

# PostgreSQL 15 Cookbook

Efficient data management with 100+ recipes



Mohammad Samsad Hussain

bpb

# PostgreSQL 15 Cookbook

Efficient data management with 100+ recipes



Mohammad Samsad Hussain

bpb

# **PostgreSQL 15 Cookbook**

---

*Efficient data management  
with 100+ recipes*

---

**Mohammad Samsad Hussain**



[www.bpbonline.com](http://www.bpbonline.com)

First Edition 2024

Copyright © BPB Publications, India

eISBN: 978-93-55519-689

*All Rights Reserved.* No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

## **LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY**

The information contained in this book is true to correct and the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete  
BPB Publications Catalogue  
Scan the QR Code:



[www.bpbonline.com](http://www.bpbonline.com)

**Dedicated to**

*The readers*

# About the Author

**Mohammad Samsad Hussain** is based in Malaysia. He is a Database Architect and consultant with more than 13 years of experience in different roles (Data Analyst, Database Solution Architect, Application Consultant) among software industries. He is an IBM DB2 DBA Certified Professional and PostgreSQL, MySQL & MariaDB database Consultant as he has worked on different technologies and architectures, such as SAP HANA, Oracle, MongoDB, MS SQL and IBMi (As400).

Samsad is currently employed as Database Infrastructure Solution Architect in a Utilities based Organization in Malaysia, and he is in charge of databases and systems management and infrastructure.

# About the Reviewers

- ❖ **Sami Lekesiz**, born in Turkiye, graduated from Istanbul Technical University. He has been working exclusively with PostgreSQL for the last five years of his career. His expertise relates to various PostgreSQL projects where he has played a crucial role in numerous critical tasks such as migrations, building highly available cluster architectures, and system optimizations. With his extensive knowledge and hands-on experience, Sami has been instrumental in the successful delivery of multiple PostgreSQL projects. He has always focused on delivering excellence in tasks that are central to these projects. He has ensured seamless migrations, set up highly available clusters from scratch, and implemented critical system enhancements. Looking ahead in his career, Sami aims to continue his dedicated focus on PostgreSQL, aspiring to further specialize and contribute within this domain.
- ❖ **Regina Obe** is the president of Paragon Corporation, a PostgreSQL database consulting company located in Boston, Massachusetts, USA. She has over 25 years of experience in various programming languages and database systems, especially PostgreSQL and spatial databases. She is a member of the project steering committee and development teams of two popular PostgreSQL spatial extensions, PostGIS and pgRouting. She is also a charter member and System Administration support team member for Open Source Geospatial Foundation (OSGeo), which focuses on developing and evangelizing free and open-source geospatial software.

She holds a BS degree in mechanical engineering from the Massachusetts Institute of Technology.

# Acknowledgement

I want to thank everyone who helped bring this book to life. Special thanks to my friend and colleague, **Abhishek Chaturvedi** and **Mohd Zaidi b. Mohd Yusoff** for their valuable contributions and strong support. Thanks to my family and friends for always cheering me on. A big thank you to the readers.

I also want to give a special thanks to the publisher of this book, BPB Publications, for believing in this project and providing the platform to make it happen. Your expertise and dedication made this dream come true. Thanks to my colleagues too, for their support and insights, adding depth to this work. Together, we have created something special, and I am truly thankful for this collaborative journey.

# Preface

Welcome to **PostgreSQL 15 Cookbook**, PostgreSQL 15 stands out as a robust and reliable SQL database server. This book is here to help you understand and master PostgreSQL 15 in a practical and straightforward way.

In this book, we cover various aspects of PostgreSQL 15, providing insights and hands-on guidance. Whether you are just starting with PostgreSQL 15 or delving into advanced topics like database hierarchy, cloud provisioning, migration, and more, each chapter is crafted to improve your understanding and skills.

PostgreSQL 15 is essential in managing data. Whether you are new to it or experienced, this recipe guide makes PostgreSQL 15 accessible and valuable. We focus on real-world applications, showing the importance of each topic in practical situations.

This book is divided into 13 chapters, each focused on a specific aspect of PostgreSQL 15. Whether you are exploring the basics, advancing your skills in managing PostgreSQL 15, or troubleshooting problems, our goal is to present information that is clear, concise, and practical. Our goal is to equip you with the knowledge to use PostgreSQL 15 effectively.

**Chapter 1: Up and Running with PostgreSQL 15** - In this chapter, we will learn the basics of databases and their core architecture concepts, seeing how these elements work together. The focus is on starting with PostgreSQL, giving you simple instructions for installing and setting it up on

Linux VMs. We will also cover database upgrades to both minor and major releases.

**Chapter 2: Database Hierarchy** - We will work on creating and using databases in PostgreSQL. We will cover best practices for user access and authorization, the hierarchy of database objects, tables inheritance, concurrent indexes, and workarounds with various database objects.

**Chapter 3: Cloud Provisioning** - In this chapter, we will work on setting up a PostgreSQL cloud instance and handling database connections using AWS EC2 and RDS instance. We will also explain how to perform native backup and restore procedures for both AWS EC2 and RDS instances. Additionally, we will explore managing PostgreSQL on the cloud, including aspects like connection management and replication.

**Chapter 4: Migration** - This chapter explores different migration options for moving SQL databases to PostgreSQL and transferring on-premises data to the AWS cloud. We will work on each migration method, including the use of tools like PgLoader. Additionally, we will explore the best practices to ensure a seamless and efficient migration process.

**Chapter 5: Transaction Log** - We will explore admin and application-specific logs, discover practical solutions, and gain a detailed understanding of PostgreSQL transaction logs. We will examine the crucial role of the Transaction Log in PostgreSQL replication, discussing topics such as Archive mode and WAL management.

**Chapter 6: Partitioning and Sharding** - We will discuss in detail the concepts of scaling up your PostgreSQL database based on your organization's limitations and operational needs. The goal is to provide practical concepts for enhancing the performance of your business-critical

data, covering topics such as partitioning, sharding, and the role of Foreign Data Wrapper.

**Chapter 7: Replication and High Availability** - In this chapter, we will learn replication and High Availability solutions for PostgreSQL, understanding the best fit for different practical scenarios and business needs. We'll also dive into the concepts of Load balancing in PostgreSQL, offering a mix of theoretical understanding and practical solutions.

**Chapter 8: Leveraging SQL** - In this chapter, we will talk about how to get to know and work with data in the database. Once you understand the basics of PostgreSQL, this section will guide you in using SQL to interact with relational data stored in PostgreSQL databases. We will also cover how to organize query access using psql, explore Postgres JSON Query, and understand the Postgres CAST Operator.

**Chapter 9: Server Controls and Auditing** - We will explore the basic concepts of authentication and encryption in the PostgreSQL database server. We'll look into common methods to encrypt your client connections and provide practical solutions for user and group control to ensure flexible access control to your database. Additionally, we have a dedicated section on database authentication through SSL.

**Chapter 10: Backup** - We will explore the basics of database backup, offering practical examples and insights into various backup tools, such as pg\_probackup and pgBackRest. We will also discuss planning for backups and strategies to improve backup performance.

**Chapter 11: Recovery** - This chapter covers the fundamentals of database recovery. We will explore the concept, discuss planning for recovery, understand crash

recovery, and explore Point-in-Time Recovery (PITR) and table-level recovery.

**Chapter 12: Monitoring and Diagnosis** - In this chapter, we will discuss how to make your setup more reliable by adding monitoring for crucial actions in your database. We will look into different monitoring approaches for database components. Additionally, we will check out tools like Prometheus and Grafana, explore statistics, find monitoring scripts, and understand the significance of fsync.

**Chapter 13: Troubleshooting** - This chapter will explore various situations where a PostgreSQL database can be beneficial. We will provide practical solutions to help database administrators manage their databases effectively. The topics covered include transaction logs and checkpoints, benchmarking with pgbench, and utilizing data checksums with pg\_checksums.

# **Code Bundle and Coloured Images**

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

**<https://rebrand.ly/j6vnarp>**

The code bundle for the book is also hosted on GitHub at **<https://github.com/bpbpublications/PostgreSQL-15-Cookbook>**. 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](mailto: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](http://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](mailto:business@bpbonline.com)** for more details.

At [www.bpbonline.com](http://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](mailto: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](http://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](http://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>**



# Table of Contents

## 1. Up and Running with PostgreSQL 15

Introduction

Structure

Objectives

Introduction to PostgreSQL 15

Exploring the history of PostgreSQL

Learning about PostgreSQL architecture

Learning about PostgreSQL database structure

Installation methodology

*Recipe 1: Working with installation from binaries*

*Recipe 2: Working with installation from source code*

*Recipe 3: Database configuration file*

Database server and client environment

*Recipe 4: Installing and configuring pgAdmin client tool*

*Recipe 5: Working with PostgreSQL server access solution*

*Recipe 6: Working with PostgreSQL remote access*

*Recipe 7: Discovering PostgreSQL database structural object*

Exploring backward compatibility

*Recipe 8: Working with upgrade on minor release*

*Recipe 9: Working with upgrade on major release*

Conclusion

## 2. Database Hierarchy

Introduction

Structure

Objectives

Access management

Database instance and object hierarchy

*Recipe 10: Adaptation with PostgreSQL Schema*

Advanced object-relational table concept

*Recipe 11: Getting insight to tables inheritance*

*Recipe 12: Indexing technique with PostgreSQL database*

*Recipe 13: Workaround with concurrent index*

Common table expression

*Recipe 14: Getting insight to SELECT in WITH queries CTE*

Discover database storage layout

*Recipe 15: Adapt database storage for PostgreSQL database*

*Recipe 16: Adapt database log directory*

*Recipe 17: Workaround with PostgreSQL TOAST*

*Recipe 18: Parse the database admin specific start-up logs*

*Recipe 19: Getting insight to PostgreSQL memory configuration*

*Local memory area*

*Shared memory area*

Conclusion

## 3. Cloud Provisioning

Introduction

Structure

Objectives

Introduction to cloud solution for PostgreSQL

*Cloud service model*

*Cloud deployment option*

Managed or self-managed options

Exploring EC2 and RDS instance

*Recipe 20: Manage PostgreSQL instance with AWS EC2*

*Recipe 21: Managing PostgreSQL instance with AWS RDS*

*Recipe 22: Native backup or restore with AWS EC2 instance*

*Recipe 23: Backup or restore with AWS RDS instance*

*Recipe 24: Working with replication on AWS for PostgreSQL*

Conclusion

## **4. Migration**

Introduction

Structure

Objectives

Introduction to migration for PostgreSQL

Exploring migration methodology

Understanding PostgreSQL database system migration types

*Recipe 25: On-premise to AWS EC2 instance migration*

*Database migration roadmap*

*Recipe 26: Migrating PostgreSQL from EC2 to RDS instance on AWS*

Getting ready with pgloader

*Recipe 27: Setting up pgloader*

*Recipe 28: Getting insight to migrate MariaDB to PostgreSQL on EC2 instance*

*Recipe 29: Using db2topg for migration to PostgreSQL*

*Recipe 30: Setting up foreign data wrapper for PostgreSQL migration*

Conclusion

## 5. Transaction Log

Introduction

Structure

Objectives

Introducing transaction log

Understanding WAL

Exploring the internal layout of WAL segment

Exploring the internal layout of XLOG record

Configuring the WAL file

*PostgreSQL WAL benefits*

*Recipe 31: How to enable/disable archive mode*

*Recipe 32: Working with remote WAL Archive options*

*Recipe 33: Working with WAL compression option for space management*

Explore SQL statement transaction sizing

*Recipe 34: Configuring and managing WAL performance parameter*

*Recipe 35: Administering continuous archiving*

Managing the WAL writer process

*PostgreSQL 15 WAL writer process versus checkpoint processing*

Optimizing checkpoint processing

Automating vacuum process

*Recipe 36: Getting insight to vacuum process in PostgreSQL database*

*Recipe 37: Debug PostgreSQL autovacuum problem*

Implementing continuous archiving and archive logs

*Recipe 38: PostgreSQL 15 continuous archiving and archive logs examples*

Enhancing performance and benefits of WAL

*Recipe 39: Proactive solution to delete PostgreSQL archive logs*

Conclusion

## **6. Partitioning and Sharding**

Introduction

Structure

Objectives

Partitioning and sharding

Partitioning classifications

*Recipe 40: Setup and exploit partitioning in PostgreSQL*

*Recipe 41: Partition management with PostgreSQL 15*

*Recipe 42: Getting insight to vertical and horizontal partitioning*

*Horizontal partitioning*

*Vertical partitioning*

*Recipe 43: Working with automatic partition with PostgreSQL*

Manage partition using pg\_partman extension

*Recipe 44: Manage partition using custom scripts with a scheduler*

*Recipe 45: Getting insight to partition pruning*

Partitioning versus sharding

Shard strategies and rebalancer

*Recipe 46: Configure sharding with Citus Data Rebalancer*

Foreign data wrapper

Conclusion

## 7. Replication and High Availability

Introduction

Structure

Objectives

Replication and high availability

Understanding the CAP theorem

Replication classification

*Recipe 47: Scaling Postgres with primary-standby replication*

Using replication slots

*Recipe 48: Scaling Postgres with multi-master replication*

*Recipe 49: Setup delay standby in PostgreSQL*

*Recipe 50: Performing PITR recovery using delay standby*

*Recipe 51: PostgreSQL 15 promote standby database to primary*

*Recipe 52: Installing and configuring repmgr for PostgreSQL cluster*

Switchover versus failover

*Recipe 53: Switchover with repmgr for PostgreSQL*

*Recipe 54: Failover with repmgr for PostgreSQL*

Understanding proxy and load balancing

*Recipe 55: Deploying PostgreSQL Automatic Failover  
for HA solution*

*Recipe 56: Using HAProxy for HA solution*

Monitoring replication

Conclusion

## 8. Leveraging SQL

Introduction

Structure

Objectives

Exploring data access

Working with tables and data

*Recipe 57: Tables and Data Operations*

*Table creation*

*Data insertion*

*Data retrieval*

*Data modification*

*Table operations*

*Data integrity and transactions*

*Recipe 58: Querying PostgreSQL data*

*Recipe 59: Querying using shell script in PostgreSQL*

Exploring DML, DDL, TCL and DCL

*Recipe 60: Database Query and Control Language*

*Data manipulation language*

*Data definition language*

*Transaction control language*

*Data control language*

*Recipe 61: DDL adaptation*

*Recipe 62: Working with dataset export/import*

*Recipe 63: Dataset load from spreadsheet/flat files*

*Recipe 64: Structuring query access with psql*

*Recipe 65: Exploring PostgreSQL join and subqueries*

*Recipe 66: Querying JSON data*

*Recipe 67: Working with PostgreSQL CAST operator*

*Recipe 68: Working with database consistency and integrity*

*Unique constraints*

*Foreign key constraints*

*Recipe 69: Getting insight to Python and Java connection*

*Recipe 70: Importing BLOB data types into PostgreSQL*

Conclusion

## **9. Server Controls and Auditing**

Introduction

Structure

Objectives

Introduction to server control and auditing

*Recipe 71: Database privileges*

*Recipe 72: PostgreSQL role management and authorization*

SSL/TLS authentication

*Recipe 73: Setting up SSL authentication in PostgreSQL*

Exploring TDE and encryption in PostgreSQL

*Recipe 74: Encryption with pgcrypto in PostgreSQL*

PostgreSQL auditing

*Recipe 75: Installing and configuring pgaudit*

*Recipe 76: Using audit log with PostgreSQL trigger*

*Recipe 77: Working with log\_statement*

LDAP authentication

*Recipe 78: Getting ready with LDAP authentication*

Conclusion

## **10. Backup**

Introduction

Structure

Objectives

Introduction to database backup

Exploring SQL dump and file system level backup

*Recipe 79: Working with logical backup*

*Recipe 80: Working with physical backup*

*Recipe 81: Automating backup*

Exploring backup tools

*Recipe 82: Working with pg\_probackup, pgBackRest tool*

*Recipe 83: Installing and configuring pg\_probackup for PostgreSQL*

*Recipe 84: Installing and configuring pgBackRest for PostgreSQL*

*Recipe 85: Installing and configuring Barman*

*Recipe 86: Incremental/differential backup*

*Recipe 87: Working with schema level backup*

*Recipe 88: Monitoring backup*

Improving backup performance

Conclusion

## **11. Recovery**

Introduction

Structure

Objectives

Introduction to database recovery

Planning recovery

*Recipe 89: Recovery from logical backup*

*Recipe 90: Recovery from physical backup*

Understanding crash recovery

Recovering with PITR

*Recipe 91: Point-in-time recovery*

Incremental/differential restore

*Recipe 92: Incremental/differential restore with Barman*

*Recipe 93: Restoring database with Barman*

Comparing dropped versus damaged table recovery

*Recipe 94: Working with tables recovery*

*Recipe 95: Working with schema level restore*

Conclusion

## **12. Monitoring and Diagnosis**

Introduction

Structure

Objectives

Introducing monitoring and diagnosis

System resource monitoring

Database monitoring

Discovering monitoring script

*Recipe 96: Utilizing script-based monitoring*

Prometheus and Grafana

*Recipe 97: Managing Prometheus and Grafana*

*Recipe 98: Kill/terminate hung query*

Explore statistics

*Recipe 99: Working with statistic collector and analyzer*

Monitoring with system view

*Recipe 100: Getting insight into query monitoring*

*Recipe 101: Getting insight to monitor database lock*

*Recipe 102: Getting insight to monitor active session*

*Recipe 103: Working with query plan*

Vacuum and bloat consideration

*Recipe 104: Getting insight into vacuum and bloat*

Explore fsync

*Recipe 105: Getting insight into fsync*

PostgreSQL other monitoring use cases

*Recipe 106: Monitoring cache hit ratio*

*Recipe 107: Monitor long-running query*

Conclusion

## 13. Troubleshooting

Introduction

Structure

Objectives

Transaction log and checkpoint

*Recipe 108: Adapt log file rotation*

*Recipe 109: Tune transaction log and checkpoints*

Benchmarking

*Recipe 110: Benchmark performance with pgbench*

*Recipe 111: Practice with key parameters in PostgreSQL*

Data checksum

*Recipe 112: Working with pg\_checksums*

Resolution steps:

Conclusion

**Index**

# CHAPTER 1

# Up and Running with PostgreSQL 15

## Introduction

PostgreSQL, aka Postgres, is an advanced open-source relational database and world's fastest-growing database management system. Being open-source, PostgreSQL is released under the PostgreSQL License, a liberal open source license, similar to the BSD or MIT licenses. As of the time of writing this book, the most recent major release of PostgreSQL is version 16.

This chapter provides an introduction to PostgreSQL 15 database systems. We begin by discussing the history of PostgreSQL. We then look at the installation, configuration and starting up PostgreSQL server. Finally, we will cover a couple of common configuration options and database client configuration.

## Structure

In this chapter, we will cover the following topics:

- Introduction to PostgreSQL 15

- Exploring the history of PostgreSQL
- Learning about PostgreSQL architecture
- Learning about PostgreSQL database structure
- Installation methodology
- Database server and client environment
- Exploring backward compatibility

## Objectives

This chapter covers a vast range of topic that provides an understanding of database overview and architecture of core concept and how these component work together. This chapter is all about getting up and running with PostgreSQL using basic recipes and step to cover the installation and setup of PostgreSQL on Linux VMs with step-by-step instructions. If you are already well versed with PostgreSQL basics, It is advisable to glance through the chapter quickly before proceeding to the next chapter.

## Introduction to PostgreSQL 15

PostgreSQL is a powerful, flexible, and scalable general-purpose database. As it is highly stable, very low effort is required to maintain this DBMS. It runs on a wide variety of operating systems flavours, including Linux, Unix and windows.

PostgreSQL 15, the latest version released in October 2022, comes with nicer and consistent functionalities. With the recent release the new enhancement is essential reasons to upgrade. Enhancement with PostgreSQL 15 offers, few of them are:

- Addition of **MERGE** command that improves Developer experience.

- support for **LZ4** and **zstd** compression to **write-ahead log (WAL)** files
- Enhancement in logging format: **jsonlog**
- Improved collation support
- Enhancement with logical replication support for using **two-phase commit (2PC)**
- Support inspection on write-ahead log files with SQL interface
- Support for **WAL** compression with **LZ4** and **Zstandard**
- enhance performance of sorts that exceed **work\_mem**

Beside of its new features, PostgreSQL is **ACID (atomicity, consistency, isolation, and durability)** complaint standard and transactional. The database programmed in C and employs a monolithic design, which means that the components are entirely united and work in a systematic manner.

## Exploring the history of PostgreSQL

PostgreSQL version history dates back in mid-1980s, as a follow-up to INGRES. Postgres has undergone several significant changes since that time. Postgres used the POSTQUEL query language until 1994, In mid-1996s the project was renamed to PostgreSQL to reflect its support for SQL.

