# SELECT COUNT(*) AS "# of Sessions", state AS "State" FROM
pg_stat_activity GROUP BY state;
```

If required, we can add some extra details, for example, the following. We can know the database name and database user alongside the last details:

```
postgres=# SELECT
datname AS "Database",
username AS "User",
state AS "State",
COUNT(*) AS "# of Sessions"
FROM pg_stat_activity
GROUP BY state, datname, username
ORDER BY "Database", "User", "# of Sessions";
```

Finally, we might want to know for how long these sessions are being connected and for how long their queries have been running:

```
postgres=# SELECT
datname AS "Database",
username AS "User",
state AS "State",
now() - backend_start AS "Session Time",
now() - query_start AS "Query Time"
FROM pg_stat_activity
ORDER BY "Database", "User", "Session Time";
```

Figure 14.3 illustrates the output from the last command:

```
postgres=# SELECT
    datname AS "Database",
    username AS "User",
    state AS "State",
    now() - backend_start AS "Session Time",
    now() - query_start AS "Query Time"
FROM pg_stat_activity
ORDER BY "Database", "User", "Session Time";
  Database |   User   | State | Session Time      | Query Time
-----+-----+-----+-----+
  postgres | postgres | active | 00:00:16.536177 | 00:00:00
    sumo   | postgres | idle   | 00:00:46.058131 |
```

Figure 14.3: Checking user session details

Check slow queries

The following queries are handy for quickly identifying queries that might have been bad-behaving and causing issues. To use the following, we need to have added the **pg_statements** extension, which we learned in *Chapter 10: Most used Extentions/Tools*; you can review it for a few extra queries.

The following will return the queries that have taken longer than 5 seconds to complete and their execution time:

```
postgres=# SELECT query, total_time
  FROM pg_stat_statements
 WHERE total_time > 5000 ORDER BY total_time DESC;
```

Check statistics

In the previous chapter, we studied the statistics the query planner uses to decide the quickest path to get the data, but also we mentioned some other kinds of statistics PostgreSQL collects regarding the usage and state of the database and its objects. These statistics can help us understand how the system behaves and identify issues.

The statistics are stored in multiple **pg_stat*** views; the following are some of the most relevant.

We can verify the last time the tables were vacuumed and analyzed. Having tables with long periods without vacuum or analysis on them can point to a potential performance issue and, in some cases, affect the whole system.

```
postgres=# SELECT
    schemaname AS "Schema", relname AS "Table",
```

```
last_vacuum, last_autovacuum, last_analyze, last_autoanalyze
FROM pg_stat_user_tables ORDER BY last_vacuum DESC;
```

We can use the following query to get an insight into how the tables are accessed. If the number of sequential scans exceeds the index scans, the system might face some latency and slowness.

```
postgres=# SELECT
    schemaname AS "Schema", relname AS "Table",
    seq_scan AS "# Seq Scan", idx_scan AS "# Index Scan"
FROM pg_stat_user_tables ORDER BY "# Seq Scan" DESC;
```

When our system includes a replication setup, verifying if the replicas are synchronized and not getting behind by too far from the changes in the primary node is valuable. We can use the following in the primary to know the status of the replicas.

```
postgres=# SELECT
    application_name AS "Client App Name", state AS "State",
    pg_current_wal_lsn() AS "Current WAL Position",
    replay_lsn AS "Replayed WAL Position",
    replay_lag AS "Replay Lag",
    pg_size.pretty(pg_wal_lsn_diff(pg_current_wal_lsn(),replay_lsn)) AS
    "Lag Size"
FROM pg_stat_replication ORDER BY "Lag Size" DESC;
```

Figure 14.4 illustrates a replication setup with two replicas in good shape:

Client App Name	State	Current WAL Position	Replayed WAL Position	Replay Lag	Lag Size
replica_2	streaming	D077/17215018	D077/17215018	00:00:00.047488	0 bytes
replica_1	streaming	D077/17215018	D077/17215018	00:00:00.009076	0 bytes

(2 rows)

Figure 14.4: Checking replication health with pg_stat_replication

Instruct PostgreSQL

In some previous chapters we have studied some special commands to execute specific actions in PostgreSQL, apart from querying the tables data. When troubleshooting an issue, sometimes we need to perform corrective actions, the following commands can help.

Vacuuming and analyzing

As you can relate already, there are special commands to execute the VACUUM and / or ANALYZE operations on demand. They can help if we have identified the cause of an issue as the absence of the vacuum cleaning or the freshness of the statistics is outdated. The commands to execute such tasks are kind of obvious:

```
postgres=# VACUUM [tablename];
```

```
postgres=# ANALYZE [tablename];
```

In both cases, we can include the name of a specific table to execute the corresponding routine just in that object; otherwise, the execution will affect the whole database. Review *Chapter 6: PostgreSQL Internals* to refresh some details and options about VACUUM.

Terminate queries or user sessions

Under certain circumstances, you might need to terminate one or more user sessions connected to the database. This can be because of an undesired execution or because the identified sessions are running bad queries and affecting all the other users by slowing down the system or blocking access to the tables.

Two options to do this:

```
-- Terminate a query but keep the connection alive.
```

```
postgres=# SELECT pg_cancel_backend(pid);
```

```
-- Terminate a query and kill the connection.
```

```
postgres=# SELECT pg_terminate_backend(pid);
```

The **pid** is the process identifier of the user session; this detail can be found in the **pg_stat_activity** catalog we saw before. We can even terminate *all* the sessions for a specific database user if we require it:

```
postgres=# SELECT pg_terminate_backend(pid)
  FROM pg_stat_activity
 WHERE username = 'database_username';
```

Manage replication

There are some special cases when we need to manage the state of the existing replication setup, for example pausing the replica, resuming it and verifying the role of a given server. The following commands come in handy.

To verify the role of a server in the replication setup, we can check if the server is *in recovery* mode, if **true** means it is a replica, **false** it is the primary server.

```
postgres=# SELECT pg_is_in_recovery()::text;
```

We can also pause the replication and then resume it again when required. Be aware that pausing an ongoing replica will cause the primary to retain WAL files, which can cause storage issues if the disk space is limited.

-- Check the replication status.

```
postgres=# SELECT pg_get_wal_replay_pause_state();
```

-- Pause an ongoing replication.

```
postgres=# SELECT pg_wal_replay_pause();
```

-- Resume a paused replication.

```
postgres=# SELECT pg_wal_replay_resume();
```

Under certain circumstances, you might need to change the role of one of the replica servers to turn it into a new primary, for example, if the current primary server becomes unavailable. In this situation, you need to handle how the database users are connecting and how to manage the former primary server rejoin. For now, these details are out of the scope of this book, but it is advisable to learn about them.

-- While connected to a replica server.

```
postgres=# SELECT pg_promote();
```

You can refer to *Chapter 8, Replicating Data*, to review some extra information about replication in PostgreSQL.

Debugging using Operating System tools and commands

When facing issues with our PostgreSQL system, we can collect relevant information from the Operating System itself. Ultimately, all the processes composing the PostgreSQL service and the user sessions run at the Operating System level.

Understanding the Operating Systems layers, components, and services is a powerful tool for any Database Administrator. In this book, we are not going in deep into all the OS concepts; however, we can study some of the basic and main tools we can use to troubleshoot our systems. In this chapter, we are considering Linux-alike Operating Systems, so we are not including tools and examples for Windows platforms.

We have a large set of tools and commands from the Operating System when we need to troubleshoot, some can help to get a global view of the running services and system-wide resources, and others will give insight about specific processes, and we also have a couple of options to get log entries or system events so that we can review them over the time.

Service and system-wide tools

When we troubleshoot an issue, we can start by checking the general status of the service and trying to identify if the system is showing some unexpected behavior from the resource consumption point of view.

systemctl

When we install PostgreSQL on different Linux flavors, we will get a service unit, and the recommendation is to handle the postgres service through it. We can verify the status of a service, stop, start, or restart it if required.

```
-- To verify the status of the PostgreSQL service
```

```
root@:~# systemctl status postgresql
```

```
-- To stop the PostgreSQL service
```

```
root@:~# systemctl stop postgresql
```

```
-- To start the PostgreSQL service
```

```
root@ubuntu-focal:~# systemctl start postgresql
-- To restart the PostgreSQL service
root@ubuntu-focal:~# systemctl restart postgresql
```

Figure 14.5 illustrates the output of the `systemctl` command when checking the status of the PostgreSQL service, note, in this case, the service has been active since four days. Also, the last lines show the latest messages from the `postgres` log file, so we can quickly see if something is happening:

```
root@ubuntu-focal:~# systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
  Loaded: loaded (/lib/systemd/system/postgresql.service; enabled; vendor preset: enabled)
  Active: active (exited) since Sun 2023-02-19 23:23:45 UTC; 4 days ago
    Main PID: 984 (code=exited, status=0/SUCCESS)
      Tasks: 0 (limit: 1131)
     Memory: 0B
       CGroup: /system.slice/postgresql.service

Feb 19 23:23:45 ubuntu-focal systemd[1]: Starting PostgreSQL RDBMS...
Feb 19 23:23:45 ubuntu-focal systemd[1]: Finished PostgreSQL RDBMS.
```

Figure 14.5: Checking the PostgreSQL service status with `systemctl`.

free

The `free` command will show information about the system memory; we get details about the total memory, what is used and free, and if the swap (an auxiliary memory space on disk) is being used. (*Reference: Linux Manual free*)

The RAM is a limited resource and one of the most critical for a database system, so running out of free memory will have an impact. *Figure 14.6* shows an output from the `free` command.

```
-- Getting memory details in human-readable format
```

```
root@ubuntu-focal:~# free -h
```

	total	used	free	shared	buff/cache	available
Mem:	976Mi	174Mi	233Mi	14Mi	568Mi	629Mi
Swap:	0B	0B	0B			

Figure 14.6: Output from the `free` command in a human-readable format.

df

The **df** command is a tool to get information about the space on the mounted filesystems. We can get an insight into whether any filesystems are running out of free space. *Figure 14.7* illustrates the output from the command **df**. (Reference: *Linux Manual df*)

```
-- Getting filesystem space usage
```

```
root@:~# df -Ph
```

Filesystem	Size	Used	Avail	Use%	Mounted on
udev	472M	0	472M	0%	/dev
tmpfs	98M	1.1M	97M	2%	/run
/dev/sda1	39G	4.1G	35G	11%	/
tmpfs	489M	28K	489M	1%	/dev/shm
tmpfs	5.0M	0	5.0M	0%	/run/lock
tmpfs	489M	0	489M	0%	/sys/fs/cgroup
/dev/loop0	64M	64M	0	100%	/snap/core20/1778
/dev/loop2	68M	68M	0	100%	/snap/lxd/22753
/dev/loop3	50M	50M	0	100%	/snap/snapd/17883
/dev/loop4	92M	92M	0	100%	/snap/lxd/24061
vagrant	466G	383G	84G	83%	/vagrant
/dev/loop5	50M	50M	0	100%	/snap/snapd/18357
/dev/loop6	64M	64M	0	100%	/snap/core20/1822
tmpfs	98M	0	98M	0%	/run/user/1000

Figure 14.7: Output from the df command

Processes-oriented tools

Additional to the previous tools, we will review a couple oriented to processes. With these, we can review information about specific processes running in the system.

top/htop

When looking at a system facing issues, we can use the **top** tool to get information about the different processes running, and we can sort them per the most demanding processes, which are using more CPU and memory or causing the highest load. The **htop** variant includes a more visual interface and supports some actions through clicks. *Figure 14.8* illustrates a **htop** output:

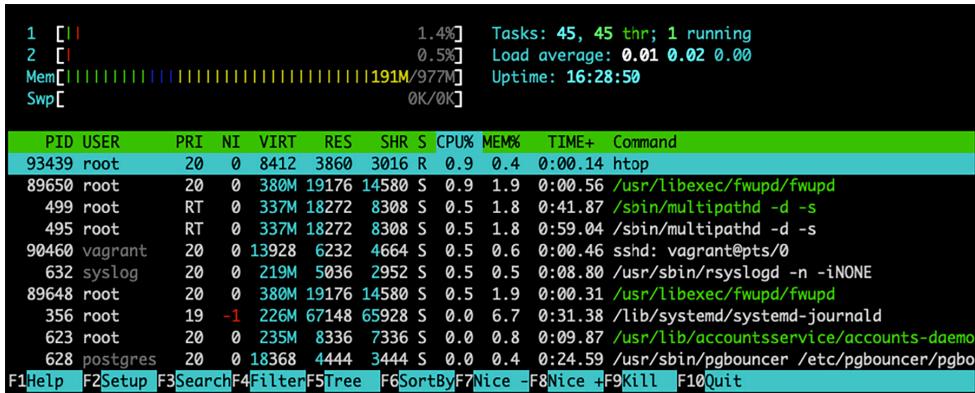


Figure 14.8: An htop command output

ps/pgrep

The **ps** and **pgrep** commands help to list and find a certain running process. We can verify if the expected processes are under execution or if any other that should not be running actually exists. For example, we can verify if all the PostgreSQL background processes are actually running.