Following is the timeline of release history for PostgreSQL database with its enhancements or changes. Here, we cover the last three major versions that have not yet achieved the **end Of life (EOL)**, please refer to [Table 1.1](#):

Database Release	Enhancements

PostgreSQL 13	<ul style="list-style-type: none"> <li>• Incremental sorting</li> <li>• Parallelized vacuuming of indexes</li> <li>• Improved performance for queries that use aggregates or partitioned tables</li> </ul>
PostgreSQL 14	<ul style="list-style-type: none"> <li>• Support in place tablespaces</li> <li>• Change in default password encryption to scram-sha-256</li> <li>• additions to remove expired index entries</li> </ul>
PostgreSQL 15	<ul style="list-style-type: none"> <li>• Support for merge command</li> <li>• Log output using JSON Format</li> <li>• Enhancement in in-memory</li> <li>• Support for zstd compression</li> </ul>

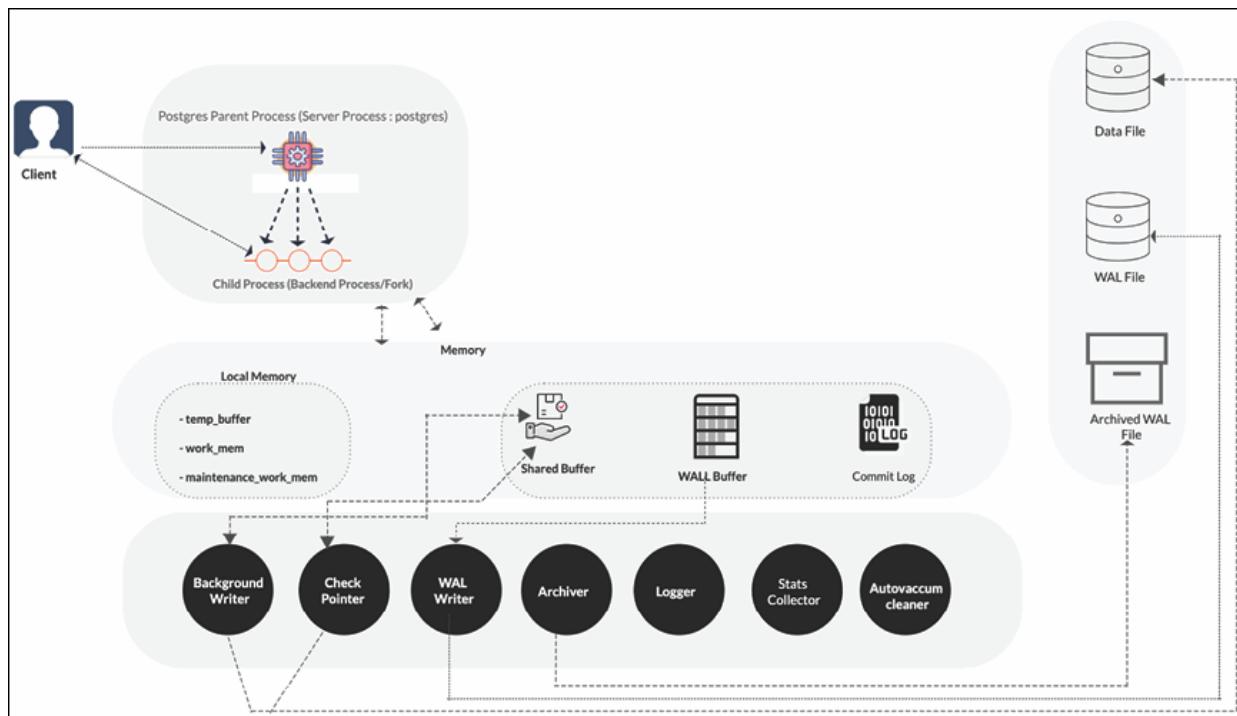
**Table 1.1:** PostgreSQL database release history - enhancement

## Learning about PostgreSQL architecture

The PostgreSQL architecture defines the relationship between various PostgreSQL systems. The basic architecture follows the client-server model, where the key concept is that the client makes a request and initiates the connection. The server responds and provide the service.

This architecture enables a PostgreSQL system to service a wide variety of different clients that can connect locally or over the network. To each successful client, connection spawn a new child process via fork.

PostgreSQL contain many internal components, some are fundamental to PostgreSQL and some are optional. Let us take a look around a typical PostgreSQL internal and external components in the following figure:



**Figure 1.1:** PostgreSQL database cluster architecture

## Learning about PostgreSQL database structure

PostgreSQL Cluster, refers to the collection of database all managing the shared data directory that contain one or more database, that are managed by single server instance. A cluster is generally created for you when you install PostgreSQL database. **initdb** initializes the database cluster with default locale and character set encoding, unless custom setting is specified with the required option.

After the initialization, a newly created instance has **template0**, **template1** and **postgres** as the built-in databases. The database is the **template** upon which the new database is built.

In PostgreSQL, the schema is the logical grouping of database objects or the collection of databases objects are maintained and organized in database. A **SCHEMA** is essentially a namespace, it makes it possible to have a table

with the same name if it belongs to a different schema. Let's get insight to PostgreSQL:

Every database objects has two parts:

**schema\_name.object\_name**

**{prefix}.{suffix}**

When you create a database object, it is always created within a schema. The prefix part logically distinguish your organizational data with other database object.

The syntax for creating a schema:

**# create schema <schema\_name>;**

or

**# create schema IF NOT EXISTS <schema\_name>;**

**Note: IF NOT EXISTS clause is an optional parameter of the CREATE SCHEMA to check whether or not the SCHEMA of the same name already exist in the database before creating it. If it already exist then it will produce an error.**

## Installation methodology

Now that you have an overview of the PostgreSQL database, the next step is to install it. In this section we walk you through the installation methodologies with step-by-step instructions. To offer flexibility, in this book we will perform the installation through binary packages and from the source code.

In this recipe we will cover the installation of PostgreSQL in the following 2 aspects:

- working with installation from binaries
- working with installation from source code

## Recipe 1: Working with installation from binaries

In this recipe, we will cover the installation of PostgreSQL from binary package. Most of the installs of PostgreSQL are on various flavours of Linux and Windows. For this recipe, we will use Red Hat version 8.4 x86\_64 Linux server.

Determine the running Linux version. The following command will give us the information we need:

```
# cat /etc/os-release
NAME="Red Hat Enterprise Linux"
VERSION="8.4 (Ootpa)"
ID="rhel"
ID_LIKE="fedora"
VERSION_ID="8.4"
```

First, assuming that we have downloaded the PostgreSQL binaries from the download site. If not then you can download the latest version of PostgreSQL from their official website at <https://www.postgresql.org/download/> after select your host operating system and version and follow the installation instructions.

When you have finished downloading PostgreSQL, follow these steps:

1. First, download **postgresql15** package from the PostgreSQL official website at [https://download.postgresql.org/pub/repos/yum/15/redhat/rhel-9-x86\\_64/](https://download.postgresql.org/pub/repos/yum/15/redhat/rhel-9-x86_64/) and continue with the installation. We can install this by using the following command from the downloaded binary location:

```
yum localinstall postgresql15-libs-15.0-1PGDG.rhel8.x86_64.rpm
postgresql15-15.0-1PGDG.rhel8.x86_64.rpm postgresql15-server-
15.0-1PGDG.rhel8.x86_64.rpm -y
```

2. Now we are ready to initialize the Postgres after the succeed with the installation steps:

```
# /usr/pgsql-15/bin/postgresql-15-setup initdb
```

3. Look for the following message output for the successful initialization:

```
Initializing database ... OK
```

4. Afterwards, you can start-up the PostgreSQL database by using **systemctl** command-line utility:

```
systemctl start postgresql-15.service
```

5. To verify successful start-up, look for the status from **systemctl** command-line utility:

```
systemctl status postgresql-15.service
```

6. In the final step of this recipe, we will enable the PostgreSQL service. Enabling the service ensures that PostgreSQL will start automatically each time your system boots up:

```
systemctl enable postgresql-15.service
```

At this stage, you are all set up with a database instance. For additional information about executing SQL statements, see recipes 1.5 and 1.6.

## **Recipe 2: Working with installation from source code**

In this recipe, we will cover the installation of PostgreSQL from source code. Major advantage of using source code installation is it can be highly customized during installation. For this recipe, we will use Red Hat version 8.4 x86\_64 Linux server.

First, assuming that we have downloaded the PostgreSQL source code from the download site and extracted it. If not then you can download the latest version of PostgreSQL from their official website at

<https://ftp.postgresql.org/pub/source/v15.0/> after select your host operating system and version and follow the installation instructions.

When you have finished downloading PostgreSQL, follow these steps:

1. Begin the installation by executing the following command from the downloaded location to extract the source code binaries.

```
# tar -xzf postgresql-15.0.tar.gz
```

2. The next step is to create a **postgres** user and set the password.

```
# useradd -d /home/postgres/ postgres
```

3. Create a **postgres** installation and data directory.

```
# Create PG installation directory and a data directory.
```

```
mkdir -p /pg_install/pgv150
```

```
mkdir -p /pg_data/DATA
```

```
# Change Ownership to PG installation directory and a data directory.
```

```
chown -R postgres:postgres /pg_install/pgv150
```

```
chown -R postgres:postgres /pg_data/DATA
```

4. The next step is to **configure** the downloaded source code.

```
# Navigate to extract directory
```

```
cd postgresql-15.0
```

```
# Configure the source code
```

```
./configure --prefix=/pg_install/pgv150/
```

5. Now that you have configured the source code, In next step we will start to build PostgreSQL using **make utility**.

```
# make
```

6. Afterward you can start the PostgreSQL installation using the following command.

```
# make install
```

7. Now initialize the database using the following command from **postgres** user.

```
# su - postgres
```

```
# /pg_install/pgv150/bin/initdb -D /pg_data/DATA/
```

8. Once you have initialized the database, start the database cluster. If you want to change the server's listening port or address, change the **postgresql.conf** file in the database server's data directory.

```
# /pg_install/pgv150/bin/pg_ctl -D /pg_data/DATA/ -l logfile start
```

### Recipe 3: Database configuration file

The configuration file is simply a text file comprised of variables and its associated values, that let you run the way you want by customizing the Postgres database configuration file. The customized setting is independent of the requirement and varies between systems.

In short, the configuration file influences the behaviour of PostgreSQL databases. The default PostgreSQL database configuration file is **postgresql.conf**, location of the configuration file can be set in the initialization steps when setting up the new PostgreSQL instance.

```
# verify the configuration file
```

```
SHOW config_file;
```

We get the following output by executing the previous command:

```
postgres=# SHOW config_file;
      config_file
-----
 /pg_data/postgresql.conf
(1 row)

postgres=#
```

**Figure 1.2:** PostgreSQL database config file path

However, there are other configuration files in addition to the **postgresql.conf** configuration file, which are detailed here:

Configuration File	Description
<b>pg_hba.conf</b>	Sets the server's <b>hba</b> configuration file.
<b>pg_ident.conf</b>	Sets the server's <b>ident</b> configuration file.

**Table 1.2:** PostgreSQL database config file

Show the values of all configuration parameters, with descriptions.

1. # To show the values of all configuration parameter with description
2. SHOW all;

We get the following output, as shown in *Figure 1.3*, by executing the previous command:

**Note: The following output in the screen-shot has been cropped due to its lengthy output.**

name	setting	description
allow_in_place tablespaces	off	Allows tablespaces directly inside pg_tbspc, for testing.
allow_system_table_mods	off	Allows modifications of the structure of system tables.
application_name	pgsql	Sets the application name to be reported in statistics and logs.
archive_cleanup_command	(disabled)	Sets the shell command that will be executed at every restart point.
archive_command	(disabled)	Sets the shell command that will be called to archive a WAL file.
archive_library	(disabled)	Sets the library that will be called to archive a WAL file.
archive_mode	off	Allows archiving of WAL files using archive command.
archive_timeout	0	Sets the amount of time to wait before forcing a switch to the next WAL file.
array_nulls	on	Enable input of NULL elements in arrays.
authentication_timeout	1min	Sets the maximum allowed time to complete client authentication.
autovacuum	on	Starts the autovacuum subprocess.
autovacuum_analyze_scale_factor	0.1	Number of tuple inserts, updates, or deletes prior to analyze as a fraction of reltuples.
autovacuum_analyze_threshold	50	Minimum number of tuple inserts, updates, or deletes prior to analyze.
autovacuum_freeze_max_age	2000000000	Age at which to autovacuum a table to prevent transaction ID wraparound.

**Figure 1.3:** Database configuration parameter

## Database server and client environment

The database server and the client are the two essential components of a client/server architecture. The client-server database is where the database is located on the server, and the client needs additional features such as database access to connect to the database server.

PostgreSQL allows you to configure multiple clients. It is a client tool, which can be used to access local or remote PostgreSQL database. In the end, the client is just a data browser and special program that sends instructions and queries to the server and allow managing data in the databases stored on the host.

The following are the PostgreSQL client application:

Tools	Description
<b>psql</b>	PostgreSQL Interactive Terminal
<b>createdb</b>	create a new PostgreSQL database
<b>clusterdb</b>	cluster a PostgreSQL database
<b>dropdb</b>	remove a PostgreSQL database
<b>createuser</b>	define a new PostgreSQL user account
<b>dropuser</b>	remove a PostgreSQL user account
<b>epcg</b>	embedded SQL C pre-processor

<b>Tools</b>	<b>Description</b>
<b>pg_dump</b>	extract a PostgreSQL database into a script file or other archive file
<b>pg_dumpall</b>	extract a PostgreSQL database cluster into a script file
<b>pg_config</b>	retrieve information about the installed version of PostgreSQL
<b>pgbench</b>	run a benchmark test on PostgreSQL
<b>pg_basebackup</b>	take a base backup of a PostgreSQL cluster
<b>pg_verifybackup</b>	verify the integrity of a base backup of a PostgreSQL cluster
<b>pg_restore</b>	restore a PostgreSQL database from an archive file created by <b>pg_dump</b>
<b>pg_amcheck</b>	checks for corruption in one or more PostgreSQL databases
<b>pg_isready</b>	check the connection status of a PostgreSQL server
<b>pg_recvwal</b>	stream write-ahead logs from a PostgreSQL server
<b>pg_recvlogical</b>	control PostgreSQL logical decoding streams
<b>reindexdb</b>	reindex a PostgreSQL database
<b>vacuumdb</b>	garbage-collect and analyse a PostgreSQL database

**Table 1.3:** PostgreSQL client tool

Of course, the above list of client application provides a command line interface for Postgres. However a lot of **graphical user interface (GUI)** tools available in the market that makes it easier and more convenient for the developer and operations to perform the administrative, designing and functional task.

Thus, we are not listing or comparing the best PostgreSQL GUI tool. But especially because of its value and maybe we need according to our requirement. **pgAdmin** is one of the

widely used open source GUI tool for PostgreSQL. It supports all PostgreSQL operations while being a free and open source. At the time of writing this book, the most recent version release is **pgAdmin 4 v6.5**.

pgAdmin is available on the following platforms:

- Linux
- macOS
- Windows

The following are some highs for pgAdmin:

- Runs as a web application cross-platform solution.
- Rearranging the user interface is easy.
- It comes with integrated SQL editor with shortcuts for more efficient work efficacy.
- Easy to develop or debug your code.
- The dashboard allows you to simplify administrative tasks. Let us you to perform about 80 percent of your administrative tasks.

## **Recipe 4: Installing and configuring pgAdmin client tool**

Download the latest version of pgAdmin from their official website at <https://www.pgadmin.org/download/> and follow the installation instructions. Select the binary, based on your platform, here we have selected the Red Hat RPM repository for installation on the RHEL platform.

1. Setup the repository on Red Hat host.

```
sudo rpm -i  
https://ftp.postgresql.org/pub/pgadmin/pgadmin4/yum/pgadmin4-redhat-  
repo-2-1.noarch.rpm
```

Continue installing pgAdmin with YUM.

```
yum install pgadmin4 -y
```

2. All we need to do now is to configure the system to run in Web mode by executing Web setup script.

```
/usr/pgadmin4/bin/setup-web.sh
```

While executing the **setup-web.sh** script, it prompts for the email ID and password. The executor must respond to the setup program prompt by entering the valid email ID as the username and password to finish configuring the web mode:

```
[root@postgresdev ~]#  
[root@postgresdev ~]# /usr/pgadmin4/bin/setup-web.sh  
Setting up pgAdmin 4 in web mode on a Redhat based platform...  
Creating configuration database...  
NOTE: Configuring authentication for SERVER mode.  
Enter the email address and password to use for the initial pgAdmin user account:  
Email address: user_name@domain.com  
Password: User Defined Password  
Retype password: User Defined Password  
pgAdmin 4 - Application initialisation  
-----  
Creating storage and log directories...  
Configuring SELinux...  
The Apache web server is not running. We can enable and start the web server for you to finish pgAdmin 4 installation. Continue (y/n)? y  
Created symlink /etc/systemd/system/multi-user.target.wants/httpd.service → /usr/lib/systemd/system/httpd.service.  
Apache successfully enabled.  
Apache successfully started.  
You can now start using pgAdmin 4 in web mode at http://127.0.0.1/pgadmin4  
[root@postgresdev ~]#
```

**Figure 1.4:** pgAdmin web setup

3. At this stage, we have pgAdmin installed and configured, and we are now ready to create the PostgreSQL database server connection.

In order to get started with pgAdmin4, open a browser window and enter the following URL in the address bar:

**<http://127.0.0.1/pgadmin4/browser/>**

**Note: Replace the IP with the actual IP address of the installed pgAdmin host.**

A login page will appear where you enter the username and password that was created during the web mode setup stage:



**Figure 1.5:** pgAdmin web UI

4. To connect to the PostgreSQL database, it is first required to add postgres database connection.

The following parameter must be specified:

- Name (Connection name)
- Host name/Address (Database server hostname/IP address)
- Port (Database instance port)
- Maintenance database (Database name)
- Username (Database server user with at least connection privileges)
- Password (Database server user password)

Subsequently, a new entry for postgres database can be added after saving the connection information. In the next recipe, we will cover the database server access in detail.

## Recipe 5: Working with PostgreSQL server access solution

The previous recipe demonstrates the installation and configuration for PostgreSQL server and client. While that also covers the understanding of the client tool and its configurations. In the next recipe, we are proceeding to introduce the database access solution. You have access to PostgreSQL via **psql** or third-party CLI/GUI tools.

The following method provides access to the data base:

- Interactive Terminal-based (**psql**)
- External tool (CLI/GUI)

Let us assume that you have already started the PostgreSQL instance. **psql** enables a default mode of PostgreSQL shell that operate in interactive mode. It is used to query database server data faster and more effectively.

Let us start with querying for PostgreSQL in **psql** shell.

1. Type the following in the Linux command prompt to start the **psql** interactive mode:

```
# start the shell "psql" if you are logged in from a user  
"postgres"  
psql
```

```
# start shell "psql" if logged in from a "root".  
psql --username=postgres --host=localhost --port=5432 --password  
# List out the available user role by executing following  
command  
\du
```

We get the following output by executing the previous command:

```

postgres=# \du
                                         List of roles
Role name |          Attributes          | Member of
-----+-----+-----+
postgres | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
postgres=#

```

**Figure 1.6:** postgres user roles

**Note:** A user is a role with the ability to login. PostgreSQL database cluster manages a set of database user and those users are separate from the operating system user. A default user is created and named **postgres** during the installation of the PostgreSQL database. It is recommended to use a user other than **postgres** for the user data since it is for administrative use.

2. In the following step, we will be accessing the database by creating a new user **test**.

```
# creating user "test".
```

```
CREATE USER test WITH PASSWORD 'Qwerty@123';
```

3. List and verify the user **test**, this gives the user being created.

```
# list all the user in PostgreSQL
```

```
/du
```

```

postgres=# \du
                                         List of roles
Role name |          Attributes          | Member of
-----+-----+-----+
postgres | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
test    |                                         | {}
postgres=#

```

**Figure 1.7:** PostgreSQL test user roles

When a user created in PostgreSQL database has default access to all databases of that instance. The following statement gives the default access privileges of the **test** user or any user created in the PostgreSQL cluster database.

```
# Query default access privileges of new user
```

```
SELECT r.rolname, r.rolsuper, r.rolinherit, r.rolcreaterole,
       r.rolcreatedb, r.rolcanlogin, r.rolconnlimit, r.rolvaliduntil,
       ARRAY(SELECT b.rolname FROM pg_catalog.pg_auth_members m
              JOIN pg_catalog.pg_roles b ON (m.roleid = b.oid) WHERE m.member
              = r.oid) as memberof , r.replication , r.rolbypassrls FROM
       pg_catalog.pg_roles r WHERE r.rolname !~ '^pg_' ORDER BY 1;
```

We get the following output by executing the above command:



	rolname name	rolsuper boolean	rolinherit boolean	rolcreaterole boolean	rolcreatedb boolean	rolcanlogin boolean	rolconnlimit integer	rolvaliduntil timestamp with time zone	memberof name[]	replication boolean	rolbypassrls boolean
1	postgres	true	true	true	true	true	-1	[null]	{}	true	true
2	test	false	true	false	false	true	-1	[null]	{}	false	false

**Figure 1.8: PostgreSQL default access**

As we can see, the **rolname test** has its value true for only column **rolinherit** and **rolcanlogin** by default after the new user **test** created. What we saw in this recipes is the understanding of access solution as an interactive way by using **psql** on local system. In the next recipe, we will see how to access the PostgreSQL database remotely.