```
-- Getting a list of the postgres processes with ps
root@:~# ps -fea | grep postgres
```

```
-- Getting the same list pgrep
root@:~# pgrep -a postgres
```

Figure 14.9 illustrates how to get a list of the postgres related processes using both commands:

```
root@ubuntu-focal:~# ps -fea | grep postgres
postgres      628          1  0 Feb23 ?        00:00:25 /usr/sbin/pgbouncer /etc/pgbouncer/pgbouncer.ini
postgres     90694          1  0 00:06 ?        00:00:00 /usr/lib/postgresql/14/bin/postgres -D /var/lib/postgresql/14/main -c config_file=/etc/postgresql/14/main/postgresql.conf
root@ubuntu-focal:~# pgrep -a postgres
90694 /usr/lib/postgresql/14/bin/postgres -D /var/lib/postgresql/14/main -c config_file=/etc/postgresql/14/main/postgresql.conf
90695 postgres: 14/main: logger
90697 postgres: 14/main: checkpointer
90698 postgres: 14/main: background writer
90699 postgres: 14/main: walwriter
90700 postgres: 14/main: autovacuum launcher
90701 postgres: 14/main: stats collector
90702 postgres: 14/main: logical replication launcher
root      93914 90477  0 01:10 pts/0    00:00:00 grep --color=auto postgres
root@ubuntu-focal:~#
root@ubuntu-focal:~# pgrep -a postgres
90694 /usr/lib/postgresql/14/bin/postgres -D /var/lib/postgresql/14/main -c config_file=/etc/postgresql/14/main/postgresql.conf
90695 postgres: 14/main: logger
90697 postgres: 14/main: checkpointer
90698 postgres: 14/main: background writer
90699 postgres: 14/main: walwriter
90700 postgres: 14/main: autovacuum launcher
90701 postgres: 14/main: stats collector
90702 postgres: 14/main: logical replication launcher
```

Figure 14.9: Listing the postgres background processes with ps and pgrep commands

Log and events

As we saw in the first section of this chapter, having a good PostgreSQL logger to keep track of the events happening in the system is a helpful tool. In the Operating System, some options exist to get information similarly. We can review historical events and verify their sequence of relevant messages, so when debugging an issue, we can better understand what happened.

journalctl

This tool will show us the log entries from the **systemd** journal; the **systemd** is the component of the Operating System in charge of handling the running services. As we saw a few paragraphs above, we can use the **systemctl** command to verify the current state of a service and stop, start, or restart it. The **journalctl** will help us review the services' historical log entries. (*Reference: Linux Manuel journalctl*)

Access to the messages thrown during the event always comes in handy when troubleshooting. The following are a couple of options we can do with **journalctl**.

```
-- Get all the available journal entries
root@:~# journalctl

-- Get all the available journal entries for a specific service
-- (postgresql)
root@:~# journalctl -u postgresql

-- Get the journal entries for a specific service (postgresql)
-- for the last 5 minutes
root@:~# journalctl -u postgresql --since "5 minutes ago"
```

Figure 14.10 shows the output of the last command example.