## Recipe 6: Working with PostgreSQL remote access

By default, PostgreSQL only accept connections from localhost and restrict remote access. In order to allow the remote IP or host to connect to PostgreSQL instance, lookout for the parameter **listen\_addresses** in the configuration file **postgresql.conf**.

1. Add the following entry in the **postgresql.conf** file:

```
# IP address(es) to listen on  
listen_addresses = '*'
```

2. The next step is to add a client authentication entry to **pg\_hba.conf** (host-based authentication) configuration file.

```
# IP address(es) to listen on  
host all all 0.0.0.0/0 scram-sha-256
```

**Note: scram-sha-256 is the authentication method used here for the password-based authentication.**

3. Afterwards, test the new connection following reboot the PostgreSQL instance for the configuration to take effect.

```
# Restart PostgreSQL Instance
```

```
systemctl restart postgresql-15.service
```

```
# Connect to the database
```

```
psql --username=test --host=localhost --port=5432 --  
dbname=postgres --password
```

We get the following output by executing the previous command:

```
[root@postgresdev log]# psql --username=test --host=localhost --port=5432 --dbname=postgres --password  
Password:  
psql (15.0)  
Type "help" for help.  
postgres=> ■
```

**Figure 1.9:** PostgreSQL remote user login

## Recipe 7: Discovering PostgreSQL database structural object

Database comprise of collection of database object. Each database consist of database objects and one of the most common database structure is a database table(a central object in the relational database structure). Certain objects within PostgreSQL, including database , schema, tables and so on.

In general, the database has the following hierarchical structure:

```
# General database structure
/database
----/schema
----/<Tables>
-----/<Indexes>, <Constraints>, <Triggers>, <Sequences>,..... <Functions>..
```

**Figure 1.10:** Database object structure

Schema:

Let us take a closer look at the database's hierarchical structure:

1. Create a **P15** database from the **psql** shell and assign the **test** user as the owner of the database.

```
# Create database P15
```

```
CREATE DATABASE P15 OWNER test;
```

2. Check the ownership of the **P15** database using the following command:

```
# List the database in PostgreSQL database cluster
```

```
\l
```

We get the following output by executing the previous command that list out the available database:

List of databases								
Name	Owner	Encoding	Collate	Ctype	ICU Locale	Locale Provider	Access privileges	
p15	test	UTF8	en_US.UTF-8	en_US.UTF-8		libc		
postgres	postgres	UTF8	en_US.UTF-8	en_US.UTF-8		libc	=Tc/postgres + postgres=CTc/postgres+	
template0	postgres	UTF8	en_US.UTF-8	en_US.UTF-8		libc	test=c/postgres =c/postgres + postgres=CTc/postgres	
template1	postgres	UTF8	en_US.UTF-8	en_US.UTF-8		libc	=c/postgres + postgres=CTc/postgres	
testing	postgres	UTF8	en_US.UTF-8	en_US.UTF-8		libc	=Tc/postgres + postgres=CTc/postgres test=c/postgres	
(5 rows)								

**Figure 1.11:** Listing p15 database

3. Connect to the **P15** database and create a **BOOKS** schema.

```
# Connect to database P15
psql --username=test --host=192.168.187.128 --port=5432 --dbname=p15
# Create Schema BOOKS
CREATE SCHEMA IF NOT EXISTS BOOKS AUTHORIZATION test;
```

4. Verify the schema by executing following command:

```
# Verify schema on database P15
SELECT * FROM pg_catalog.pg_namespace ORDER BY nspname;
```

We get the following output by executing the above command:

p15=> SELECT * FROM pg_catalog.pg_namespace ORDER BY nspname;			
oid	nspname	nspowner	nspacl
32777	books	24578	
13175	information_schema	10	{postgres=UC/postgres,=U/postgres}
11	pg_catalog	10	{postgres=UC/postgres,=U/postgres}
99	pg_toast	10	
2200	public	6171	{pg_database_owner=UC/pg_database_owner,=U/pg_database_owner}
(5 rows)			

**Figure 1.12:** Listing schema

5. The final step in the recipe is the attempt to create a table **pg\_books15** under schema **BOOKS**.

```
CREATE TABLE books.pg_books15
```

```
s_no int NOT NULL,
```

```
book_id int,  
PRIMARY_KEY (book_id)  
);  
  
Query Query History  
1 CREATE TABLE books.pg_books15 (  
2     s_no int NOT NULL,  
3     book_id int,  
4     PRIMARY KEY (book_id)  
5 );  
6  
Data Output Messages Notifications  
CREATE TABLE  
Query returned successfully in 82 msec.
```

**Figure 1.13:** Create table in book Schema

There are more sections of this recipe in the following chapter, which give detailed information about the database structure and its hierarchy.

## Exploring backward compatibility

PostgreSQL 15 comes with a number of changes that can affect compatibility with the earlier version. While PostgreSQL strives to keep the backward compatibility between some of their releases, but not all enhancement or changes in that release are supported.

Major releases normally change the internal format of system tables and data files. Such changes are often complex, making it impossible to maintain compatibility of all stored data. Backwards compatibility is important to understand prior to upgrading your database to a major version.

Migration or upgrading is a common procedure, in the PostgreSQL database, using **pg\_dumpall** for dump and retrieve to upgrade database is the traditional method.

Alternatively, it is possible to update a database using **`pg_upgrade`** or logical replication.

Let us look at the next recipe to upgrade the PostgreSQL with minor and major release.

Planning to upgrade your database, Following are the few pre-upgrade checklist:

- Meet the installation prerequisite, verify the supported OS requirement with the new release of PostgreSQL.
- Clone the database environment, test the application following database upgrade on the sandbox environment before moving to the development/quality/production environment.

## **Recipe 8: Working with upgrade on minor release**

Upgrading the database to the minor version is a straight forward process. However, minor upgrades only include changes, bugfixes within same release. In this recipe, we will try to cover the steps in detail for upgrading PostgreSQL with minor release:

**Note: Upgrading to the minor release does not require dump or restore method, but the best practice for the production environment is to keep the latest backup. While upgrading to a minor release, the backup step is optional and can bypass that step depending on its environment.**

1. To prepare for a PostgreSQL database backup with **`pg_dumpall`**, the initial step is to restrict database access, preventing any application from connecting during the backup process. You can follow these steps before proceeding with the backup.

Add an entry that allows only local connections, which might look like this:

```
# Allow local connections only
```

```
host all all 127.0.0.1/32 scram-sha-256
```

```
# Restart PostgreSQL service
```

```
systemctl restart postgresql-15.service
```

Make sure to save the changes to **pg\_hba.conf**. This ensures that only local connections can access the database.

2. The next step is to perform a PostgreSQL database cluster backup using **pg\_dumpall**.

```
# Performing database Backup
```

```
pg_dumpall -U postgres -W -f /pg-data/backup/dumpall.sql
```

3. Execute the following query to check the version of the existing database cluster.

```
# Check Version before upgrade
```

```
select version();
```

4. In the next step we stop the PostgreSQL database cluster

```
# Stop PostgreSQL Cluster
```

```
systemctl stop postgresql-14.service
```

5. The next step is to execute the installation from PostgreSQL binaries path, the yum process will update the PostgreSQL version from 14.4 to 14.5.

```
# execute the command from the binary path
```

```
yum localinstall postgresql14-14.5-1PGDG.rhel8.x86_64.rpm postgresql14-libs-14.5-1PGDG.rhel8.x86_64.rpm postgresql14-server-14.5-1PGDG.rhel8.x86_64.rpm
```

6. At this point PostgreSQL database cluster upgraded to minor version 14.5 from 14.4 ,now verify the version following start of the PostgreSQL database cluster.

```
# Start PostgreSQL Cluster
systemctl start postgresql-14.service
# Check Version post upgrade from psql shell
select version();
```

## Recipe 9: Working with upgrade on major release

There are multiple strategies to upgrade the PostgreSQL version. For this recipe, we use the strategy **pg\_upgrade** to upgrade our database cluster. The major advantage of using **pg\_upgrade** method allows you to minimize your downtime.

Assuming that we have downloaded the PostgreSQL binaries from the download site. If not then you can download the latest version of PostgreSQL from their official website at <https://www.postgresql.org/download/> after select your host operating system and version and follow the installation instructions.

As part of the first footstep of the upgrade, you install the binary of the higher database version. The binaries can be installed on the same server or on a different server. In this recipe, we use the approach to install the binary on the same server to update the existing PostgreSQL 14 database cluster to PostgreSQL 15:

1. The first step is to perform a PostgreSQL database cluster backup using **pg\_dumpall**:

```
# Performing database Backup
pg_dumpall -U postgres -W -f /pg-data/backup/dumpall.sql
```

2. Execute the following query to check the version of the existing database cluster:

```
# Check Version before upgrade
select version();
```

The following output screenshot shows our existing major version 14:

```
postgres=# select version();
              version
-----  
PostgreSQL 14.5 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-10), 64-bit  
(1 row)
postgres=#
```

**Figure 1.14:** PostgreSQL instance version information- pre-upgrade

3. The next step is to stop the PostgreSQL database cluster:

```
# Stop PostgreSQL Cluster
systemctl stop postgresql-14.service
```

4. Now we are ready, to begin with, the installation of PostgreSQL 15 by executing the following command from the downloaded binary location:

```
# Execute the command from PostgreSQL installable
binary path
yum localinstall postgresql15-libs-15.0-1PGDG.rhel8.x86_64.rpm
postgresql15-15.0-1PGDG.rhel8.x86_64.rpm postgresql15-server-15.0-
1PGDG.rhel8.x86_64.rpm -y
```

The output of the above command will be similar to the following screenshot:

```
[root@pgdevsrv postgre15]# yum localinstall postgresql15-libs-15.0-1PGDG.rhel8.x86_64.rpm postgresql15-15.0-1PGDG.rhel8.x86_64.rpm postgresql15-server-15.0-1PGDG.rhel8.x86_64.rpm -y
Updating Subscription Management repositories.
Unable to read consumer identity

This system is not registered to Red Hat Subscription Management. You can use subscription-manager to register.

Last metadata expiration check: 0:32:33 ago on Mon 07 Nov 2022 03:04:46 PM +08.
Dependencies resolved.
=====
| Package           | Architecture | Version      | Repository | Size
|=====             |=====         |=====        |=====       |=====
| Installing:      |              |              |            | 
| postgresql15     | x86_64       | 15.0-1PGDG.rhel8 | @CommandLine | 1.6 M
| postgresql15-libs | x86_64       | 15.0-1PGDG.rhel8 | @CommandLine | 297 k
| postgresql15-server | x86_64      | 15.0-1PGDG.rhel8 | @CommandLine | 5.8 M
|=====             |=====         |=====        |=====       |=====

Transaction Summary
=====
| Install 3 Packages
| Total size: 7.7 M
| Installed size: 32 M
| Downloading Packages:
| Running transaction check
| Transaction check succeeded.
| Running transaction test
| Transaction test succeeded.
| Running transaction
| Preparing:          1/1
| Installing:         1/3
|   postgresql15-libs-15.0-1PGDG.rhel8.x86_64
| Running scriptlet:  1/3
|   postgresql15-libs-15.0-1PGDG.rhel8.x86_64
| Installing:         2/3
|   postgresql15-15.0-1PGDG.rhel8.x86_64
| Running scriptlet:  2/3
|   postgresql15-15.0-1PGDG.rhel8.x86_64
| Installing:         3/3
|   postgresql15-server-15.0-1PGDG.rhel8.x86_64
| Running scriptlet:  3/3
|   postgresql15-server-15.0-1PGDG.rhel8.x86_64
| Installing:         3/3
|   postgresql15-server-15.0-1PGDG.rhel8.x86_64
| Running scriptlet:  3/3
|   postgresql15-server-15.0-1PGDG.rhel8.x86_64
| Verifying:          1/3
|   postgresql15-libs-15.0-1PGDG.rhel8.x86_64
| Verifying:          2/3
|   postgresql15-15.0-1PGDG.rhel8.x86_64
| Verifying:          3/3
|   postgresql15-server-15.0-1PGDG.rhel8.x86_64
| Installed products updated.
|=====
| Installed:
|   postgresql15-15.0-1PGDG.rhel8.x86_64
|   postgresql15-libs-15.0-1PGDG.rhel8.x86_64
|   postgresql15-server-15.0-1PGDG.rhel8.x86_64
|=====
| Complete!
[root@pgdevsrv postgre15]#
```

**Figure 1.15:** PostgreSQL installation-YUM

- Now that we have successfully installed PostgreSQL 15, the next step is to initialize the Postgres from the latest installed path.

```
# Initialize database from PostgreSQL 15 installed path
/usr/pgsql-15/bin/postgresql-15-setup initdb
```

- At this point PostgreSQL cluster is initialized and ready for the upgrade, but before that consistency check to be perform for the database to be upgraded using following steps.

```
# Login/switch to the user "postgres"
su - postgres
# Perform the database consistency check
/usr/pgsql-15/bin/pg_upgrade -b /usr/pgsql-14/bin -B /usr/pgsql-15/bin -d
/var/lib/pgsql/14/data -D /var/lib/pgsql/15/data --check
```

We get the following output by executing the above command:

```
[postgres@pgdevserv ~]# /usr/pgsql-15/bin/pg_upgrade -b /usr/pgsql-14/bin -B /usr/pgsql-15/bin -d /var/lib/pgsql/14/data -D /var/lib/pgsql/15/data --check
Performing Consistency Checks
-----
Checking cluster versions ok
Checking database user is the install user ok
Checking database connection settings ok
Checking for prepared transactions ok
Checking for system defined composite types in user tables ok
Checking for reg* data types in user tables ok
Checking for contrib/lsn with bigint-passing mismatch ok
Checking for presence of required libraries ok
Checking database user is the install user ok
Checking for prepared transactions ok
Checking for new cluster tablespace directories ok

*Clusters are compatible*
[postgres@pgdevserv ~]#
```

**Figure 1.16:** PostgreSQL upgrade consistency check

Finally, the database consistency check passed and database cluster is ready for the upgrade:

```
# Login/switch to the user "postgres"
su - postgres
# Perform the database consistency check
/usr/pgsql-15/bin/pg_upgrade -b /usr/pgsql-14/bin -B /usr/pgsql-15/bin -d
/var/lib/pgsql/14/data -D /var/lib/pgsql/15/data -j 10
```

7. Parse the following output after executing the above command, to verify for any error or warnings:

```
[postgres@pgdevsrv ~]$ /usr/pgsql-15/bin/pg_upgrade -b /usr/pgsql-14/bin -B /usr/pgsql-15/bin -d /var/lib/pgsql/14/data -D /var/lib/pgsql/15/data --check
Performing Consistency Checks
-----
Checking cluster versions ok
Checking database user is the install user ok
Checking database connection settings ok
Checking for prepared transactions ok
Checking for system-defined composite types in user tables ok
Checking for reg* data types in user tables ok
Checking for contrib/bsn with bigint-passing mismatch ok
Checking for presence of required libraries ok
Checking database user is the install user ok
Checking for prepared transactions ok
Checking for new cluster tablespace directories ok

*Clusters are compatible*
[postgres@pgdevsrv ~]$ /usr/pgsql-15/bin/pg_upgrade -b /usr/pgsql-14/bin -B /usr/pgsql-15/bin -d /var/lib/pgsql/14/data -D /var/lib/pgsql/15/data -j 10
Performing Consistency Checks
-----
Checking cluster versions ok
Checking database user is the install user ok
Checking database connection settings ok
Checking for prepared transactions ok
Checking for system-defined composite types in user tables ok
Checking for reg* data types in user tables ok
Checking for contrib/bsn with bigint-passing mismatch ok
Creating dump of global objects ok
Creating dump of database schemas ok

Checking for presence of required libraries ok
Checking database user is the install user ok
Checking for prepared transactions ok
Checking for new cluster tablespace directories ok

If pg_upgrade fails after this point, you must re-initdb the
new cluster before continuing.

Performing Upgrade
-----
Analyzing all rows in the new cluster ok
Freezing all rows in the new cluster ok
Deleting files from new pg_xact ok
Copying old pg_xact to new server ok
Setting oldest XID for new cluster ok
Setting next transaction ID and epoch for new cluster ok
Deleting files from new pg_multixact/offsets ok
Copying old pg_multixact/offsets to new server ok
Copying old pg_multixact/offsets to new server ok
Deleting files from new pg_multixact/members ok
Copying old pg_multixact/members to new server ok
Setting next multixact ID and offset for new cluster ok
Resetting WAL archives ok
Setting frozenxid and minmxid counters in new cluster ok
Restoring global objects in the new cluster ok
Restoring database schemas in the new cluster ok

Copying user relation files ok
Setting next OID for new cluster ok
Sync data directory to disk ok
Creating script to delete old cluster ok
Checking for extension updates ok

Upgrade Complete
-----
Optimizer statistics are not transferred by pg_upgrade.
Once you start the new server, consider running:
/usr/pgsql-15/bin/vacuumdb --all --analyze-in-stages

Running this script will delete the old cluster's data files:
./delete_old_cluster.sh
```

**Figure 1.17:** PostgreSQL upgrade logs

- As we have successfully upgraded the database, the next step is to start the PostgreSQL 15 database cluster and verify the version.

```
# Start PostgreSQL Cluster
systemctl start postgresql-15.service
# Check Version post upgrade
select version();
```

The following output shows the PostgreSQL database cluster version 15.0, which confirms that our database has been successfully upgraded.

```
[postgres@pgdevsrv ~]$ psql
psql (15.0)
Type "help" for help.

postgres=# select version();
              version
-----
PostgreSQL 15.0 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-10), 64-bit
(1 row)

postgres=#
```

**Figure 1.18:** PostgreSQL instance version information- post-upgrade

## Conclusion

This chapter introduced PostgreSQL, its history and the different aspects of interaction with the PostgreSQL database cluster. You have learned the PostgreSQL architecture, implementation and database upgrade scenario that helps you decide on the good fit architectural design.

In the following chapter, you will start using the database architectural component and learn details about each feature it provides.

## 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)



# CHAPTER 2

# Database Hierarchy

## Introduction

In this chapter, we will delve into the intricacies of database hierarchy within PostgreSQL, exploring the creation and utilization of databases. As part of this journey, we will also provide you with valuable insights into best practices for access management and authorization, catering to users, groups, and roles. Our exploration will extend to the hierarchical arrangement of database objects within the PostgreSQL database, including essential topics like views, indexes, triggers, functions, and much more. By the end of this chapter, you will have a robust understanding of the structural and operational dimensions of PostgreSQL databases, empowering you to design, manage, and optimize your database systems with confidence.

## Structure

In this chapter, we will cover the following topics:

- Access management
- Database instance and object hierarchy
- Advanced object-relational table concept

- Common table expression
- Discover database storage layout

## Objectives

We will start by demystifying the realm of access management, equipping you with the knowledge to establish stringent access controls while ensuring a seamless user experience. A fundamental understanding of the hierarchical structure of database instances and objects is crucial. We will unravel this hierarchy, providing you with insights into how databases, schemas, and objects interconnect. The power of **common table expressions (CTE)** lies in their ability to simplify complex queries. We will guide through their usage, helping you master this indispensable tool.

## Access management

Access control guarantees that only authorized individuals can obtain the data they are permitted to access. The PostgreSQL access control lets the database administrator access the database securely, using user authentication to ensure granular access to the database.

Database access in PostgreSQL is managed with the concept of role, and the user is nothing more than a role that has the ability and privilege to login by default (A new user created within PostgreSQL has a default login privilege).

In general, Users, groups, and roles are the same within PostgreSQL. PostgreSQL also provides predefined built-in roles that are defined at the database level, and user-defined roles that can be defined for the specific action according to need.

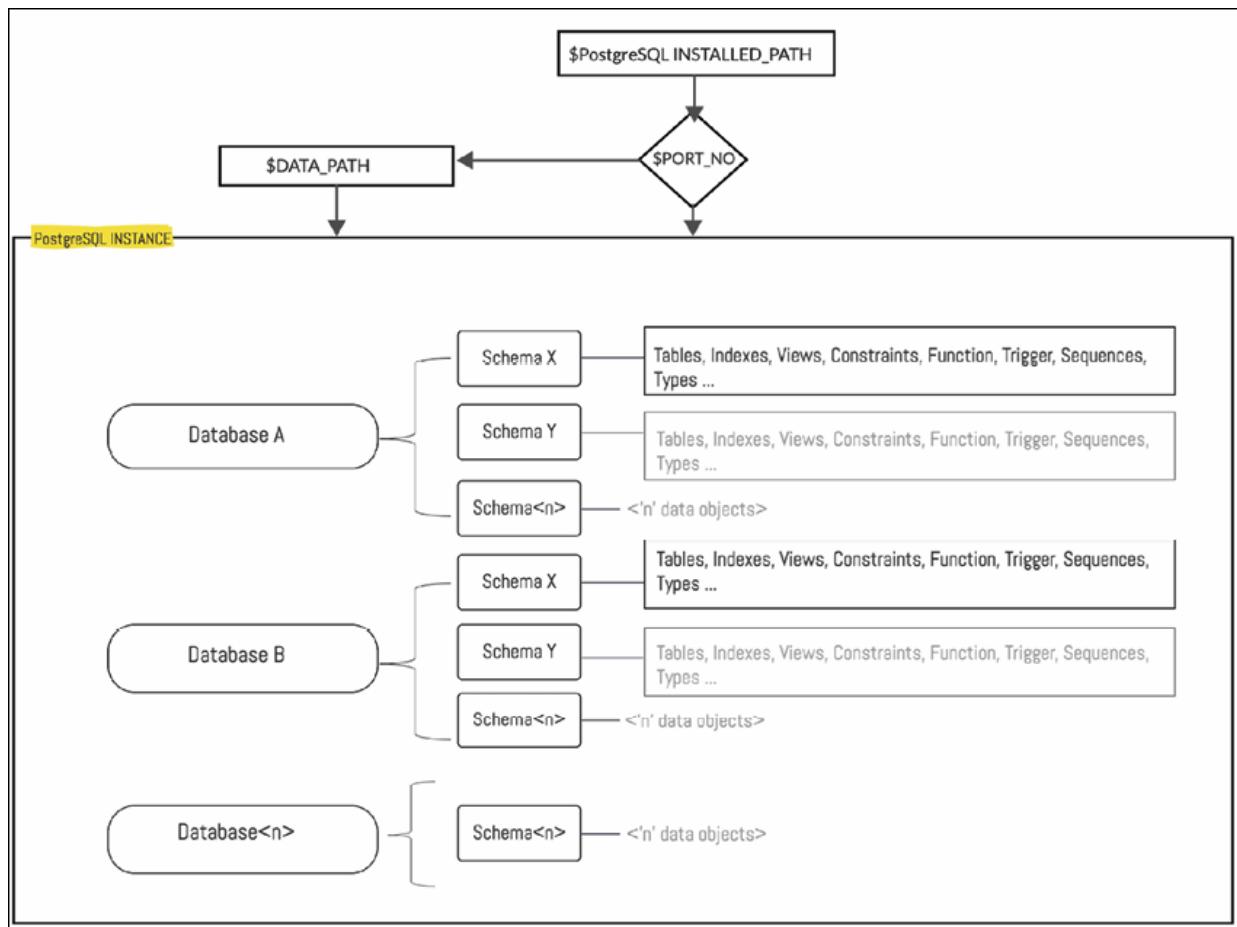
## Database instance and object hierarchy

An instance is a database hierarchy or a set of processes that describe the complete database environment. There can be one or more running instances on a single machine that are isolated with its separate service. PostgreSQL has a hierarchy of database objects, all being shared by the single-running PostgreSQL instance.

The database object hierarchy comprises a collection of database objects. Each database consists of database objects and one of the most common database structures is a database table (a central object in the relational database structure). These database objects are somehow structured and embedded inside the instance.

Firstly, let us focus on the internal components of PostgreSQL.

Beginning with the PostgreSQL instance, as shown in [\*Figure 2.1\*](#), all databases are consolidated into one instance (where the instance is just a set of services, executed on a specified port with its initialized data directory):



**Figure 2.1:** PostgreSQL database object hierarchy

You can start many PostgreSQL instances from the single installed binary, but each instance service maintains its port number and has its data directory.

PostgreSQL listens on to the 5432 default port and the default data directory to **/var/lib/pgsql/<VersionNo>/data/**. However, the PostgreSQL instance's port number and the data directory path can be overridden with the custom port number and data directory path.

Also, we can see within the PostgreSQL instance, a single instance can manage multiple databases simultaneously. Moving further, each of the databases keep an individual set of schemas.

**Schema** allows multiple applications to store data in a single database with no namespace collisions. Schema manages the logical grouping of database objects such as tables, indices, views, triggers, and much more.

Below is the list of database objects maintained in the schema:

**Tables:** Tables are the logical object that stores data. A table comprises a vertical block called a **column** and a horizontal block called a **row**. In addition, each column organizes its own data type that defines the data storage format.

PostgreSQL comes with an inbuilt view called **pg tables**, which contain information about each of the tables available in the database.

**Indexes:** A database index is a pointer, that maps the search key to the corresponding data set on the tables to speed up the data retrieval. The index is used to quicken the search by reducing the number of records to search for.

In PostgreSQL, you can create different types of indexes, upon consideration of which works best for your needs.

Following, in *Table 2.1*, are the types of index PostgreSQL offer:

Index type	Description
B-tree	Default that you get when you do <b>CREATE INDEX</b> .
Hash Index	Hash Indexes are best suited to work with equality operators. Faster lookup than B-tree index.
GiST Index	Generalized Search Tree, this index works best for complex and balanced data such as geometric data and network address data.
SP-GiST Index	Space Partition GiST Index, this index works best for non-balanced data structure using partition search tree.
GIN Index	Generalized Inverted Index, works best for static data. Commonly used for composite column like hstore or array data structures.

BRIN Index	Block Range Index, works best for searching over large time-series data.
------------	--

**Table 2.1:** PostgreSQL index type

## Recipe 10: Adaptation with PostgreSQL Schema

Designing schema is one the fundamental elements in the PostgreSQL database. This provides an isolated environment for managing multiple application datasets. In this recipe, we will create some schema and see how it provides a logical environment that is separate from one another.

For this recipe, we will require a PostgreSQL instance to be up and running. The user invoking the command must have **CREATE** privileges on that database. Refer to the *Working with Installation from Binaries* recipe from [Chapter 1, Up and running with PostgreSQL 15](#) for instructions on how to start the instance.

First start the shell with **psql**, the shell environment will get loaded and then connect to the database on which to create the schema.

```
# Login to the psql shell
psql
# Perform Database Connection
\c postgres
```

Now let us look at creating a new schema, but before that consider an output of **schema\_name** column for the list of pre-defined schema existing in the database Postgres from the following query:

```
# List out the existing schema in database postgres
SELECT schema_name FROM information_schema.schemata;
```

**Note: In PostgreSQL, A new database is initialized with a pre-defined set of schemas and those are:**

- **public:** A default Schema; Tables will get created in public schema by default if the schema detail is not provided during the

### table creation.

- **information\_schema:** Contains information about the database objects defined in the current database
- **pg\_catalog:** Stores the metadata information about the database and cluster
- **pg\_toast:** Hold TOAST storage for large tables.

In the next step let us create a new user-defined schema:

```
# Create schema BOOKS  
create schema BOOKS;
```

Let us take it a bit further and try exploring the range of parameter options available to create a schema, as *Figure 2.2* shows:

# General syntax for creating schema		
CREATE SCHEMA schema_name [ AUTHORIZATION username ] [schema_element [...]]		
# command for creating schema with specification (SCHEMA_NAME, IF NOT EXISTS, AUTHORIZATION)		
CREATE SCHEMA IF NOT EXISTS books AUTHORIZATION test;		
PARAMETER OPTION	SPECIFICATION	DETAIL
schema_name	books	Name of Schema
user_name	test	User name that owns schema
schema_element	CREATE TABLE pg_books15 (s_no int NOT NULL, book_id int, PRIMARY KEY (book_id));	SQL for creating object under Schema
IF NOT EXISTS	books	Notify if the schema with the same name already exist

*Figure 2.2: Schema parameter option*

As shown in the *Figure 2.2*, to create a schema, an input string **books** and **test** was specified against the parameter option **schema\_name** and **user\_name**. Superuser privileges are required to use the **AUTHORIZATION** option. The parameter option **schema\_element** cannot be used with **IF NOT EXISTS**, whereas **schema\_element** provides a step for the usage of the **CREATE TABLE**, **CREATE VIEW**, **CREATE INDEX**, **CREATE SEQUENCE**, **CREATE TRIGGER**, and **GRANT** clause in **CREATE SCHEMA**.

In our previous chapter, [Chapter 1](#), under the recipe *Discovering PostgreSQL Database Structural Objects*, we created a table named **pg\_books15** in the schema named **books**. We know the prefixing of the table with the schema name is required to access the table while accessing the table without explicitly qualifying the schema name ends up with an error. Or, if we create a table without the qualifying schema name, then that table would be created under the **public** schema. This is because the schema for the querying table does not exist in the **search\_path** variable.

You can check the default value for the **search\_path** variable with the following query:

```
# List value for search_path variable
SHOW search_path;
# Following the Output of the above query
search_path;
-----
"$user", public
```

(1 row)

The next step is to change the **search\_path** default configuration to user-defined path. This is done so that we may omit typing the schema name by setting specific schema in the **search\_path** variable using the following query:

```
# Set schema name in the Search_path
SET search_path TO books,public;
# List value for search_path variable
SHOW search_path;
# Following the Output of the above query
search_path;
-----
books, public
```

(1 row)

In the previous step, we configured the **search\_path** schema to **books**. To determine whether the result of a DML

statement uses the **books** or **public** schema, we will create a new table named **Book\_list**. Notably, we would not explicitly prefix the schema name during the table creation (for example, **book\_list**). The table will be created in the **books** schema, as we have set the default **search\_path** to **books** in preceding steps. This behavior can be confirmed using a **DML** statement with **EXPLAIN VERBOSE**, which reveals the execution plan of the statements. (We will go into detail about the **EXPLAIN** option in *Chapter 13, Troubleshooting*). In the following script, we create a table named **book\_list** without specifying the schema name, and then we retrieve the detailed DML execution plan:

```
# Create table book_list
create table book_list (name text);

# Execute a Select statement without schema
name
select * from book_list;

# Following the Output of the above query
name
-----
(0 row)

# Execute a Select statement without schema
by using EXPLAIN VERBOSE
EXPLAIN VERBOSE select * from book_list;

# Following the Output of the above query
QUERY PLAN
-----
Seq Scan on books.book_list  (cost=0.00..23.60 rows=1360
width=32)  Output: name
```

## Advanced object-relational table concept

In PostgreSQL 15, the *Advanced object-relational table concept* includes an intriguing feature known as table inheritance, which builds upon the fundamental principles of relational databases. Inheritance, in this context, allows a database object to inherit both the structure and behaviour from other database objects. Specifically, table inheritance enables the creation of child tables that inherit the column attributes of a parent table, offering a clear and efficient way to model data hierarchies. In simpler terms, the child table incorporates all the columns of the parent table while also accommodating additional columns unique to itself.

The advantages of implementing table inheritance in PostgreSQL are multifaceted. Firstly, it brings notable performance improvements, especially when dealing with data that naturally exhibits hierarchical relationships. This streamlined data modelling enhances query efficiency and reduces the need for complex **JOIN** operations. Secondly, table inheritance minimizes maintenance efforts, as changes to the parent table's schema automatically propagate to the child tables, ensuring data consistency and reducing the risk of errors.

Additionally, this concept can lead to a reduction in index size, as indexing on common attributes shared by parent and child tables can be optimized, resulting in more compact and efficient indexes. This, in turn, contributes to improved query performance.

In conclusion, PostgreSQL 15's advanced object-relational table concept, particularly table inheritance, represents a powerful tool for efficient and hierarchical data modelling. While it offers significant advantages in terms of performance, maintenance, and index size reduction, developers must be mindful of its limitations and carefully assess its suitability for their specific use cases.

## **Recipe 11: Getting insight to tables inheritance**

This recipe is all about the detailed understanding of table inheritance in PostgreSQL database. It demonstrates the use of cases of inheritance, so let us start.

For this recipe, we will be:

- Creating parent table as author under **books** schema.
- Creating child table as **book\_list** under **books** schema

We will start this recipe by creating a parent table with a few columns:

#### # Create Parent Table

```
CREATE TABLE books.author (id integer NOT NULL  
GENERATED ALWAYS AS IDENTITY ( INCREMENT 1 START 1  
MINVALUE 1 MAXVALUE 10000000 CACHE 1 ), author_name text,  
CONSTRAINT id_pk PRIMARY KEY (id) )
```

The next step is to create a child table by referencing the parent table with **INHERITS** clause by executing the following query:

#### # Drop the Table books.book\_list that we created in previous recipe

```
DROP TABLE books.book_list
```

#### # Create Child Table with INHERITS clause

```
CREATE TABLE books.book_list (book_id integer, book_name text  
) INHERITS (books.author);
```

Since the child table **book\_list** inherits from the parent table (author), now **book\_list** table has all the columns defined in the author table. Let us try to access both the parent and child tables.

#### # select the data from Parent Table

```
select * from books.author;
```

#### # Following the Output of the above query

id	author_name
----	-------------

-----	-----
-------	-------

```
(0 rows)
```

```
# select the data from Child Table
```

```
select * from books.book_list;
```

```
# Following the Output of the above query
```

id	author_name	book_id	book_name

```
(0 rows)
```

Let us insert some sample data to **book\_list** table, as we can understand from the following output that, the rows of both the tables populated with data inserted in **book\_list** table. However, the data that populated in the author table belongs to the **book\_list** table.

```
# Insert data to book_list table
```

```
INSERT INTO books.book_list (id, author_name, book_id,  
book_name) VALUES (1, 'TEST', 804, 'PGS');
```

```
INSERT INTO books.book_list (id, author_name, book_id,  
book_name) VALUES (1, 'REST', 805, 'PGS');
```

```
# select the data from book_list Table
```

```
select * from books.book_list;
```

```
# Following the Output of the above query
```

id	author_name	book_id	book_name

```
1 | TEST | 804 | PGS
```

```
1 | REST | 805 | PGS
```

```
(2 rows)
```

```
# select the data from Parent Table
```

```
select * from books.author;
```

```
id | author_name
```

```
1 | TEST
```

```
1 | REST
```

```
(2 rows)
```

Let us insert some sample data to the author table. We can understand from the following output that the inserted rows are only populated in the author table and not in the **book\_list** table.

# Insert data to author table

```
INSERT INTO books.author (author_name) VALUES ('sun'),  
('moon');
```

# select the data from parent Table

```
select * from books.author;
```

# Following the Output of the above query

id	author_name	
1	SUN	
2	MOON	
	TEST	
1	REST	(4 rows)

# select the data from Child Table

```
select * from books.book_list;
```

# Following the Output of the above query

id	author_name	book_id	book_name
1	TEST	804	PGS
1	REST	805	PGS

(2 rows)

The next step is to identify the data that belongs to each table, and this can be done by using the **ONLY** clause. Let us take a look by executing the following query:

# select the data from parent Table using ONLY Clause

```
select * from ONLY books.author;
```

# Following the Output of the above query

```
id | author_name
----+-----
1 | SUN
2 | MOON
(4 rows)
```

As we can see that the first 2 rows belong to the author table, here **ONLY** clause indicates that the query only retrieves the output from the author table.

## Recipe 12: Indexing technique with PostgreSQL database

In this recipe, we will get hands-on with the indexing technique and will cover the benefit and performance improvement with indexing.

In the previous recipe, we had dived into the concept of tables and the use of cases with different types of tables existing in the PostgreSQL database. As we know already, indexes are created on top of the table and the index uses a pointer to access the actual data on the table. Additionally, this recipe is intended to get working on the dataset based on the previous recipe. So, let us begin the journey on the indexing technique.

Let us start by creating a simple index on table **books.book\_list** and list the indexes using the following query:

```
# Create simple index
```

```
create index book_list_idx on books.book_list (book_id);
```

```
# List Index for table books.book_list
```

```
SELECT indexname, indexdef FROM pg_indexes WHERE  
tablename = 'book_list';
```

```
# Following the Output of the above query
```

```
indexname | indexdef
```

```
-----+-----  
book_list_idx | CREATE INDEX book_list_idx ON  
books.book_list USING btree (book_id)  
(1 rows)
```

From the output we can identify that **INDEX** command by default creates a **B-tree** index unless index type is specified using the **USING** clause.

In the next step we will try to distinguish performance improvement between indexed and non-indexed table. To do so, we will create a table named **book\_list\_ni** mirroring the definition and data dataset of **book\_list**, but without creating any indexes on the **book\_list\_ni** table. To do so, we will create a table named **book\_list\_ni** mirroring the definition and data dataset of **book\_list** but without creating any indexes on the **book\_list\_ni** table.

#### **Non-indexed table:**

- a. Create the table **book\_list\_ni** with the same definition and dataset as the **book\_list** table, using the following statement:

```
# Create a table named book_list_ni mirroring the  
structure of the existing book_list table.
```

```
CREATE TABLE books.book_list_ni (book_id integer, book_name  
text ) INHERITS (books.author);
```

```
# Insert dataset on books.book_list_ni from  
books.book_list table
```

```
INSERT INTO books.book_list_ni select * from books.book_list;
```

```
# Verify the dataset from book_list_ni Table
```

```
select * from books.book_list_ni;
```

```
# Following the Output of the above query
```

id	author_name	book_id	book_name
----	-------------	---------	-----------

```
-----+-----+-----+-----
```

1 | REST | 805 | PGS

(2 rows)

- b. Explaining Verbose Execution Plan from non-indexed table by execution the following command:

# Select from non-indexed table

```
EXPLAIN VERBOSE select * from book_list_ni;
```

At Seq Scan **on** book\_list\_ni table, it filters the qualified rows for the output from the available rows.

In *Figure 2.3*, we can see our cost ranges from **0.00..10.10** and we estimate that a total of **810** rows will be produced. The width statistic describes the estimated width for each row in byte. Please refer to the following figure:

```
postgres=# EXPLAIN VERBOSE select * from book_list_ni;
          QUERY PLAN
-----
Seq Scan on books.book_list_ni  (cost=0.00..18.10 rows=810 width=72)
  Output: id, author_name, book_id, book_name
  Query Identifier: -2661352764044314866
(3 rows)
```

**Figure 2.3:** Explain plan for non-indexed table

### Indexed table:

- a. Verify the dataset from **book\_list** table using the following statement:

# Verify the dataset from book\_list Table

```
select * from books.book_list;
```

# Following the Output of the above query

id	author_name	book_id	book_name
----	-------------	---------	-----------

-----+-----+-----+-----

1 | TEST | 804 | PGS

1 | REST | 805 | PGS

(2 rows)

- b. Explaining Verbose Execution Plan from indexed table by execution the following command:

```
# Select from indexed table
```

```
EXPLAIN VERBOSE select * from book_list;
```

At **Seq Scan on book\_book\_list** table, it filters the qualified rows for the output from the available rows.

In *Figure 2.4*, we can see that our cost ranges from **0.00..1.02** and we estimate that a total of 2 rows will be produced. The width statistic describes the estimated width for each row in bytes. Please refer to the following figure:

```
postgres=# EXPLAIN VERBOSE select * from book_list;
          QUERY PLAN
-----
Seq Scan on books.book_list  (cost=0.00..1.02 rows=2 width=72)
  Output: id, author_name, book_id, book_name
  Query Identifier: 7744572311566274039
(3 rows)
```

*Figure 2.4: Explain plan for indexed table*

## Recipe 13: Workaround with concurrent index

In PostgreSQL, the **concurrent index** is the process of creating a new index, without acquiring lock, while transaction is running. In concurrent index build, the index is initially entered into the system catalogs in one transaction. Subsequently, two table scans occur in two additional transactions. Before each table scan, the index build must wait for existing transactions that have modified the table to terminate.

In this recipe, we will work on the concurrent index. The intent of this recipe is to understand the use cases of the concurrent index.

Let us start by creating a table **books.test\_table** and insert some sample datasets in a loop using the following query:

```

# Create a table books.test_table
CREATE TABLE book.test_table (test_id integer , test_name text
);
# Insert data to book_list table
INSERT INTO book.test_table (test_id) select * from
generate_series(1, 10000000);

```

In this step, we simultaneously create a concurrent index on table **book.test\_table** from another session and verify the progress of the index creation.

```

# Create concurrent index on table books.test_table
CREATE INDEX idx1_test_table ON book.test_table (test_id);
# Verify the progress of the create index script
SELECT l.relation::regclass, l.transactionid, l.mode, l.GRANTED,
s.query, s.query_start, age(now(), s.query_start) AS "age", s.pid
FROM pg_stat_activity s JOIN pg_locks l ON l.pid = s.pid WHERE
mode= 'ShareUpdateExclusiveLock' ORDER BY s.query_start;

```

We can see the output of the above script in the following *Figure 2.5:*

relation	mode	query	query_start	age	pid
book.test_table	ShareUpdateExclusiveLock	create index CONCURRENTLY idx1_test_table on book.test_table (test_id);	2022-11-20 20:34:37.925836+08	00:00:08.328216	14729

**Figure 2.5:** Concurrent Index Progress

The script we used to see the progress of the index creation will give a blank output if the index creation query succeeds in creating an index.