```
root@ubuntu-focal:~# journalctl -u postgresql@14-main.service --since "5 minutes ago"
-- Logs begin at Thu 2022-12-15 00:09:17 UTC, end at Fri 2023-02-24 22:37:17 UTC. --
Feb 24 22:36:26 ubuntu-focal systemd[1]: Stopping PostgreSQL Cluster 14-main...
Feb 24 22:36:26 ubuntu-focal systemd[1]: postgresql@14-main.service: Succeeded.
Feb 24 22:36:26 ubuntu-focal systemd[1]: Stopped PostgreSQL Cluster 14-main.
Feb 24 22:36:26 ubuntu-focal systemd[1]: Starting PostgreSQL Cluster 14-main...
Feb 24 22:36:29 ubuntu-focal systemd[1]: Started PostgreSQL Cluster 14-main.
```

Figure 14.10: Output of `journalctl` for the `postgresql` service in the last 5 minutes

Conclusion

As a system administrator, in our case a Database Administrator, we will face situations with unexpected and never seen errors. Being able to find the cause and provide a fix is one of the more relevant responsibilities of the role.

The good thing, there are plenty of tools and techniques to debug and troubleshoot the issues. In this chapter, we have learned a few from the database and operating system points of view. We can use some of the studied queries and commands to get valuable information. And also, we reviewed the log files and event messages are a helpful resource, so knowing how to configure them and access them is essential.

In the next and final chapter, we will learn about the PostgreSQL Community and the importance of the member's contributions, and how we can be part of it.

Bibliography

- PostgreSQL Community log_line_prefix: <https://www.postgresql.org/docs/14/runtime-config-logging.html#GUC-LOG-LINE-PREFIX>
- Linux Manual free: <https://man7.org/linux/man-pages/man1/free.1.html>
- Linux Manual df: <https://man7.org/linux/man-pages/man1/df.1.html>
- Linux Manual journalctl: <https://man7.org/linux/man-pages/man1/journalctl.1.html>
- strftime: <https://strftime.org/>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 15

Contributing to PostgreSQL Community

Introduction

In the previous chapters, we have seen multiple PostgreSQL concepts and how to use, configure, and manage them from the operational point of view. This chapter will cover how to contribute to the community and a basic understanding of how the PostgreSQL Community works.

Structure

This chapter includes the following sections:

- PostgreSQL Community and its structure
 - Core Members of the PostgreSQL Community
 - Working pattern of PostgreSQL Community
 - Earning of PostgreSQL Community
- Different ways to contribute
 - Code contributor
 - Bug reporter
 - Participate in the mailing lists

- Improving or creating new documentation
- Participating in events
- Supporting the community

Objectives

The main objective of this chapter is to understand how PostgreSQL Community works and how one can contribute to improving the software and services.

PostgreSQL community and its members

Before understanding how to contribute to PostgreSQL, let us first try to understand the PostgreSQL Community and its functioning briefly in the next few sections.

Core members of the PostgreSQL community

The PostgreSQL community is made up of many contributors and developers, but there is a small group of core members who are responsible for the overall direction and management of the project. The core members of the PostgreSQL project are:

Tom Lane: Tom Lane is one of the longest-standing members of the PostgreSQL community and has significantly contributed to the project over the years. He is involved in all aspects of PostgreSQL, including new features, performance improvements, and bug evaluating and fixing. (*Reference: Tom Lane*)

Bruce Momjian: Bruce Momjian is another long-standing PostgreSQL community member, co-founder and core team member of the PostgreSQL Global Development Group. He has made significant contributions to the project, including writing much of the original documentation and helping to develop key features such as the PL/pgSQL procedural language.

Robert Haas: Robert Haas is a prominent member of the PostgreSQL community and a core development team member. He has made significant contributions to the project, including developing several key features such as parallel query execution, EXPLAIN PLAN in XML/JSON output, backup compression, backup targets, backup manifests, and many such features.

Magnus Hagander: Magnus Hagander is a core member of the PostgreSQL development team and is responsible for the PostgreSQL infrastructure to keep the services of postgresql.org up and running, including the PostgreSQL build farm, website, and mailing lists.

Peter Eisentraut: Peter Eisentraut is a long-standing member of the PostgreSQL community and a core development team member. He has made significant contributions to the project and has developed several popular PostgreSQL extensions, including pgAdmin, the PostgreSQL JDBC driver, and the PostgreSQL ODBC driver. These extensions have helped to enhance the functionality and usability of PostgreSQL and have made it easier for developers to work with the software.

These are few among many core team members of the PostgreSQL Community, and mentioning every member is outside the scope of this book. Overall, the core members of the PostgreSQL project are responsible for managing the overall direction and development of the software and play a key role in ensuring the quality and reliability of the PostgreSQL database management system.

Working pattern of PostgreSQL community

The PostgreSQL Community is an open and collaborative community that works together to develop and maintain the PostgreSQL database management system. Here are some key aspects of how the PostgreSQL Community works:

Community-driven development: PostgreSQL development is community-driven, with contributors from around the world working together to develop and improve the software. Anyone can contribute to the development of PostgreSQL, and a team of experienced developers reviews all contributions.

Governance: The PostgreSQL Global Development Group (PGDG) is the official governing body of the PostgreSQL project. The PGDG manages the overall direction and development of PostgreSQL and makes decisions on critical issues such as major feature development, release management, and community outreach.

Mailing lists and forums: The PostgreSQL community has several mailing lists and forums where developers and users can discuss various PostgreSQL development and usage aspects. These forums provide a platform for discussion, feedback, and collaboration.

Code review and testing: All contributions to PostgreSQL are subject to code review and testing. This ensures that the code is of high quality and meets the community's standards and requirements.

Release management: PostgreSQL releases are managed by the PGDG, with new releases typically occurring every year or two. Each release undergoes an extensive testing and is thoroughly reviewed before being released to the public.