## Common table expression

**Common table expression (CTE)**, aka **WITH** clause. It is commonly used to streamline complex queries. The CTE result is not persistent which means that the result set exists only during the execution of the query. **WITH** clause can be

referenced with **SELECT**, **INSERT**, **UPDATE**, or **DELETE** **SQL** statements.

The advantages of the CTE are as follows:

- Improves the maintainability and readability of an SQL query.
- Possible reuse of result set multiple times within the same session, which reduces the computational cost.

## **Recipe 14: Getting insight to SELECT in WITH queries CTE**

In this recipe, we will dive into the world of common table expressions in PostgreSQL 15. We will explore how CTEs can be utilized to gain deeper insights and improve the readability of complex queries. CTEs, often referred to as **WITH** queries, provide a powerful mechanism for breaking down intricate SQL logic into manageable steps, enabling to gain better insights and control over your data manipulation tasks. To illustrate their usage, we will work with a scenario based on e-hailing data, focusing on analyzing driver earnings and passenger ratings in the following steps of this recipe.

### **1. Create a sample data:**

Before we start, let us set up a database by populate it with e-hailing data. Create tables for drivers and trips, and insert sample data:

- a. Create tables for both drivers and trips, and insert sample data.

```
# Create a table for Drivers
```

```
CREATE TABLE drivers (
```

```
    driver_id serial PRIMARY KEY,
```

```
    driver_name VARCHAR,
```

```

    car_make VARCHAR,
    car_model VARCHAR
);
# Create a table for Trips
CREATE TABLE trips (
    trip_id serial PRIMARY KEY,
    driver_id INT,
    passenger_name VARCHAR,
    fare numeric,
    passenger_rating numeric
);
# Insert the sample data to Drivers table
INSERT INTO drivers (driver_name, car_make, car_model)
VALUES
    ('Charlie Brown', 'Toyota', 'Camry'),
    ('Bugs Bunny', 'Honda', 'Accord');

# Insert the sample data to Trips table
INSERT INTO trips (driver_id, passenger_name, fare,
passenger_rating)
VALUES
    (1, 'Bart', 25.50, 4.8),
    (1, 'Felix', 18.75, 4.9),
    (2, 'Jerry', 30.25, 4.7);

```

## 2. Basic SELECT queries:

Let us begin by querying the driver and trip data separately.

- Retrieve the list of drivers and trip details:

```

# Retrieve the list of drivers
SELECT * FROM drivers;
# Execution output of the above SQL to retrieve
drivers list

```

```
driver_id | driver_name | car_make | car_model
```

```
-----+-----+-----+
```

```
1 | Charlie Brown | Toyota | Camry
```

```
2 | Bugs Bunny | Honda | Accord
```

```
(2 rows)
```

*# Retrieve the list of Trips*

```
SELECT * FROM trips;
```

*# Execution output of the above SQL to retrieve trips list*

```
trip_id | driver_id | passenger_name | fare | passenger_rating
```

```
-----+-----+-----+-----+
```

```
1 | 1 | Bart | 25.50 | 4.8
```

```
2 | 1 | Felix | 18.75 | 4.9
```

```
3 | 2 | Jerry | 30.25 | 4.7
```

```
(3 rows)
```

### 3. Introduction to CTE (WITH query):

Now, let us use a CTE to gain insights into driver earnings and ratings.

- Write and execute a query using a CTE to calculate total earnings for each driver:

```
WITH driver_earnings AS (
```

```
    SELECT driver_id, SUM(fare) AS total_earnings
```

```
    FROM trips
```

```
    GROUP BY driver_id
```

```
)
```

```
    SELECT d.driver_name, d.car_make, d.car_model,
```

```
    de.total_earnings
```

```
    FROM drivers d
```

```
    JOIN driver_earnings de ON d.driver_id = de.driver_id;
```

Refer to the *Figure 2.6*, the execution of the above query effectively combines data from both the **drivers** and **trips** tables using the CTE **driver\_earnings** and provides valuable insights into the total earnings of each driver, simplifying the assessment of their financial performance in the e-hailing context.

	driver_name character varying	car_make character varying	car_model character varying	total_earnings numeric
1	Charlie Brown	Toyota	Camry	44.25
2	Bugs Bunny	Honda	Accord	30.25

**Figure 2.6:** CTE- Driver earnings

#### 4. Adding calculations with CTE:

We can use another CTE to calculate average passenger ratings.

- a. Write and execute a query using a CTE to calculate average passenger ratings for each driver:

```

WITH driver_ratings AS (
    SELECT driver_id, AVG(passenger_rating) AS avg_rating
    FROM trips
    GROUP BY driver_id
)
SELECT d.driver_name, d.car_make, d.car_model, dr.avg_rating
FROM drivers d
JOIN driver_ratings dr ON d.driver_id = dr.driver_id;

```

Refer to the *Figure 2.7*, the execution of above query effectively combines data from both the **drivers** and **trips**

tables using the CTE **driver\_ratings** and provides valuable insights into the average ratings of each driver, making it easier to evaluate their performance. Please refer to the following figure:

	driver_name character varying	car_make character varying	car_model character varying	avg_rating numeric
1	Charlie Brown	Toyota	Camry	4.850000000000000
2	Bugs Bunny	Honda	Accord	4.700000000000000

**Figure 2.7: CTE- Driver ratings**

## 5. Joining CTEs with main query:

Combining CTEs with joins allows us to analyze both earnings and ratings in one query.

- Write and execute a query using CTEs to analyze driver earnings and average ratings:

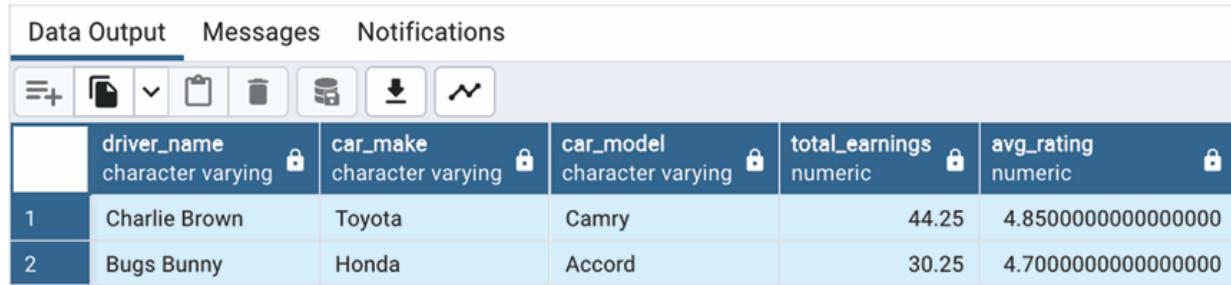
```

WITH driver_earnings AS (
    SELECT driver_id, SUM(fare) AS total_earnings
    FROM trips
    GROUP BY driver_id
),
driver_ratings AS (
    SELECT driver_id, AVG(passenger_rating) AS avg_rating
    FROM trips
    GROUP BY driver_id
)
SELECT d.driver_name, d.car_make, d.car_model, de.total_earnings,
dr.avg_rating
FROM drivers d
JOIN driver_earnings de ON d.driver_id = de.driver_id

```

```
JOIN driver_ratings dr ON d.driver_id = dr.driver_id;
```

Refer to the *Figure 2.8*, this execution of above query masterfully combines data from both the **driver\_earnings** and **driver\_ratings** CTEs, enabling a comprehensive assessment of each driver's financial performance and passenger ratings. It demonstrates the power of CTEs in simplifying complex data analysis tasks by breaking them into manageable steps and combining the results efficiently for meaningful insights. Please refer to the following figure:



The screenshot shows a PostgreSQL query result window. At the top, there are tabs for 'Data Output' (which is selected), 'Messages', and 'Notifications'. Below the tabs is a toolbar with icons for new query, file operations (open, save, etc.), and refresh. The main area is a table with the following data:

	driver_name character varying	car_make character varying	car_model character varying	total_earnings numeric	avg_rating numeric
1	Charlie Brown	Toyota	Camry	44.25	4.8500000000000000
2	Bugs Bunny	Honda	Accord	30.25	4.7000000000000000

**Figure 2.8: CTE- Driver ratings & earnings**

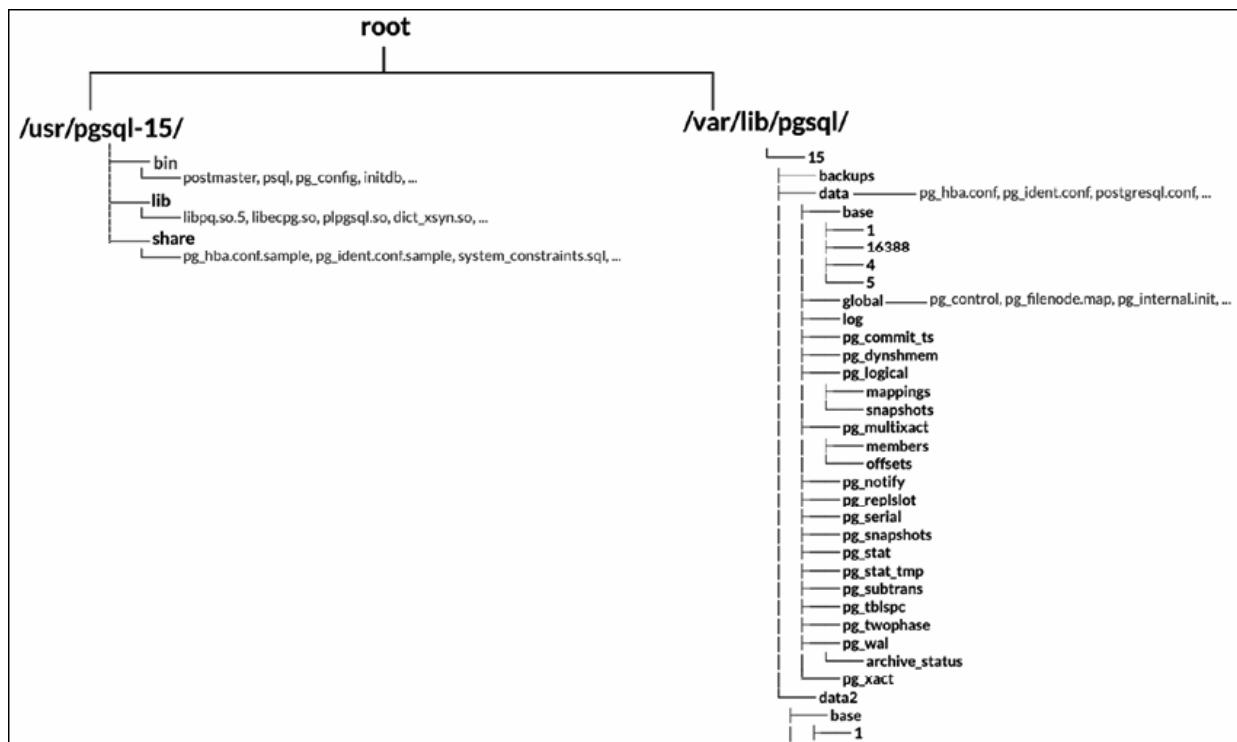
In conclusion, CTEs provide a valuable tool for gaining insights into complex data scenarios. By breaking down intricate queries into manageable steps, you can analyze and combine data from multiple sources effectively. In our e-hailing scenario, we used CTEs to calculate driver earnings and average ratings, providing a clearer picture of driver performance. As you continue exploring PostgreSQL 15, consider incorporating CTEs into your data analysis toolkit to enhance your ability to glean meaningful insights from your data.

## Discover database storage layout

Database storage is divided into **Physical** and **Logical** structure. All information in the database is arranged and organized into database tables but the organization of this data is logical in nature. Although, we must also remember that PostgreSQL does have physical storage later too. Physical structures are those that can be seen and operated

on from the operating system, such as the physical files that store data on disk.

All the data needed for your PostgreSQL database cluster is stored within the cluster's data directory and is controlled by **PGDATA** variable. The common location of the data directory for the **RHEL-based** system is **/var/lib/pgsql/15/data** unless a custom location is specified during the initialization of the database. Please refer to the following figure:



**Figure 2.9: Database storage layout**

In the left section of the [Figure 2.9](#), **directory/usr/pgsql-15/** is the default parent directory for your PostgreSQL executables and libraries. All PostgreSQL command installed in the **bin** directory and its library function are included in the **lib** directory.

The one on the right section is the default data directory path unless a custom location is specified during the

initialization of database. The subdirectory data from **/var/lib/pgsql** is the directory to use for the data storage. The base directory in PostgreSQL is the directory where PostgreSQL stores all the data you have inserted in your databases and global directory is used for storing cluster wide database object.

PostgreSQL has many configuration files that control the database server behaviour. The configuration file can be searched in cluster database directory (PGDATA).

Following is the list of configuration files in *Table 2.2*:

File	Description
<b>postgresql.conf</b>	The main configuration file for PostgreSQL, containing settings that control the behaviour and performance of the database server. It covers a wide range of parameters, including database connection settings, resource allocation, and other server-related configurations.
<b>postgresql.auto.conf</b>	Auto-generated configuration settings managed by the server, typically for dynamic reconfiguration. <b>ALTER SYSTEM</b> writes parameter settings to this file.
<b>pg_hba.conf</b>	Stands for <i>Host-Based Access Control</i> , this file defines client authentication rules for connections to the PostgreSQL server. It specifies which hosts are allowed to connect, which databases they can access, and the authentication methods required.
<b>pg_ident.conf</b>	The PostgreSQL <b>pg_ident.conf</b> file is used to map system user names to database user names. It is particularly useful when integrating PostgreSQL with external authentication systems, allowing the mapping of operating system users to database users based on specific rules.

**Table 2.2:** Configuration file detail

Although, the data path is controlled by **PGDATA** variable but is often changeable. In the next recipe we will see how to adapt with PostgreSQL data directory.

## Recipe 15: Adapt database storage for PostgreSQL database

In this recipe, we will work with database storage adaptation and perform database instance start-up with the new data path.

Before moving the data directory to a new path, first consider the default data directory. We can see the default directory is **/var/lib/pgsql/15/data** by executing the following command from **psql** command prompt as shown in *Figure 2.10*:

```
postgres=# show data_directory;
      data_directory
-----
/var/lib/pgsql/15/data
```

**Figure 2.10:** Default Data Directory-PGDATA

The next step is to set-up a directory for the new data path. To do so, we have created **/pg\_data** as a new data directory path and changed the ownership to the **postgres** user.

*# Create a new directory and change the ownership to postgres*

```
mkdir /pg_data ; chown postgres:postgres /pg_data
```

Now, we shut down the PostgreSQL database cluster service and make sure that the service is not running to ensure the integrity of the datasets.

*# Stop PostgreSQL database cluster service*

```
systemctl stop postgresql-15.service
```

```
# List Index for table book.test_table
```

```
systemctl status postgresql-15.service
```

```
# Following the Output of the above command
```

- postgresql-15. service - PostgreSQL 15 database server

```
Loaded : loaded (/usr/lib/systemd/system/postgresql-15.service);
```

```
    vendor preset: disabled)
```

```
Active:
```

```
inactive (dead)
```

```
Docs: https://www.postgresql.org/docs/15/static/
```

Now that we have executed the shutdown steps in the preceding part, let us proceed to copy the existing database directory to the new location using **rsync** tool.

```
# Copy the data from old data directory to the new  
data directory
```

```
rsync -av /var/lib/pgsql/15/data/ /pg_data
```

Now it is time to update the **postgresql.conf** configuration file with the new configuration of the data path, as described in the following step:

```
# Update data_directory with new path in  
postgresql.conf file
```

```
data_directory = '/pg_data'
```

We have updated the new data path in the **postgresql.conf** configuration file but on systemd-based system, you need to modify **systemd** unit file for postgres. It can be in the **/usr/lib/systemd/system/postgresql-15.service** file or **/usr/lib/systemd/system/** directory. The following is the required update:

```
# Update data path with new path in postgresql  
unit file
```

```
Environment=PGDATA=/pg_data
```

```
#Reload the systemd configuration
```

```
systemctl daemon-reload
```

We finally start the PostgreSQL database cluster service and verify the updated data path with the following script:

```
# Verify the updated data path.  
show data_directory;  
  
# Following the Output of the above command  
data_directory  
-----  
/pg_data
```

After going through the previous section, we now know the data path adaptation in PostgreSQL database cluster. In the next recipe, we will see how to adapt with database log directory.

## Recipe 16: Adapt database log directory

The primary method of monitoring a database is with the help error reporting and logging mechanism of PostgreSQL database. PostgreSQL supports several methods of error reporting but the default is logging **stderr**.

In this recipe, we will cover the logging method available in PostgreSQL and practice with log reporting options.

Let us begin the recipe by creating a PostgreSQL database cluster instance connection with **psql** and run the following command to find the **postgresql.conf** configuration file path setup:

```
# Verify the configuration file location.  
show config_file;  
  
# Following the Output of the above command  
config_file  
-----  
/pg_data/postgresql.conf
```

In the next step we will update the **postgresql.conf** configuration file with the logging parameter, as described in the following step:

```
# Update below logging parameter in
postgresql.conf file

log_destination = 'csvlog'
logging_collector = on
log_directory = '/pg_data/log'
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
```

We, then, restart the PostgreSQL database cluster service and verify the updated logging configuration with the following script:

```
# Verify the logging configuration
select name,setting from pg_settings where name IN
('log_destination','logging_collector','log_directory','log_filename');
```

*# Following the Output of the above command*

name	setting
log_destination	csvlog
log_directory	/pg_data/log
	postgresql-%Y-%m-%d_%H%M%S.log.
logging_collector	on

Here we set up PostgreSQL to generate the log in CSV format and place it in the **/pg\_data/log** directory. The **log\_directory** parameter value may either be an absolute or a relative path, but we provide the absolute path for this recipe. For **log\_filename** parameter values, we defined the prefix as **postres** with the suffix as the timestamp for the log file name.

## Recipe 17: Workaround with PostgreSQL TOAST

In PostgreSQL data is organized in the form of page of size 8KB. The database will allocate space in a table in some given block size. A page size of 8KB is used for storing tuples, indexes, and queries execution plan.

In PostgreSQL, a row or tuple cannot extend across multiple pages, but there are no restrictions on the size of individual database rows. In order to address this constraint, PostgreSQL employs **The Oversized-Attribute Storage Technique (TOAST)** mechanism. This approach ensures that a tuple does not surpass the size of the default page size by storing oversized attributes separately. Therefore, the block size serves as an absolute maximum limit for row size.

TOAST is the mechanism, which is mainly used to store the value of the large field.

In this recipe, we will work around with TOAST mechanism and understand the operational use cases.

Let us begin by creating a table **book.author\_list** with a large field using following query

```
# Create table book.author_list
    create table book.author_list(auth_id int, author_name text,
book_list text);
# list the relational information for table
book.author_list
\d+ book.author_list
```

```
# Following the Output of the above command
```

If we notice at the **Column, Storage** in the [Figure 2.11](#), the TOAST policy applied to each of its field **auth\_id**, **author\_name** and **book\_list** by default.

Table "book.author_list"								
Column	Type	Collation	Nullable	Default	Storage	Compression	Stats target	Description
auth_id	integer				plain			
author_name	text				extended			
book_list	text				extended			
Access method: heap								

**Figure 2.11:** Table information for TOAST

**TOAST** table also get created for out-of-line storage automatically. Let us dig for the **TOAST** table in the following step:

```
# Query the pg_class catalog table to list the TOAST table
```

```
select relname, relfilenode, reltoastrelid from pg_class where  
relname = 'author_list';
```

```
# Following the Output of the above command
```

relname	relfilenode	reltoastrelid
author_list	24581	24584

The **relfilenode** column in the preceding output, which indicates the **OID** for the **author\_list** table is **24581** and the corresponding **TOAST** table is **pg\_toast\_24581**. Let us get a closer look of the **TOAST** table **pg\_toast\_24581** relational information using the following query:

```
# list the relational information for table  
pg_toast_24581
```

```
\d+ pg_toast.pg_toast_24581;
```

```
# Following the Output of the above command
```

Column	Type	Storage
chunk_id	oid	plain
chunk_seq	integer	plain
chunk_data	bytea	plain

Owning table: "book.author\_list"

Indexes:

```
"pg_toast_24581_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
```

Access method: heap

We now insert sample data to **book.author\_list** table and subsequently verify the data from **TOAST** table **pg\_toast\_24581**.

```
# Insert sample data to book.author_list table
insert into book.author_list values(842, 'tester', 'First Book');
# Verify inserted data from toast table
pg_toast_24581
```

```
select * from pg_toast.pg_toast_24581;
chunk_id | chunk_seq | chunk_data
-----+-----+
(0 rows)
```

*# Verify the length of the dataset*

```
select auth_id,author_name,length(book_list) from book.author_list;
```

*# Following the output of the above command*

```
auth_id | author_name | length
-----+-----+
842    | tester      | 10
```

As the data length in the **book\_list** column is currently limited to 10 characters, which is insufficient, it needs to be expanded to accommodate more than 2 KB. To verify the length of the inserted dataset field, we will populate the **book\_list** column with data exceeding 2 KB in length, continuing until a new row is generated in the TOAST table. Please execute the following script.

```
# Verify the length of the dataset for column
book_list
```

```
select auth_id, author_name, length(book_list) from
book.author_list;
```

*# Following the Output of above command*

```
auth_id | author_name | length
-----+-----+
842    | tester      | 3961
```

*# Verify the data in the TOAST table*

```
select chunk_id, chunk_seq, length(chunk_data) from
```

```
pg_toast.pg_toast_24581
```

```
# Following the Output of above command
```

chunk_id	chunk_seq	length
24593	0	1996
24593	1	1965

Eventually, we can see that the TOAST table **pg\_toast.pg\_toast\_24581** was generated with the dataset for the row more than the **book.author\_list** table.