User groups and conferences: The PostgreSQL community has a strong presence in the open source and database communities, with many user groups and conferences held around the world. These events provide opportunities for users and developers to network, learn, and share their knowledge and experiences.

Overall, the PostgreSQL community is a vibrant and collaborative community that values openness, inclusivity, and high-quality software development.

Earning of PostgreSQL community

The PostgreSQL Community is a non-profit organization and does not earn revenue directly. However, the PostgreSQL project receives funding and support from various sources to help sustain and grow the project. Here are some ways that the PostgreSQL Community earns support:

Donations: The PostgreSQL Global Development Group accepts donations from individuals and companies to help fund the development of PostgreSQL. Donations can be made through the PostgreSQL website or through third-party platforms such as Patreon.

Sponsorship: The PostgreSQL Community receives sponsorship from various companies that use and support PostgreSQL. Sponsorship can be in the form of financial support, in-kind contributions, or contributions of developer time.

Consulting services: Many companies offer consulting services for PostgreSQL, providing support, training, and development services to software users. Some of these companies also contribute to the PostgreSQL project through sponsorship or developer time.

Commercial support: Some companies offer commercial support for PostgreSQL, providing support and services to enterprise software users. These companies typically charge for their services but also contribute back to the PostgreSQL project through sponsorship or developer time.

Overall, the PostgreSQL Community relies on the support of its users and supporters to sustain and grow the project. The community is committed to keeping PostgreSQL open source and freely available to all, while also ensuring that the software is of high quality and meets the needs of its users.

Different ways to contribute

There are multiple different ways to contribute to the PostgreSQL Community. If you are interested in being part of this vibrant community, you are plenty of options. In the following paragraphs, we will review the main options you have.

Code contributor

Becoming a code contributor might be among the most engaging and rewarding experiences, but you need to consider this also is one of the contribution options that demand a high knowledge and expertise. (*Reference: PostgreSQL Developer FAQ*)

Before starting to work on a code contribution, you need to be highly familiar with PostgreSQL, know how it works, and its design goals. Fortunately, the code base is available for everyone at the PostgreSQL git repository, you can start your path to becoming a code contributor by getting and studying it.

This is obvious, but being a code contributor means you will participate in the community development processes. The PostgreSQL project has its current status thanks to the highly collaborative development processes. Hence getting involved in the community discussion about bugs, fixes, and enhancements is important.

Another important aspect is to have a clear goal and choose the opportunity area to focus on. The PostgreSQL code is large, robust, and complex, the best option is to define a specific feature or module for which you have identified an opportunity for improvement.

Getting familiar with the code and its design, participating in community discussions, and focusing on a specific module or area will lead us in the correct direction. Once we start working with the code, the next would be the stages:

- **Submit a patch:** Once your code, or patch, is ready, you must submit it to the pgsql-hackers mailing list for review. It needs to be well-documented and follow the code conventions.
- **Actively participate in the review process:** The community may share feedback and suggestion about the patch, it is important to engage and work collaboratively to refine it and ensure it aligns with the general project goals.
- **Get your patch accepted:** After refining and reviewing, the patch will be approved and accepted into the codebase.

Contributing to the codebase will allow you to work and share knowledge with true experts from around the globe. Following the collaborative processes, you can help to make PostgreSQL even better and more powerful.

Bug reporter

Another option to contribute is reporting the bugs we might hit when using, testing, installing, upgrading, and the like, our PostgreSQL databases. Reporting bugs will enable the developers to find fixes to solve them and prevent affecting other users.

To report a bug, there are a few guidelines we can take for a more effective process:

- **Verify if the bug has not already been identified and reported:** The PostgreSQL Community has a TODO list with the already identified bugs; before reporting any, we must verify it. (*Reference: PostgreSQL Bugs and Features*)
- **Create a bug report:** The PostgreSQL Community already has a bug form we can use to report any new bug. It is important to follow the guidelines for bug reports and include all the required information. This will help the developers to reproduce the bug and find the source.
- **Collaborate with the developers:** Following the collaborative perspective of the PostgreSQL Community, you may receive feedback and requests for extra input, it is essential to keep involved in the process so the fix can be addressed as quickly and efficiently as possible.

Identifying bugs is a fundamental way to collaborate and keep PostgreSQL stable and reliable for everyone. Remember, wrong, unclear, or missing documentation is also a bug.

Participate in the mailing lists

The PostgreSQL Community uses different and multiple mailing lists for various purposes. You can collaborate with the community by joining the discussions and sharing knowledge and experience. (*Reference: PostgreSQL Mailing Lists*)

The following are the main aspects you can follow when thinking about participating in the mailing lists:

- **Choose a mailing list:** As said before, there are multiple mail lists for a variety of topics, so you can start by selecting one or two in which you have a special interest or expertise and subscribe to it.
- **Read the mailing list archive:** Before start posting new messages to the mailing list is a good idea to go through the archives; this way, you can get a good sense of the tone and topics for the discussions. Also, it will help to prevent you from posting about already discussed and closed topics.

- **Follow the community guidelines:** The community has adapted some standards to keep the conversation going respectfully and consider others' time and contributions. You need to follow the same.
- **Participate in the discussions:** Contribute to the discussion by sharing your perspective and experience and asking questions. Also, if you know how to answer some of the questions, please do; this will help build your reputation and establish you as a valuable contributor.

Improving or creating new documentation

Getting involved with the Community by helping with the PostgreSQL documentation is also a great option. There is no doubt the official documentation is where everybody looks for references and details while working with our loved PostgreSQL, from anyone taking their first steps into PostgreSQL to experts who need to refer to a particular feature.

Contributing to the project documentation will require you to get familiar with the current documentation, so you know the style, the topics, and the guidelines. Like the previous contribution options, you will need to define an area or module to focus on and get involved with the collaborative work. The following are the main aspects to consider when contributing to the documentation:

- **Get involved with the community:** There is a special mailing list for the PostgreSQL documentation: `pgsql-docs`. You will get the community documentation style guide by joining it and participating. Also, you can identify and confirm an actual documentation opportunity area.
- **Make your contribution:** Once you have defined the area, feature, or module, you can work on your contribution. This could be reporting documentation bugs, fixing them, adding content, or writing new documentation from scratch.
- **Collaborate with the community:** As in previous options, collaborating with the document may require you to share efforts with more community members while receiving feedback or questions to refine your document. Please engage and keep the collaborative spirit.
- **Be persistent:** Documentation is a permanent ongoing effort, invariably, there will be room for improvements. Being persistent and making contributions over time will help to keep PostgreSQL documentation accurate, up-to-date, and accessible for everyone.

Participating in events

The PostgreSQL Community includes many members, organizations, and enterprises. Being as active and vibrant as always, these groups usually organize different events around the globe. These events are a truly awesome way to expand PostgreSQL knowledge and share the experience and enthusiasm for this technology with everyone.

Participating in such events is another fantastic option to contribute to the community. You might consider the following options:

- **Attend a PostgreSQL event:** This is the most straightforward way to participate in an event. It provides an excellent option to network with other community members, learn about new features and best practices, and get involved with the community's ongoing efforts.
- **Speak at a PostgreSQL event:** If you have experience and knowledge working with PostgreSQL, consider submitting a talk proposal to the event. Giving a talk at an event is a terrific way to build your presence as an expert in the community and helps to share knowledge and insights with other members.
- **Volunteer at a PostgreSQL event:** Many events, in part, rely on altruistic volunteer efforts. The volunteers can help with everything, from event registration to planning or technical support. Collaborating this way can ensure the event's success. Also, it is a good way to get involved with the community and engage with the members.
- **Organizing a PostgreSQL event:** If you have some experience in this regard, you can organize an event in your local area. This can be from a small group meeting to a full-blown conference. Even when this option involves a lot of work, it will help to build a network, establish yourself as a leader in the community, and contribute to the ongoing success of PostgreSQL.

The event options are vast, listing some of the most important: (*Reference: PostgreSQL Events*)

- **PostgreSQL Conference:** This is the official PostgreSQL conference organized by the PostgreSQL Global Development Group. It is held annually in different locations worldwide and includes talks and workshops covering various PostgreSQL-related topics.
- **PGConf:** PGConf is a series of PostgreSQL conferences held in different locations around the world. The PostgreSQL community organizes these events and typically includes talks, workshops, and networking opportunities for PostgreSQL users, developers, and contributors.

- **PostgresOpen:** PostgresOpen is an annual conference for PostgreSQL users and developers held in different locations around the world. The conference includes talks, tutorials, and workshops covering various PostgreSQL-related topics.
- **FOSDEM:** FOSDEM is a free and open-source software conference held annually in Brussels, Belgium. While not exclusively focused on PostgreSQL, the conference typically includes talks and workshops related to PostgreSQL and opportunities to meet and network with other PostgreSQL community members.

Supporting the community

All the previous options to contribute to the PostgreSQL Community are related to your experience with PostgreSQL or your availability to participate directly in the different aspects the community works on. However, there is still at least another option to participate with the community, which does not require your having experience or time to get involved with specific work. This is being a financial sponsor.

If you can support the community financially, you can do it at least in the two following ways:

- **Donate directly to the PostgreSQL project:** PostgreSQL is a free and open-source project, but it relies on donations to support ongoing development and maintenance. You can donate directly to the PostgreSQL project through the PostgreSQL Foundation's website. (*Reference: PostgreSQL Donate*)
- **Sponsor PostgreSQL events:** Many PostgreSQL events rely on the support of sponsors to cover costs and provide additional resources for attendees. By sponsoring a PostgreSQL event, you can help ensure that the event is a success and contribute to the ongoing development of PostgreSQL.

Conclusion

With this, we came to the end of this chapter and the end of this book. To conclude, we have covered not only the concepts of PostgreSQL but also how the PostgreSQL Community is managed and how one can contribute to the community.

The PostgreSQL Community is one of the most important parts of the PostgreSQL software. The software will survive as long as Community is there; the vice versa may/may not be true. It is WE, as a user, who can make the community stronger by contributing not only in the form of code but also in other ways discussed in this chapter.

Bibliography

- Tom Lane: [https://en.wikipedia.org/wiki/Tom_Lane_\(computer_scientist\)](https://en.wikipedia.org/wiki/Tom_Lane_(computer_scientist)))
- PostgreSQL Developer FAQ: https://wiki.postgresql.org/wiki/Developer_FAQ#Getting_Involved