## Recipe 18: Parse the database admin specific start-up logs

Parsing database admin-specific start-up logs is a crucial task for database administrators responsible for maintaining PostgreSQL 15 databases. These logs contain valuable information related to the database's initialization, authentication, and access control, which can be essential for troubleshooting performance issues and ensuring the database's security.

In this recipe, we will guide you through the process of parsing the start-up logs for PostgreSQL 15, focusing on database admin-specific information. Parsing these logs can be useful for monitoring and troubleshooting database performance issues. We will use Linux as the operating system for this example:

### 1. Access the log files:

- a. Open a terminal window on PostgreSQL Linux machine and navigate to the directory containing PostgreSQL log files. By default, PostgreSQL stores log directory and files within the PGDATA directory:

```
# Verify the log directory and current log file
```

```
SELECT pg_current_logfile();
# Execution output of the above SQL
```

```
pg_current_logfile
```

```
-----  
log/postgresql-Sat.csv
```

```
(1 row)
```

The output **log/postgresql-Sat.csv** tells you that the current log file for PostgreSQL on the day we ran this command is named **postgresql-Sat.csv**, and it is located within the **log** directory of your PostgreSQL data directory.

## 2. View log files:

- Use the **cat** command to view the **log** files. PostgreSQL **log** filenames typically include a date and time stamp.

```
# Verify the Log entries
```

```
cat /pg_data/log/postgresql-Sat.csv
```

Observe the log entries. The log file will contain various entries related to database activities, including start-up information.

## 3. Parse database admin specific start-up logs:

- To focus on database admin-specific start-up logs, we can use grep to filter the log entries. For example, to find log entries related to authentication and access control, we can use:

```
# Parse the Log entries related to authentication
```

```
[postgres@pgsrvdev log]$ cat postgresql-Sat.csv |grep -i  
authentication
```

2023	-07-01	20:36:40.648
+08,"postgres_exporter","postgres",12276,"192.168.187.133:41640",64a01		
dd8.2ff4.1,"authentication",2023-07-01		20:36:40

```
+08,6/429,0,FATAL,28P01,"password authentication failed for user  
""postgres_exporter""","Role ""postgres_exporter"" does not exist.
```

2023-07-01

20:36:44.703

```
+08,"postgres_exporter","postgres",12279,"192.168.187.133:41114",  
64a01ddc.2ff7,1,"authentication",2023-07-01 20:36:44  
+08,6/430,0,FATAL,28P01,"password authentication failed for user  
""postgres_exporter""","Role ""postgres_exporter"" does not exist.
```

Review the output to identify relevant entries. Note the timestamps and any error messages or warnings.

#### 4. Analyze the logs:

- Examine the relevant log entries to identify potential issues or patterns. Look for error messages, authentication failures, or unusual start-up behaviour.

#### 5. Tail logs in real-time (Optional):

- To monitor logs in real-time as they are generated, we can use the **tail** command with the **-f** flag:

```
# Parse the realtime Log entries
```

```
tail -f postgresql-Sat.csv
```

**Note: It is important to handle log data securely, as logs may contain sensitive information. Ensure proper access controls and privacy measures are in place when working with log files.**

## Recipe 19: Getting insight to PostgreSQL memory configuration

PostgreSQL memory manager is grouped into local and shared memory area. Memory might be allocated to the particular memory area when a specific event occurs, or it

might be reallocated in response to the configuration change.

## Local memory area

All the backend process of PostgreSQL allocates memory from the local memory area for query processing. This local memory area has further sub-areas as follows:

**work\_mem**: This allows one to use the amount of memory used by internal sorting operations and hash tables before writing to temporary disk files. Specifying the appropriate value of the **work\_mem** parameter can result in less disk swap, and consequently much faster query processing.

By default, every **postgres** activity uses up to 4 MB, but for complex queries, 4 MB is not sufficient. As a result, the database uses a disc-based sort instead which slow down performance.

Therefore, a database environment where regular data sorting occurs, increased **work\_mem** might be one of the most effective ways to accelerate your server. The maximum configurable value for **work\_mem** is approximately 2 TB.

```
# Check maximum configurable memory for  
work_mem
```

```
SELECT unit,max_val FROM pg_settings WHERE  
name='work_mem';
```

```
# Following the Output of above command
```

unit	max_val
kb	2147483647

The formula for calculating the optimal value for **work\_mem** is straightforward as follows.

```
Total RAM * 0.25 / max_connections = work_mem
```

**max\_connection:** This determines the maximum number of concurrent connections to the database server. The default value for this parameter is 100 connections.

**maintenance\_work\_mem:** This allows you to use the amount of memory taken up by maintenance operations, such as **VAC UUM**, **CREATE INDEX**, and **ADD FOREIGN KEY**.

The memory area allocated per session for any of the aforementioned operations is 64 MB by default. It is recommended to set this value significantly larger than **work\_mem**. More extensive adjustments could improve the vacuum and restoration of database dump. The maximum configurable value for **maintenance\_work\_mem** is approximately 2 TB which can be checked by the following script:

```
# Check maximum configurable memory for  
maintenance_work_mem
```

```
SELECT unit,max_val FROM pg_settings WHERE  
name='maintenance_work_mem';
```

```
# Following the Output of above command
```

unit	max_val
kb	2147483647

**temp\_buffer:** **temp\_buffer** is a memory area, it is used as cache for temporary tables. It is assigned at the start of each session and remains in use for the duration of each database session. A session will allocate temporary buffers as needed till the limit given by **temp\_buffer** is reached.

## Shared memory area

All the background process of PostgreSQL allocates memory from the shared memory area for query processing. This shared memory area has further sub-areas as follows:

**shared\_buffers**: The **shared\_buffers** configuration parameter determines how much memory is dedicated to PostgreSQL for caching data. The default size of **shared\_buffers** is 128 MB; however, it is recommended that the initial size of **shared\_buffers** should be 25% of the available system memory. Setting the **shared\_buffers** is heavily depends on the data set and environment. Though, the right balance of the **shared\_buffers** can be achieved through benchmarking with multiple data sets. The following script shows the **shared\_buffers** configured value:

```
# Check the shared_buffers configured value
```

```
SHOW shared_buffers;
```

```
# Following the Output of above command
```

```
Shared_buffers
```

```
-----
```

```
128MB
```

As PostgreSQL uses double buffering, that means PostgreSQL uses its own buffer and also uses kernel buffered IO.

**WAL buffers**: **WAL buffers** refers to the transaction log buffers. It contains the history of all changes made to the database. With every change made to the database, **postgres** writes the changes to the WAL buffers and these buffers are flushed to the WAL file when a commit is issued. Temporary tables are not WAL-logged.

## Conclusion

As we conclude our journey, we have acquired a wealth of knowledge about the intricacies of database hierarchy. We embarked on a comprehensive exploration of access management, empowering you to control database interactions effectively. The hierarchical arrangement of database instances and objects is now a familiar terrain, enabling you to create and manage databases with

confidence. By delving into advanced object-relational concepts, you are better equipped to design intricate database structures.

Furthermore, the practical application of CTE has been demystified, allowing you to simplify complex queries and enhance your data manipulation capabilities. Additionally, our insights into database storage layout strategies have positioned you to optimize performance through informed decisions.

In the next chapter, we will focus on understanding how to use PostgreSQL in the cloud. We will look at the benefits and things to think about when using cloud solutions for databases like PostgreSQL. This is like a big change in the way we do things. We will explore why this change is good and what we need to be careful about.

## **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

# Cloud Provisioning

## Introduction

In today's rapidly evolving technological landscape, the cloud has emerged as a cornerstone of modern infrastructure deployment. This chapter delves into the dynamic realm of cloud provisioning for PostgreSQL 15, where we explore innovative implementation scenarios within the AWS ecosystem. Our focus is to unravel the diverse array of options available for managing PostgreSQL instances within the cloud environment. By scrutinizing each method in detail, we aim to equip you with comprehensive insights and best practices, enabling you to adeptly navigate the challenges and harness the benefits of PostgreSQL on the cloud.

## Structure

In this chapter, we will cover the following topics:

- Introduction to cloud solution for PostgreSQL
- Managed or self-managed options
- Exploring EC2 and RDS instance

## Objectives

In this chapter we begin by immersing ourselves in the world of cloud solutions for PostgreSQL, unravelling the inherent advantages and considerations. By grasping the foundations of cloud provisioning, you will be poised to make informed decisions in the complex landscape of PostgreSQL deployment. Delving into the intricacies of managed and self-managed options, we dissect the pros and cons of each approach.

This comparative analysis empowers you to select the most suitable deployment model for your PostgreSQL instance, factoring in factors such as scalability, maintenance, and control.

Our exploration takes us into the heart of AWS's offerings – the **Elastic Compute Cloud (EC2)** and the **Relational Database Service (RDS)** instances. Through comprehensive case studies and practical examples, we unravel the unique features, management considerations, and performance implications of both instances.

## Introduction to cloud solution for PostgreSQL

The concept of a *Cloud solution for PostgreSQL 15* marks a pivotal shift in how we approach database management. It involves leveraging the power of cloud computing, particularly platforms like **Amazon Web Services (AWS)**, to host and manage PostgreSQL 15 databases. This approach offers numerous advantages, including scalability, flexibility, reduced operational overhead, and improved accessibility. Within this landscape, PostgreSQL 15 becomes the cornerstone, enabling the creation of robust, scalable, and highly available database deployments. Key components include virtual machines called instances, categorized into types like EC2 and RDS, each offering distinct features and management options. Exploring this chapter, you will gain insights into the foundational elements and step-by-step processes involved in deploying PostgreSQL 15 in the cloud, setting the stage for a database strategy aligned with the dynamic capabilities of modern cloud environments.

## Cloud service model

AWS offers many different cloud service models to run the PostgreSQL database in the cloud. These include the following:

**Infrastructure as a Service (IaaS):** It offers resources like servers, networks, operating systems, and data storage to be delivered virtually without purchasing hardware outright.

**Platform as a Service (PaaS):** Like IaaS, PaaS provides infrastructure such as servers, networks, operating systems and data storage, but also included with managed services and software that provide developer-ready environment.

**Software as a Service (SaaS):** This service model provides a licensing and delivery model that enables a wide selection of software or application that delivers to the end-users through an internet browser.

## **Cloud deployment option**

Cloud deployment model is based on the infrastructure resources at consumer location (on-premise), cloud provider location, or both.

PostgreSQL is available on-premise as well in the cloud.

Let us take a moment to review each deployment option:

- On-premise deployment option refers to all infrastructure resources such as software, hardware, network installed and managed by the consumer.
- Cloud deployment option refers to the cloud service that owns the infrastructure resources, meaning that the consumer does not have to be concerned with managing the infrastructure.
- Hybrid deployment option is a combination of on-premises and cloud services. One of the primary benefits of the hybrid cloud infrastructure is its flexibility and security from which organizations can choose based on their business model, data sovereignty demands, and regulatory requirements.

## **Managed or self-managed options**

Business models and needs drive the selection of the cloud service option. The self-managed option, consists of managing the software, services, installation and configuration done by the consumer on the cloud provider infrastructure.

In managed option, managing only the core business service by the consumer, setup of the services and infrastructure to be made out by the cloud service provider. By leveraging the benefits of a managed service option, the customer can expand critical business services without the distraction of infrastructure management.

AWS offer following services for the PostgreSQL database:

- Amazon Elastic Compute Cloud
- Amazon Relational Database Service

## **Exploring EC2 and RDS instance**

Exploring EC2 and RDS instances for PostgreSQL 15 delves into the powerful capabilities of AWS that enhance the deployment and management of PostgreSQL databases. With Amazon EC2, users gain access to flexible and resizable compute resources, empowering them to tailor computing solutions according to precise requirements. This

self-managed, on-demand service not only optimizes resource utilization but also streamlines cost efficiency by enabling pay-as-you-go usage. Complementing EC2, Amazon RDS offers a fully managed cloud database solution, eliminating complexities in database administration. As an open-source option, RDS simplifies the launch, configuration, operation, and scalability of PostgreSQL databases. Together, these AWS offerings provide PostgreSQL 15 users with a dynamic environment for seamless, scalable, and cost-effective database deployment and management.

## **Recipe 20: Manage PostgreSQL instance with AWS EC2**

Having learned about the PostgreSQL database installation, configuration and architecture in an on-premise environment. This recipe will walk us through configuring a PostgreSQL instance on AWS EC2. The prerequisite and steps required in configuring are explained here.

First, assuming that we have signed up for an AWS account. If not then you can sign up from the website at <https://aws.amazon.com/> and follow the instructions.

When you have completed signing up process, log in to the AWS account and follow these steps:

1. At First, select the **Root user** at the AWS account login prompt and click **Next** to continue the login process. **Root user** credential provides full access to all resources, whereas **IAM** credential can secure access to AWS services and resources for users of your AWS account. Refer to the following figure:



## Sign in

**Root user**

Account owner that performs tasks requiring unrestricted access. [Learn more](#)

**IAM user**

User within an account that performs daily tasks. [Learn more](#)

Root user email address

[REDACTED] com

**Next**

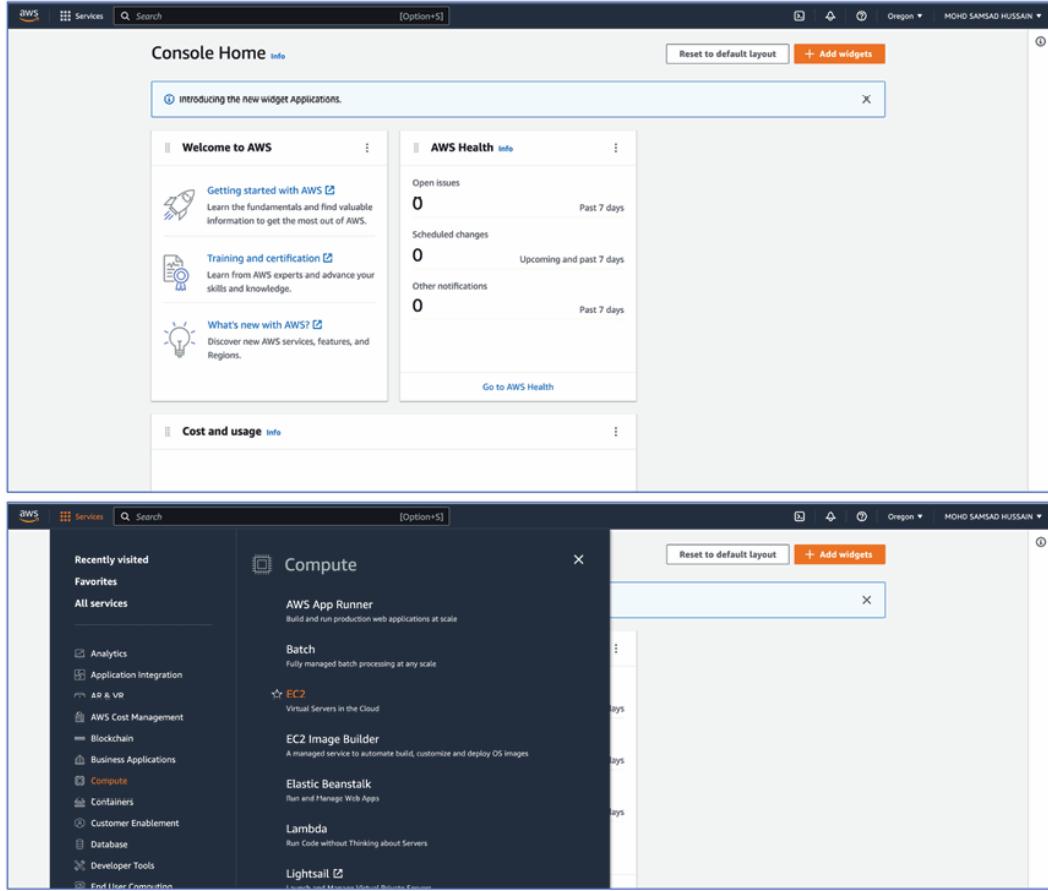
By continuing, you agree to the [AWS Customer Agreement](#) or other agreement for AWS services, and the [Privacy Notice](#). This site uses essential cookies. See our [Cookie Notice](#) for more information.

— New to AWS? —

[Create a new AWS account](#)

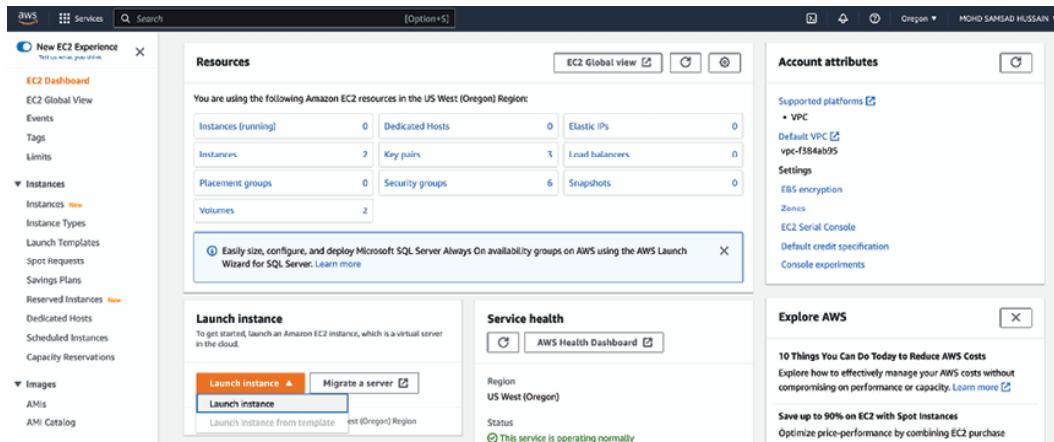
*Figure 3.1: AWS account login option*

2. Now we are in the AWS **Console Home** page. Refer to *Figure 3.2*, click the **Services** tab listing all the services. All we have to do is select the **EC2** in the **Compute** service category, which will take us to the **EC2** dashboard:



**Figure 3.2:** AWS Console Home page and Compute service category

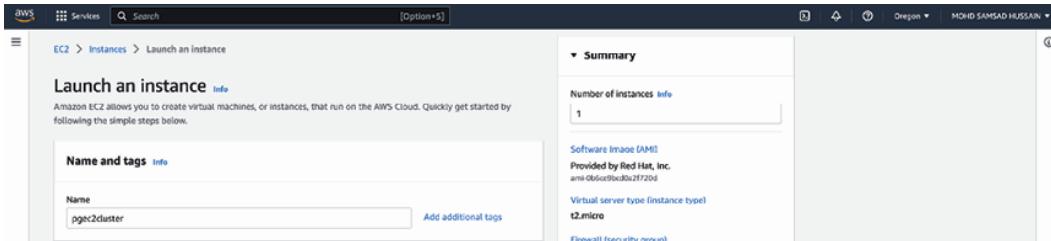
3. From the **EC2 Dashboard**, in the **Launch instance** area, choose **Launch instance** and then select **Launch instance** from the available options as shown in *Figure 3.3*:



**Figure 3.3:** AWS EC2 Dashboard

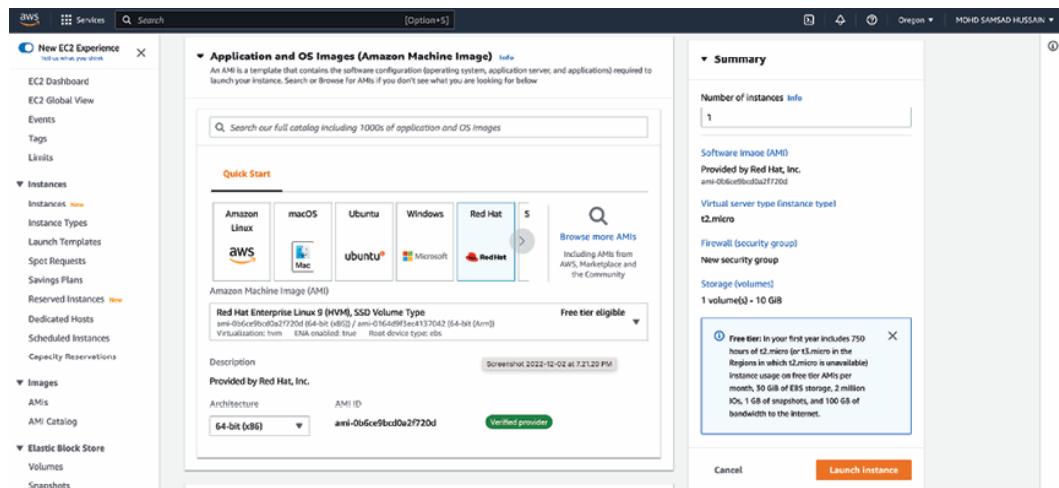
4. Here **Launch an instance** dashboard allows you to input the following and click **Launch instance**:

- a. Refer to *Figure 3.4*, input the name of your instance under the **Name and tags**. For this recipe we use the name **pgec2cluster**:



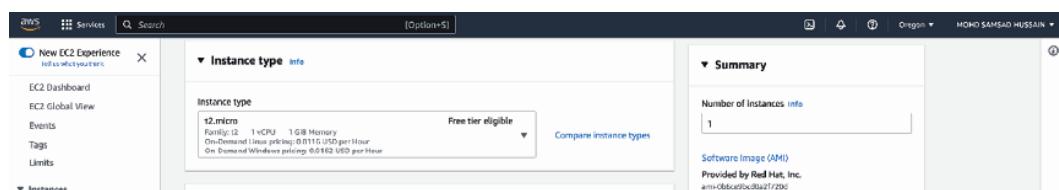
**Figure 3.4:** AWS Launch instance page

- b. From the **Amazon Machine Image**, select **Free tier eligible Red Hat Enterprise Linux 9 (HVM)** as shown in *Figure 3.5*:



**Figure 3.5:** AWS AMI list for EC2

- c. Select **Free tier eligible t2.micro** as the **Instance type** as shown in *Figure 3.6*:



**Figure 3.6:** AWS Instance type for EC2

- d. Refer to *Figure 3.7*, the next option is to create a new key

pair under the **Key pair (login)** using the following steps:

- i. Click **Create new key pair** under the **Key pair (login)**, a pop-up will appear to enter the key pair details and then click **Create key pair**. For this recipe, we provided the following detail to create a key pair:

Key pair name: **con\_pg**

Key pair type: **RSA**

Private key file format: **pem**

- ii. A file named **con\_pg.pem** will be downloaded to your local machine by clicking **Create key pair** and this file would be needed for subsequent access to the AWS machine.