- PostgreSQL Bugs and Features: https://wiki.postgresql.org/wiki/FAQ#How_do_I_find_out_about_known_bugs_or_missing_features.3F
- PostgreSQL Mailing Lists: <https://www.postgresql.org/list/>
- PostgreSQL Events: <https://www.postgresql.org/about/events/>
- PostgreSQL Donate: <https://www.postgresql.org/about/donate/>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

A

aggregate functions 209
Amazon Web Services (AWS) 46
archiver 79
archive recovery 139, 140
Atomicity 92
Atomicity, Consistency, Isolation,
and Durability (ACID) 16, 92
authentication
about 152
address 155, 156
database 154, 155
method 156
[Options] 156, 157
pg_hba.conf 152, 153
user 155
authentication methods
about 157
pg_ident.conf 158, 159
authorization
about 161
object ownership 163
object privileges 164, 165
role attributes 161, 162
autovacuum 79, 94

B

background processes
about 77
archiver 79
autovacuum 79
background writer 78
checkpointer 78
logger 79
postmaster 77
stats collector 79
WAL receiver 81
WAL sender 80
WAL writer 80
writer 78
background writer 78
backup
about 108
logical backup 112
pg_basebackup 109
backup and restore tools
about 118
Barman 125, 126
pgBackRest 119-125
pg_probackup 127-133
Barman 125, 126

Binary installation

about 33, 34

package list, updating 35

repository configuration, creating 34

repository signing key, importing 35

Block Range INdexes (BRIN) 237

Brin index 237

b-tree index 236

C

cascading 142, 143

checkpoint 78

Classless Inter-Domain Routing (CIDR) 155

CLOG buffer 75

Command-Line Interface (CLI) 174

Commit Log (CLOG) 75

Commit LOG Files (CLOG Files) 86

Consistency 92

constraints

used, for enforcing data integrity 197, 199

CREATE DATABASE command 67, 68

createdb program 68

cron 170

custom data type 230

D

data

manipulating, with DML Queries 201

database

about 65, 66

CREATE DATABASE command 67, 68

createdb program 68

pgAdmin Wizard, using 69-72

Database Administrators (DBAs) 179, 233

Data Definition Language

(DDL) 147, 185, 192

data files 84

data integrity

enforcing, with constraints 197-200

data types 192, 193

DB cluster 186

DB Objects

data types 192, 193

managing, with DDL commands 192

table 193

DDL commands

used, for managing DB Objects 192

debugging

with log files 256

with Operating System tools

and commands 268

with PostgreSQL tools and commands 261

Defense Advanced Research

Projects Agency (DARPA) 6

delayed replica 144-146

df command 270

Digital Ocean (DO) 46

Disaster Recovery (DR) models 135

distributed architecture

with PostgreSQL 17

distributed architecture, PostgreSQL

data partitioning 18

horizontal scaling 17

replication 18

DML Queries

about 102, 147, 185, 201

aggregate functions 209, 210

data, deleting 203

data, inserting 201, 202

data, selecting 203

data, updating 202

full outer join 208

inner join 204, 205

joins, using in data retrieval 204

used, for manipulating data 201

Docker 47

Docker Hub

reference link 47

Domain Name System (DNS) 156

Durability 92

E

event trigger 227
 executor 105
 explain plan 248, 250
 expressions 237
 extension
 about 168
 pg_cron 169, 170
 pg_repack 174-177
 pg_stat_statements 171, 172
 extension tools
 about 177
 pgbadger tool 177-179
 pgbench tool 179-181
 pgbouncer tool 181-183

F

Free and Open-source Software (FOSS) 7
 free command 269
 Free Software Foundation (FSF) 3, 5
 full outer join 208
 function
 about 212, 213
 managing 212
 function execution
 example 215
 syntax 214

G

General Public License (GPL) 3
 Generic Security Service Application Program Interface (GSSAPI) 154
 Geographic Information Systems (GIS) 17
 Gin index 237
 GiST index 237
 Google Cloud Platform (GCP) 46
 groups 60-63

H

hash index 237

High-Availability (HA) 135
 hostnogssenc 154
 hypervisor 43

I

indexes
 about 234, 235, 238
 reindex 235, 236
 index types
 about 236
 Brin index 237
 Btree index 236
 Gin index 237
 GiST index 237
 hash index 237
 SP-GiST index 237

inner join
 about 204, 205
 left outer join 206
 right outer join 207
 Isolation 92

J

journalctl 272

L

left outer join 206
 libre software 2
 Linux kernel 4
 log files
 about 86
 debugging with 256
 log_autovacuum_min_duration 260
 log_connections 260
 log_destination 257
 log_directory 257
 log_disconnections 260
 log_filename 258
 logging 259
 logging_collector 256
 log_line_prefix 259

log_lock_waits 260
log_min_duration_statement 260
log_rotation_age 258
log_rotation_size 258
log_truncate_on_rotation 258
parameter summary 261
storing 256
logger 79
logical backup
 about 112
 cons 115
 examples 118
 pg_dump 112
 pg_dumpall 112, 114
 pros 115
logical replication
 about 146
 architecture 147
 limitations 150
 publication 147
 publisher node 148, 149
 subscription 148
 subscription node 148, 149
 use cases 149
log_line_prefix 259
Log Sequence Number (LSN) 85

M

manual VACUUM
 about 95
 options 95
pg_repack command 96
phases 96
memory architecture
 about 74
 maintenance work memory 77
 process memory 76
 shared buffer 75
 shared memory 74
 WAL buffer 75
 work memory 76

Microsoft Azure 46
MINIX 4
Multiversion Concurrency Control (MVCC) 8, 16, 75, 93, 174

N

National Science Foundation (NSF) 6
Netscape browser 4

O

open-source
 about 2
 concept 3-5
 free software 2, 3
 overview 5, 6
Operating System tools and commands
 debugging with 268
 log and events 272
 pgrep commands 271
 processes-oriented tools 270
 ps commands 271
 service and system-wide tools 268

P

parser
 about 103
 join tree 104
 others 104
 qualification 104
 Range Table Entry (RTE) 104
 result relation 104
 target list 104
 type 104
PersistentVolumeClaim (PVC) 48
PersistentVolume (PV) 48
pgAdmin Wizard
 using 69-72
pgBackRest 120-125
pgbadger tool 177-179
pgbench tool 180, 181
pgbouncer tool 181-183

pg_cron 169
pg_dump 112
pg_dumpall 112, 114
pg_hba.conf
 about 152, 153
 host 153
 hostgssenc 154
 hostnogssenc 154
 hostnossal 154
 hostssl 154
 local 153
pg_ident.conf
 about 158, 159
 examples 159-161
pg_probackup 127-133
pg_reorg 176
pg_repack 174-176
pg_repack command 96
pg_restore 117
pg_stat_statements 171, 172
phantom read 99
phenomena types
 dirty read 97
 non-repeatable read 98
physical backup
 about 109
 cons 111
 pros 111
physical files
 Commit LOG Files (CLOG Files) 86
 log files 86
 stat files 86
 temporary files 85
 WAL archive files 86, 87
 WAL files 84, 85
physical file structure
 data files 84
 defining 81-83
physical replication
 about 136-138
 archive recovery 139, 140

} cascading 142, 143
configuration 146
delayed replica 144-146
hot standby 138, 139
streaming replication 141, 142
planner 104
Point in Time Recovery (PITR) 17, 111
PostGIS extension 17
Postgres 1, 61
Postgres95 7
POSTGRES project 6
PostgreSQL
 2ndQuadrant 15
 about 7, 42
 advantages 16
 cloud 45, 46
 containers 44, 45
 database 31
 data directory, validating 30
 distributed architecture 17
 enhancing 14, 15
 EnterpriseDB 15
 extension, adding 173
 history 6
 initializing 28
 installing 36-38
 market impact 12, 13
 on cloud 53-57
 on Docker 47, 48