**Create key pair** Number of instances: Info X

Key pairs allow you to connect to your instance securely.

Enter the name of the key pair below. When prompted, store the private key in a secure and accessible location on your computer. **You will need it later to connect to your instance.** [Learn more](#) 

**Key pair name**

The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.

**Key pair type**

**RSA**  
RSA encrypted private and public key pair

**ED25519**  
ED25519 encrypted private and public key pair (Not supported for Windows instances)

**Private key file format**

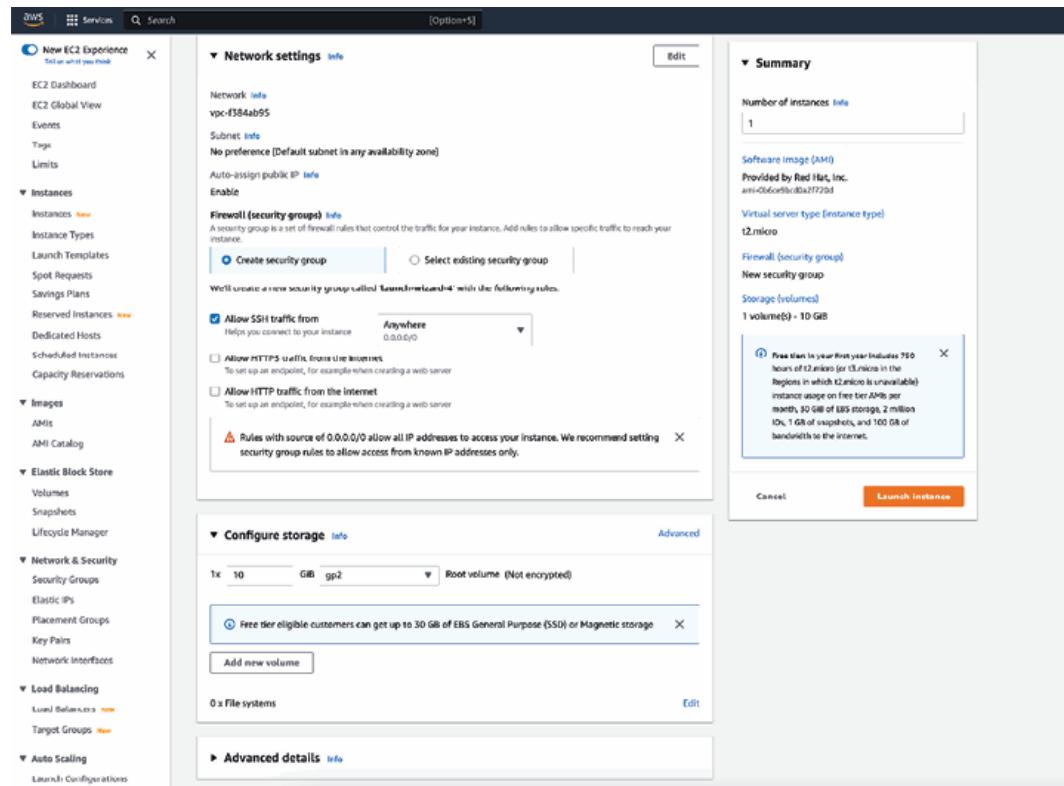
**.pem**  
For use with OpenSSH

**.ppk**  
For use with PuTTY

**Cancel** **Create key pair**

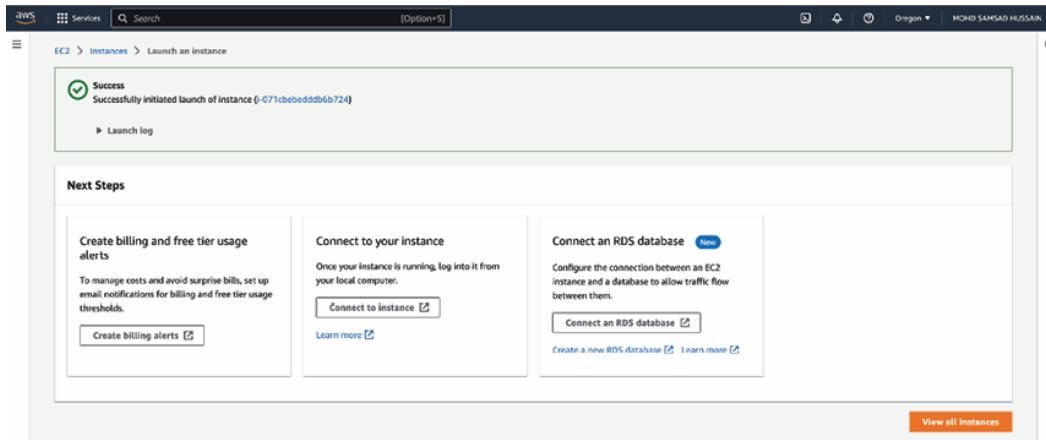
**Figure 3.7:** AWS Instance - Create key pair

- e. At this point, we keep the **Network settings**, **Configure Storage** and **Advanced detail** options as default and ready to create our EC2 instance named **pgec2cluster** by clicking on the **Launch instance** tab as showing in *Figure 3.8*:



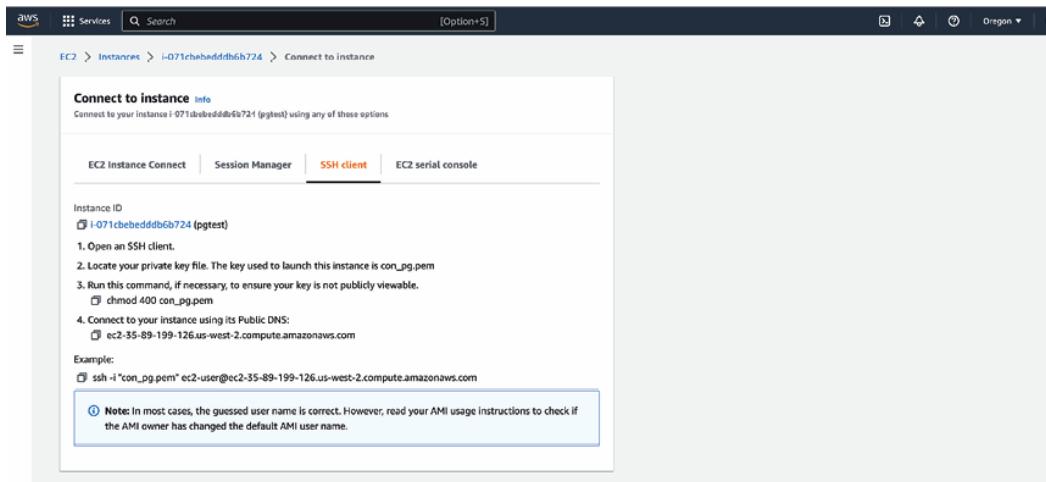
**Figure 3.8:** AWS EC2 Instance - Default settings

5. We are now ready to connect to the EC2 instance named **pgec2cluster** by selecting the **Connect to instance** tab as shown in *Figure 3.9*:



**Figure 3.9:** AWS EC2 Instance - Network settings

6. The **Connect to instance** tab redirects to the new window as shown in [Figure 3.10](#), which provides various connection methods. However, here we use the **SSH client** option to log in to the **pgec2cluster** instance from our local machine:



**Figure 3.10:** AWS EC2 - Instance launch succeed

7. The SSH connection information is outlined in [Figure 3.10](#), and we use this detail to configure the connection of the pgec2cluster instance from Mac OS. Refer to [Figure 3.11](#). At this point, we launched the Mac terminal, navigated to the directory that contains the private key named **con\_pg.pem** that we generate in step 4 of this recipe:

```
[mohammad@MOHAMMADs-MacBook-Pro ~ %]
[mohammad@MOHAMMADs-MacBook-Pro ~ % cd Downloads/AWS_Pair_key]
[mohammad@MOHAMMADs-MacBook-Pro AWS_Pair_key % ls
con_pg.pem
[mohammad@MOHAMMADs-MacBook-Pro AWS_Pair_key % chmod 400 con_pg.pem
[mohammad@MOHAMMADs-MacBook-Pro AWS_Pair_key %
mohammad@MOHAMMADs-MacBook-Pro AWS_Pair_key %]
```

**Figure 3.11:** AWS EC2 Instance- Ker pair setting with Local SSH client

- Now initiates the connection using its public DNS using following command:

**# Connect to EC2 instance using SSH.**

```
ssh -i "con_pg.pem" ec2-user@ec2-54-244-149-254.us-west-
2.compute.amazonaws.com
```

The message will be prompted to add an entry to the known host, type yes and you have successfully logged on to the pgec2cluster instance as shown in

**Figure 3.12:**

```
[mohammad@MOHAMMADs-MacBook-Pro AWS_Pair_key %
[mohammad@MOHAMMADs-MacBook-Pro AWS_Pair_key % ssh -i "con_pg.pem" ec2-user@ec2-54-244-149-254.us-west-2.compute.amazonaws.com
The authenticity of host 'ec2-54-244-149-254.us-west-2.compute.amazonaws.com (54.244.149.254)' can't be established.
ED25519 key fingerprint is SHA256:tjb3t2NF80As6mKrxk++RfzdmDp4QI0E05q/PgEYxow.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'ec2-54-244-149-254.us-west-2.compute.amazonaws.com' (ED25519) to the list of known hosts.
Register this system with Red Hat Insights: insights-client --register
Create an account or view all your systems at https://red.ht/insights-dashboard
[ec2-user@ip-172-31-10-232 ~]$ [ec2-user@ip-172-31-10-232 ~]$ [ec2-user@ip-172-31-10-232 ~]$
```

**Figure 3.12:** AWS EC2 Instance- Login with local SSH client

- At this point we are ready to begin the installation of PostgreSQL, but before that we have to add a PostgreSQL repository using the following script:

**# Add PostgreSQL 15 repository**

```
sudo dnf install -y
```

```
https://download.postgresql.org/pub/repos/yum/reporpms/EL-9-x86\_64/pgdg-redhat-repo-latest.noarch.rpm
```

The execution output of the above script is shown in following figure:

```

AWS_Pair_key - ec2-user@ip-172-31-10-232:~ ssh -i con_pg.pem ec2-user@ec2-54-244-149-254.us-west-2.compute.amazonaws.com - 155x39
[ec2-user@ip-172-31-10-232 ~]$ [ec2-user@ip-172-31-10-232 ~]$ [ec2-user@ip-172-31-10-232 ~]$ sudo dnf install -y https://download.postgresql.org/pub/repos/yum/reporpm/EL-9-x86_64/pgdg-redhat-repo-latest.noarch.rpm
Updating Subscription Management repositories.
Unable to read consumer identity

This system is not registered with an entitlement server. You can use subscription-manager to register.

Last metadata expiration check: 1:48:36 ago on Sun 27 Nov 2022 09:36:19 AM UTC.
pgdg-redhat-rpm-latest.noarch.rpm
Dependencies resolved.
=====
Package           Architecture   Version      Repository    Size
=====
Installing: pgdg-redhat-repo          noarch     42.0-28    @commandline 12 k
Transaction Summary
=====
Install 1 Package

Total size: 12 k
Installed size: 14 k
Downloading Packages:
Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
Preparing : 1/1
Installing : pgdg-redhat-repo-42.0-28.noarch 1/1
Verifying  : pgdg-redhat-repo-42.0-28.noarch 1/1
Installed products updated.

Installed:
  pgdg-redhat-repo-42.0-28.noarch

Complete!
[ec2-user@ip-172-31-10-232 ~]$

```

**Figure 3.13:** AWS EC2 Instance- PostgreSQL repository setting

- Now that the **PostgreSQL** repository is in place, the next step is to run the installation with the following command:

*# Disable the built-in PostgreSQL module:*

```
sudo dnf -qy module disable postgresql
```

*# Execute the PostgreSQL 15 Installation*

```
sudo dnf install -y postgresql15-server
```

The execution output of the above script is shown in following figure:

```

AWS_Pair_key - ec2-user@ip-172-31-10-232:~ ssh -i con_pg.pem ec2-user@ec2-54-244-149-254.us-west-2.compute.amazonaws.com - 157x39
[ec2-user@ip-172-31-10-232 ~]$ [ec2-user@ip-172-31-10-232 ~]$ [ec2-user@ip-172-31-10-232 ~]$ sudo dnf install postgresql15-server -y
Updating Subscription Management repositories.
Unable to read consumer identity

This system is not registered with an entitlement server. You can use subscription-manager to register.

Last metadata expiration check: 0:01:38 ago on Sun 27 Nov 2022 11:41:26 AM UTC.
Dependencies resolved.
=====
Package           Architecture   Version      Repository    Size
=====
Installing: postgresql15-server        x86_64      15.1-1PGDG.rhel9      pgdg15      6.0 M
Installing dependencies:
  libicu             x86_64      67.1-9.el9      rhel-9-baseos-rhui-rpms  9.6 M
  lzo               x86_64      1.9.3-5.el9      rhel-9-baseos-rhui-rpms  62 k
  postgresql15      x86_64      15.1-1PGDG.rhel9      pgdg15      1.5 M
  postgresql15-libs  x86_64      15.1-1PGDG.rhel9      pgdg15      296 k
Transaction Summary
=====
Install 5 Packages

Total download size: 17 M
Installed size: 66 M
Downloading Packages:
(1/5): postgresql15-libs-15.1-1PGDG.rhel9.x86_64.rpm      161 kB/s | 296 kB  00:01
(2/5): lzo-1.9.3-5.el9.x86_64.rpm      1.1 kB/s | 62 kB  00:00
(3/5): libicu-67.1-9.el9.x86_64.rpm      52 kB/s | 9.6 MB  00:00
(4/5): postgresql15-15.1-1PGDG.rhel9.x86_64.rpm      676 kB/s | 1.5 MB  00:02
(5/5): postgresql15-server-15.1-1PGDG.rhel9.x86_64.rpm      1.7 kB/s | 6.0 MB  00:03

Total
PostgreSQL 15 for RHEL / Rocky 9 - x86_64
Importing GPG key 0x642DF0F8:
Userid   : "PostgreSQL RPM Building Project <pgsql-pkg-yum@postgresql.org>"
Fingerprint: 68C9 E2B9 1A37 D136 F674 D176 1F16 D2E1 442D F0F8
1.6 kB/s | 1.7 kB  00:00
4.8 MB/s | 17 MB  00:03

```

**Figure 3.14:** AWS EC2 Instance- PostgreSQL installation

- We are now ready to initialize Postgres after successful completion of the installation in step 9:

```
# Initialize database
sudo /usr/pgsql-15/bin/postgresql-15-setup initdb
# Look for the following message output for the successful initialization
Initializing database ... OK
```

12. Afterwards, you can start-up the **PostgreSQL** database by using **systemctl** command-line utility:

```
# Enabling PostgreSQL 15 Service
sudo systemctl enable postgresql-15.service
# Start the PostgreSQL database cluster Instance
sudo systemctl start postgresql-15.service
# Verify the status of PostgreSQL database cluster
Instance
sudo systemctl status postgresql-15.service
```

Now that you have learned how to set up the PostgreSQL database cluster for the AWS EC2 instance, Let us get ready with the next recipe for setting up the PostgreSQL database cluster on the AWS RDS instance.

## Recipe 21: Managing PostgreSQL instance with AWS RDS

Following are the steps to manage PostgreSQL instance with AWS RDS:

1. At First, select the **Root user** at the AWS account login prompt and click **Next** to continue the login process. **Root user** credential provides full access to all resources, whereas **IAM** credential can secure access to AWS services and resources for users of your AWS account as shown in [Figure 3.15](#):



## Sign in

**Root user**

Account owner that performs tasks requiring unrestricted access. [Learn more](#)

**IAM user**

User within an account that performs daily tasks. [Learn more](#)

Root user email address

[REDACTED].com

**Next**

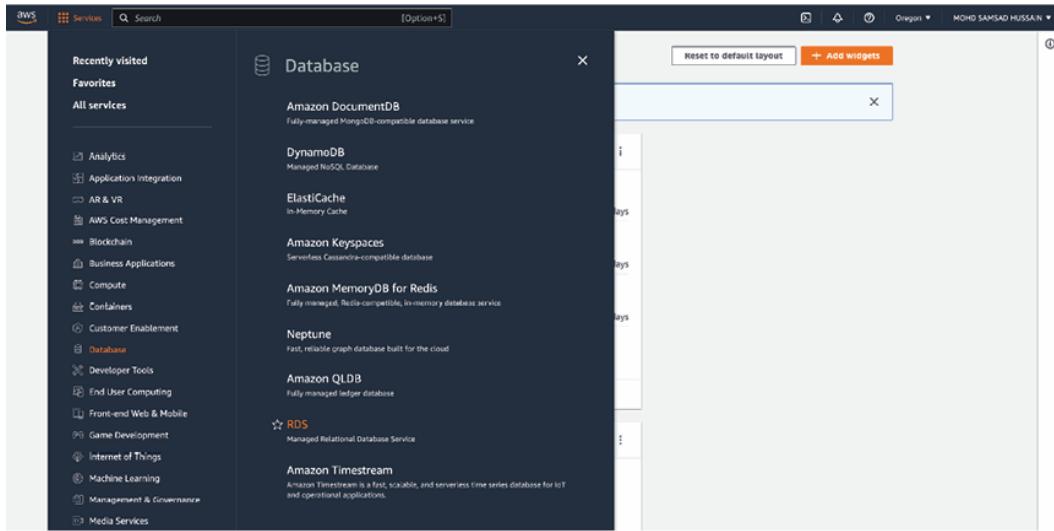
By continuing, you agree to the [AWS Customer Agreement](#) or other agreement for AWS services, and the [Privacy Notice](#). This site uses essential cookies. See our [Cookie Notice](#) for more information.