 on Kubernetes 48, 49, 52, 53
 on modern systems 46
 on-premise 42, 43
 Pivotal 15
 Postgres95 7
 postgres process, verifying 31, 32
 POSTGRES project 6
 statefulset protocol, using 18
 using 14
 virtualization 43, 44
PostgreSQL community
 about 276

bug reporter 280
code contribution 279
core members 276, 277
documentation, creating 281
documentation, improving 281
earning 278
event participating 282, 283
mailing lists, participating 280, 281
reference link 86
supporting 283
working pattern 277, 278

postgresql.conf parameters, best practices
 about 250
 autovacuum 251
 effective_cache_size 252
 maintenance_work_mem 252
 max_connections 252
 shared_buffers 251
 work_mem 251

PostgreSQL members 276
PostgreSQL release cycle 10, 11
PostgreSQL stats
 on single image 10
PostgreSQL tools and commands
 ANALYZE operations, executing 266
 database and object size, checking 262
 database connections, checking 263
 debugging with 261
 Instruct PostgreSQL 266
 PostgreSQL version, checking 262
 replication, managing 267
 slow queries, checking 264
 statistics, checking 264, 265
 VACUUM operations, executing 266

PostgreSQL versions
 key features 8
 versions 6.0 - 8.0 8
 versions 8.1 - 9.6 8, 9
 versions 10 - 14 9, 10
postmaster 77

} predefined roles 63
procedure
 about 216, 217
 managing 212
procedure execution
 about 219, 220
 syntax 217-219
processes-oriented tools
 about 270
 htop tool 270
 top tool 270
process memory
 about 76
 temporary buffer 76
psql command 115, 116
public schema
 about 188
SEARCH_PATH attribute 189-191

Q

queries
 terminating 266
query processing
 about 102, 103
 executor 105
 parser 103
 planner 104
 rewriter 104

R

reindex 235, 236
Relational Database Management System
 (RDBMS) 73, 97, 186, 192
Relational Database Service (RDS) 54
replication & high availability
 advance features 17
restore
 about 115
 pg_restore 117
psql command 115, 116

rewriter 104
 right outer join 207
 role attributes 161, 162
 roles 60-63
 rules
 managing 227, 228
 versus trigger 230

S

schemas
 about 188
 database 188
 DB cluster 186, 187
 managing 186
 public schema 188
 tablespaces 188
 users/roles 188
 SEARCH_PATH attribute 189, 190
 serialization anomaly 99, 100
 service and system-wide tools
 about 268
 df command 270
 free command 269
 systemctl 268, 269
 shared buffer 75
 shared memory 74
 source code installation
 about 22
 downloading 23, 24
 pre-requisites 23
 version 22, 23
 source code installation process
 about 25, 26
 data directory, creating 28
 directory structure, verifying 27
 postgres user, adding 27
 SP-GiST index 237
 statefulset protocol
 with PostgreSQL 18

} stat files 86
 statistics
 about 238-240
 common values list 246, 247
 functional dependencies 243, 244
 in pg_statistics 240-242
 in pg_statistics_ext_data 242, 243
 number of distinct values counts 244, 245
 system catalogs 238
 types 238
 stats collector 79
 streaming replication 141
 subscription 148
 superuser 161
 systemctl 268, 269

T

table
 about 193
 altering 194
 creating 193
 dropping 194
 sequences 195, 196
 truncating 194
 viewing 195
 tablespace 64, 65
 temporary buffer 76
 temporary files 85
 transaction isolation levels
 about 97
 phenomena 97
 phenomena isolation levels 100
 read committed 100
 read uncommitted 100
 repeatable read 101
 Serializable 102
 Transmission Control Protocol/
 Internet Protocol (TCP/IP) 153
 trigger
 versus rules 230

trigger function 222-226

triggers

managing 222

U

UNiplexed Information Computing System (UNIX) 4

users 60-63

user sessions

terminating 266

V

vacuum

about 94

autovacuum 94

manual VACUUM 95

transaction ID wraparound
failures, preventing 97

VACUUM FULL 95

VACUUM FULL 95

Virtual Machines (VM) 43

W

WAL archive files 86, 87

WAL buffer 75

WAL files 84, 85

WAL receiver 81, 141

WAL sender 80, 141

WAL writer 80

work memory 76

Write-Ahead Logging (WAL) 8, 16, 75, 80, 136

writer 78

PostgreSQL for Jobseekers

DESCRIPTION

PostgreSQL is a powerful open-source relational database management system (RDBMS) that is widely used in the industry. If you are seeking to acquire knowledge about PostgreSQL, this book is for you.

This comprehensive book provides you with a solid foundation in working with PostgreSQL, a popular open-source database management system. It covers a broad spectrum of topics, allowing you to successfully install and configure PostgreSQL across various platforms and methods. By delving into the internal components that constitute a PostgreSQL service and their interplay, you will gain a deep understanding of how these elements collaborate to deliver a robust and dependable solution. From comprehending the process model and shared memory to mastering query execution and optimization, you will acquire comprehensive knowledge of PostgreSQL's internal workings. Furthermore, the book explores essential tasks performed by a database administrator (DBA), including backup and restore operations, security measures, performance tuning, and troubleshooting techniques. Lastly, it explores widely used extensions and compatible tools that can enhance the functionality of PostgreSQL.

Upon completing this book, you will have developed a comprehensive understanding of the internal components that comprise a PostgreSQL service and their collaborative dynamics, resulting in a reliable and robust solution.

KEY FEATURES

- Acquire in-depth knowledge of PostgreSQL's key capabilities and gain a comprehensive understanding of its inner workings.
- Discover the art of extending PostgreSQL's core features and effectively troubleshooting any challenges that may arise.
- Explore the vibrant community and open-source ecosystem that forms the foundation of PostgreSQL's development and innovation.

WHAT YOU WILL LEARN

- Gain proficiency in installing and preparing PostgreSQL for various methods and platforms.
- Develop a solid understanding of the internal components of a PostgreSQL service and their collaborative dynamics to deliver a comprehensive solution.
- Acquire knowledge about essential tasks performed by PostgreSQL DBAs, including backup/restore operations, security measures, tuning, and troubleshooting.
- Explore popular extensions and compatible tools that can expand and enhance the capabilities of PostgreSQL.
- Discover the PostgreSQL Community and learn how to actively contribute to the project's development and growth.

WHO THIS BOOK IS FOR

This book is highly recommended for Entry Level Database Administrators, as it provides a suitable starting point for their journey. It assumes some prior knowledge of Database Management Systems (DBMS) to ensure a smooth learning experience. Additionally, senior or experienced developers will find value in this book, particularly in gaining insights into the latest features incorporated in the most recent version of the DB, enhancing their understanding and proficiency in its use.



BPB PUBLICATIONS

www.bpbonline.com

ISBN 978-93-5551-400-4



9 789355 14004