— New to AWS? —

**Create a new AWS account**

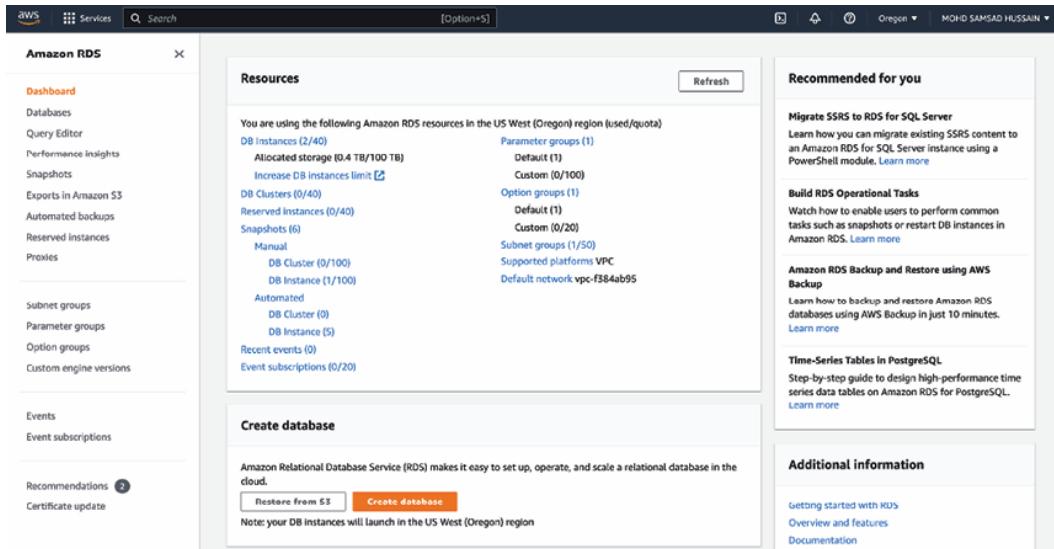
**Figure 3.15:** AWS account login option

2. Now we are in the **AWS Management console** home page, click the **Services** tab that list all of the services. All we have to do is select the **RDS** in the **Database** service category, which will take us to the **Amazon RDS** console as shown in [Figure 3.16](#):



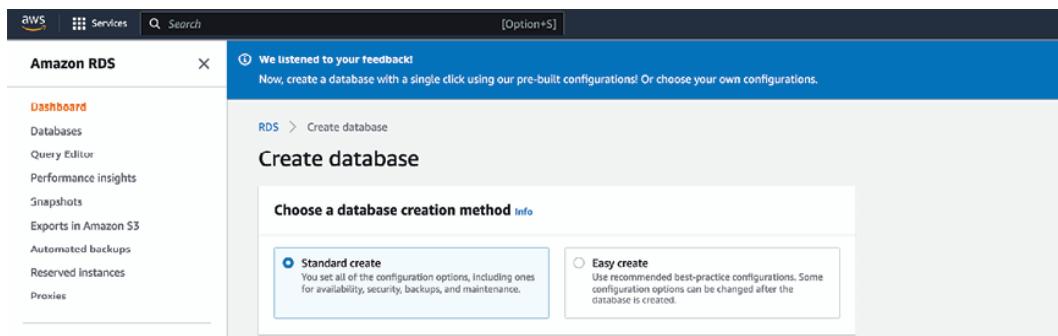
**Figure 3.16:** AWS Database service category - RDS

3. Refer to [Figure 3.17](#), from the **RDS Console** dashboard, in the **Create database** section, choose **Create database**. This will navigate to the **RDS Create database** dashboard.



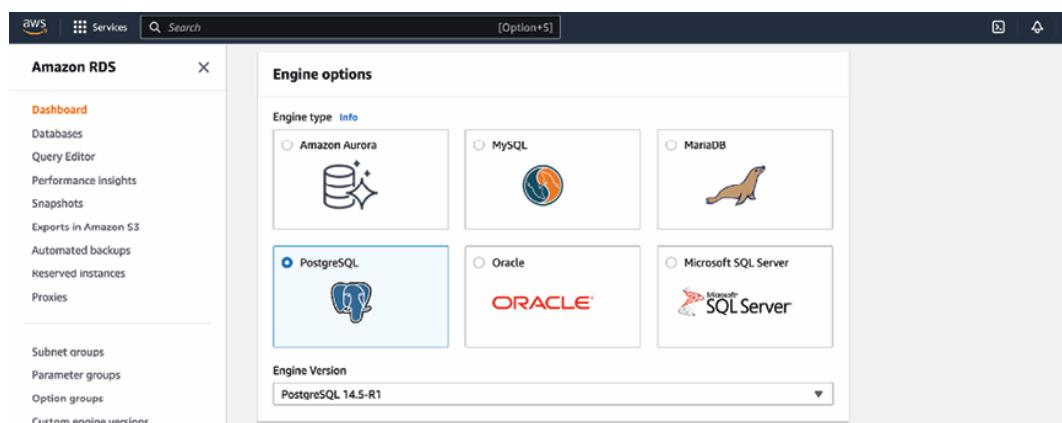
**Figure 3.17:** AWS RDS console dashboard

4. Here RDS **Create database** dashboard allows you to input the following:
  - a. Select the option **Standard create** under the section **Create database** as shown in [Figure 3.18](#):



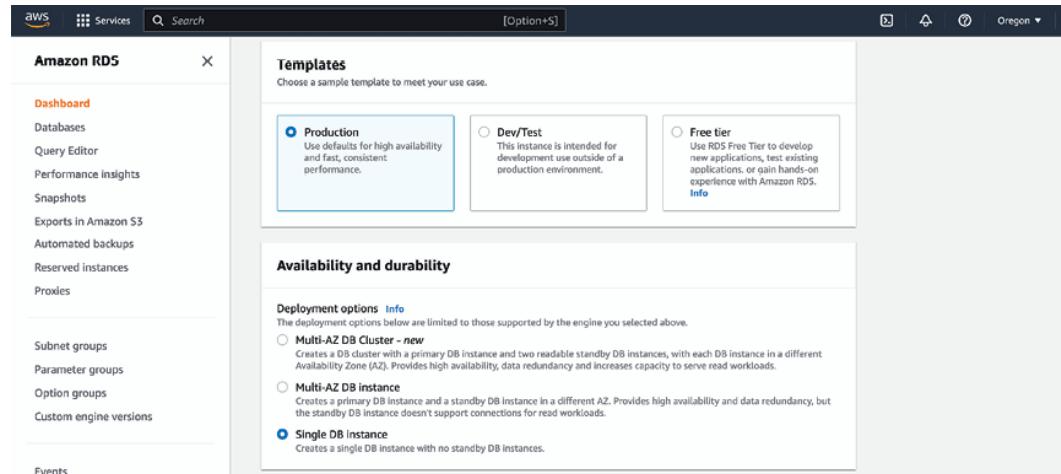
**Figure 3.18:** AWS RDS - Create database page

- Refer to [Figure 3.19](#), under **Engine options** section, choose PostgreSQL as the engine type and **PostgreSQL 14.5-R1** as the engine version:



**Figure 3.19:** AWS RDS - Database engine options

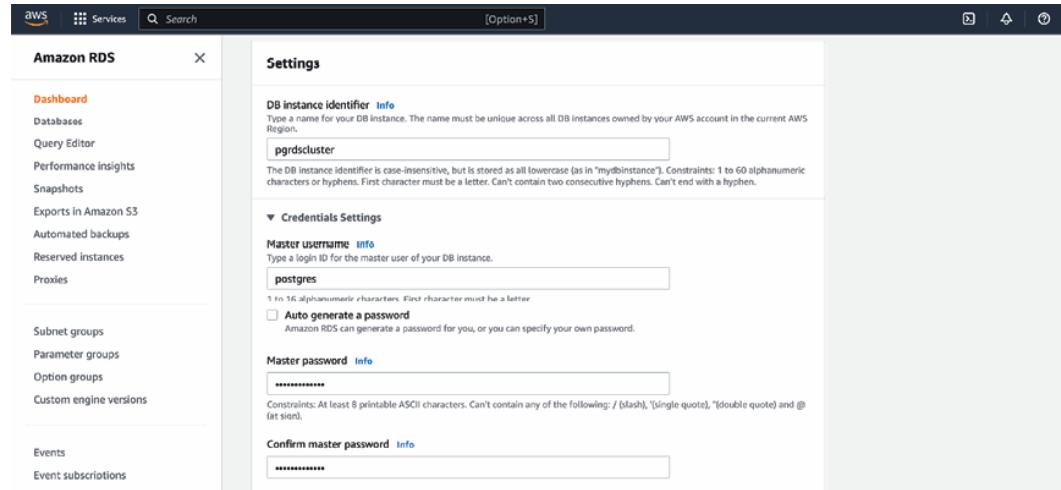
- Refer to [Figure 3.20](#), we are choosing the **Free tier** Under the **Template** section and select **Single DB instance** under the **Availability and durability** section:



**Figure 3.20:** AWS RDS - Database templates

d. Under **Settings** section:

- Input the name of your DB instance as **DB instance identifier**, for this recipe we use the name **pgrdscluster**.
- Enter the username and password in the **Master username** and **Master password** section. This user name must be used when logging into the database instance:



**Figure 3.21:** AWS RDS - Database instance setting

- At this point, we keep the **Instance configuration**, **Storage**, **Connectivity**, **Database authentication** and **Monitoring** options as default setting and ready to create our RDS instance named **pgrdscluster** by clicking on the

**Create database** tab as shown in *Figure 3.22* and *Figure 3.23*:

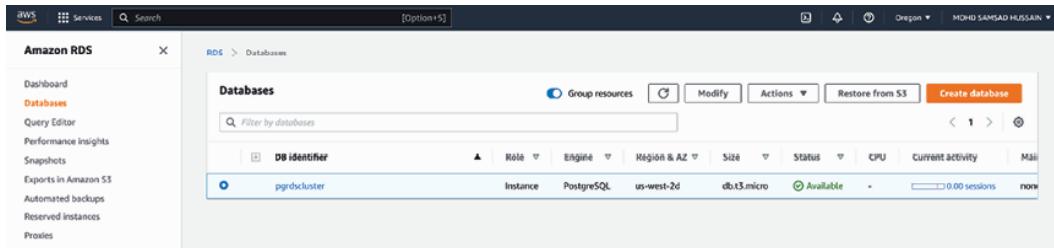
The figure consists of three vertically stacked screenshots of the AWS RDS console.

- Screenshot 1 (Top): Instance configuration**
  - DB instance class:** Standard classes (includes m-class)
    - Standard classes (includes m-class)
    - Provisioned IOPS (SSD) classes
    - Backups (includes t (Standard))
  - Storage:**
    - Storage type:** Provisioned IOPS (SSD) (1000 GB)
    - Allocated storage:** 4000
    - Performance IOPS:** 3000
  - Storage auto-scaling:**
    - Enable storage auto-scaling
    - Scaling target threshold:** 1000
- Screenshot 2 (Middle): Connectivity**
  - Connectivity:**
    - Don't connect to an EC2 compute resource
    - Connect to an EC2 compute resource
  - Network type:**
    - IPv4
    - Dual-stack mode
  - VPC and private IP address (VPC):** Choose the VPC. The VPC defines the virtual networking environment for this DB instance. Default VPC (vpc-034a605)
  - DB subnet group:** Choose the DB subnet group which defines which subnets and IP ranges the DB instance can use in the VPC that you selected. default-vpc-034a605
  - Public access:**
    - Yes
    - No
  - VPC security group (VPC):** Choose one or more VPC security groups to allow access to your database. Make sure that the security group rules allow the appropriate incoming traffic.
    - Choose existing: Create new VPC security group
  - Default VPC security group:** Choose user or instance options. default
  - Availability Zone:** No preference
- Screenshot 3 (Bottom): Creating Database page**
  - Creating database pgardscluster**: Your database might take a few minutes to launch.
  - Databases** table:
 

DB identifier	DB cluster identifier	Role	Engine	Region & AZ	Size	Status	CPU	Cloud
pgardscluster	-	Instance	PostgreSQL	db.t3.micro	-	Creating	-	-

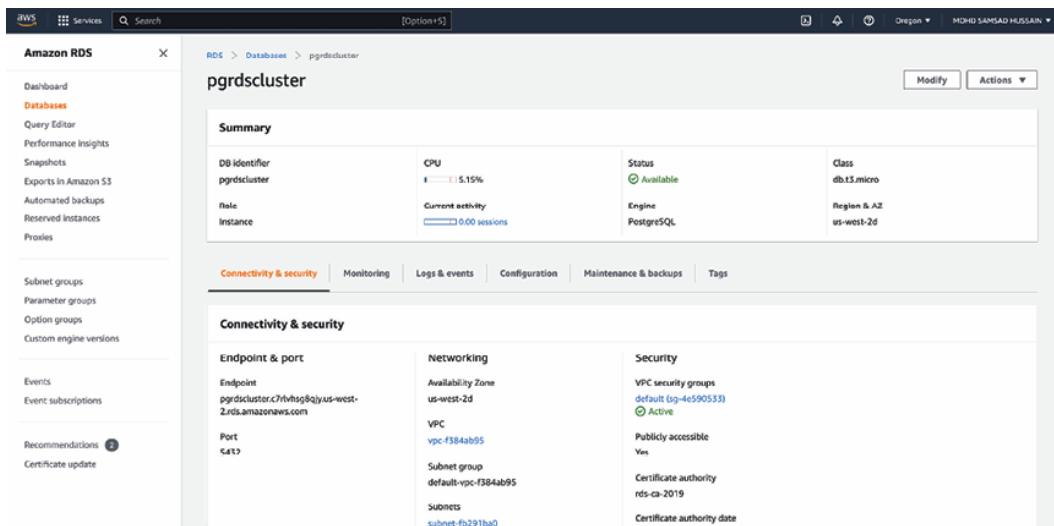
**Figure 3.22:** AWS RDS – instance creation settings and AWS RDS – Creating Database page

- Refer to *Figure 3.23*, instance is now being created and we are ready to connect to the PostgreSQL RDS instance named **pgardscluster**. Connection to the instance need a client tool. Refer to the recipe *Install and configure pgAdmin client tool* of *Chapter 1* for detail on how to install pgAdmin tool.



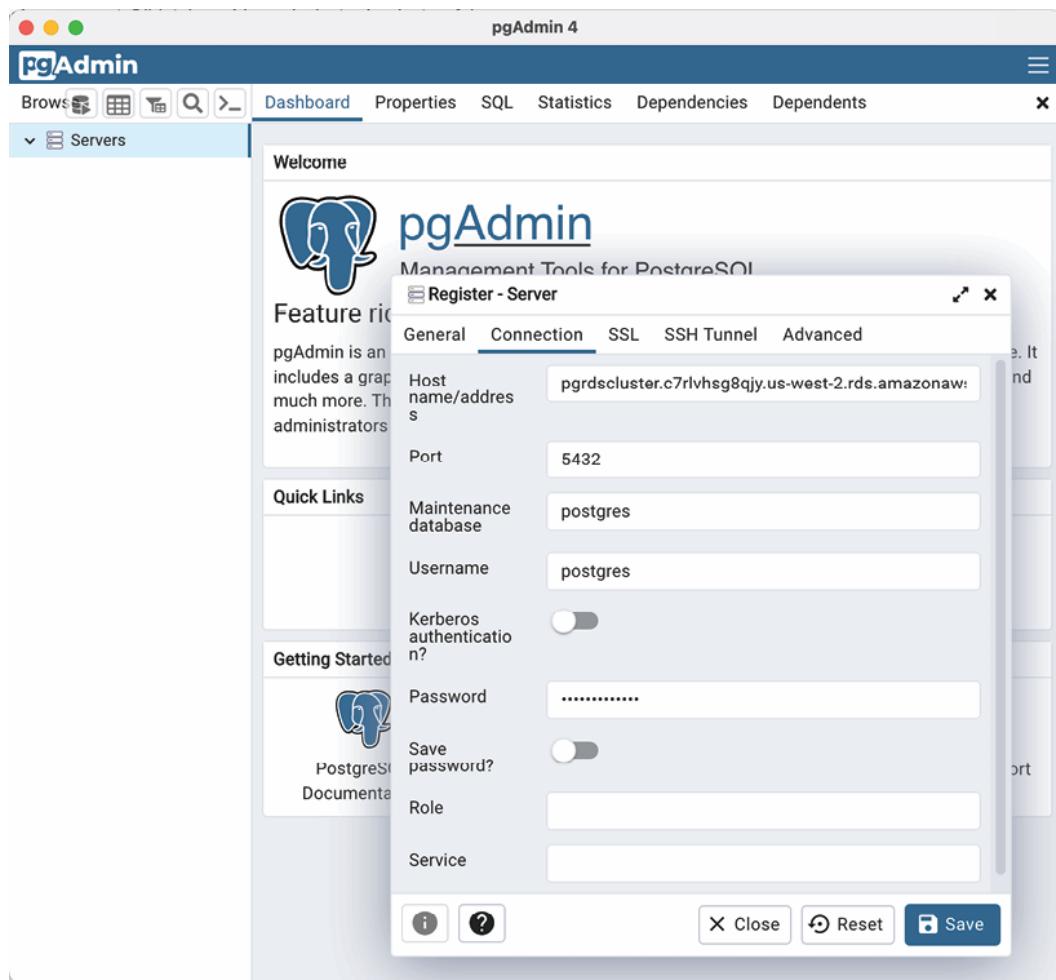
**Figure 3.23:** AWS RDS -Database dashboard

6. About [Figure 3.23](#), the PostgreSQL RDS instance has already been created, select **pgrdscluster** in the **Databases** tab of the Amazon RDS Databases dashboard. This will take you to the **pgrdscluster** cluster detail dashboard shown in [Figure 3.24](#), which contains the **Endpoint & port** information under **Connectivity & security** tab. Using **Endpoint & port** information, the connection to the PostgreSQL RDS instance must be done from any client tool.



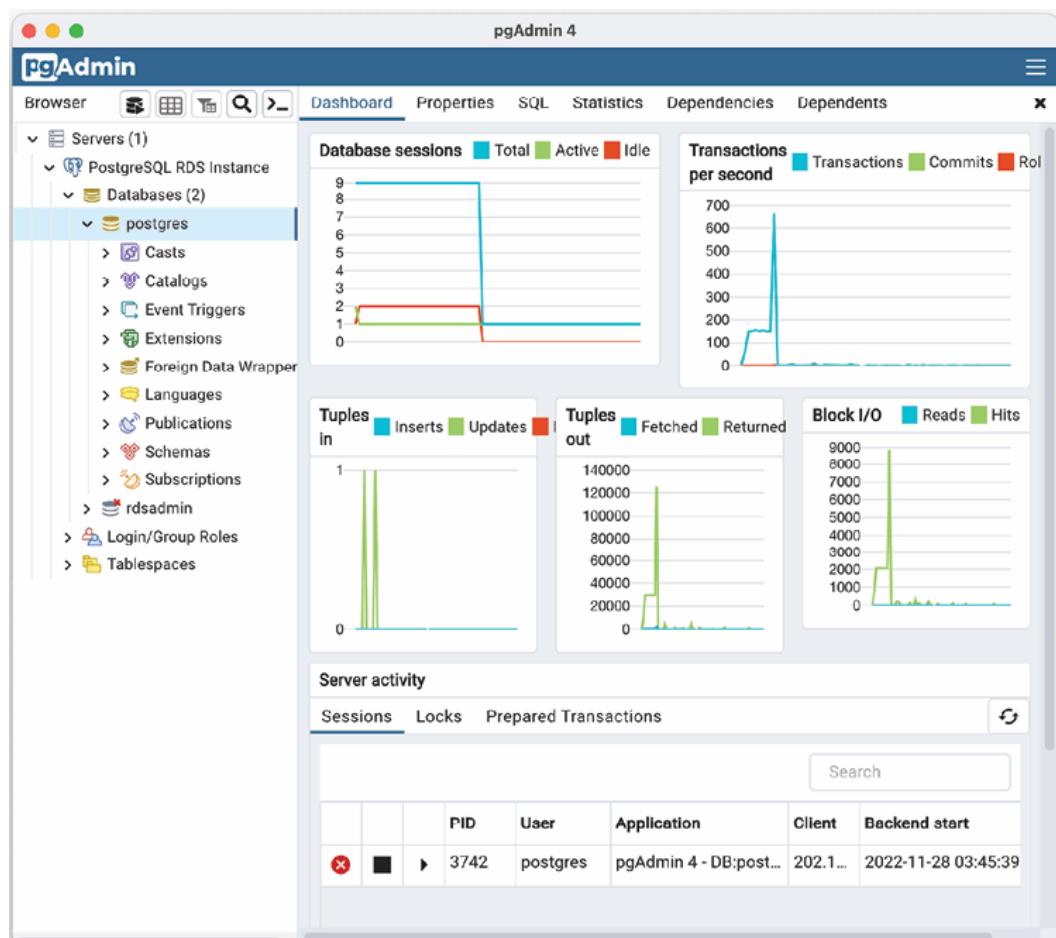
**Figure 3.24:** AWS RDS -Database cluster information

7. The **Endpoint & port** information described in [Figure 3.24](#) under the **Connectivity & security** tab, and we use this information to set-up the connection of the pgrdscluster instance from the **pgAdmin** tool. Refer to the following figure:



**Figure 3.25:** AWS RDS -pgAdmin connection

8. At this stage, we have succeeded in configuring the PostgreSQL database cluster on the AWS RDS instance as shown in figure:



**Figure 3.26:** AWS RDS -Database connection dashboard

Eventually, if we want an SSH connection to the RDS instance, it can be done using an SSH tunnel to channel through an EC2 machine. Once connected to the EC2 instance, use DB endpoint credentials to connect to the RDS instance using SQL connection commands.

Now that you have learned how to set up the PostgreSQL database cluster for the AWS RDS instance. The following section of the recipe is specific to backing up and restoring the PostgreSQL database on the AWS EC2 and RDS instances.

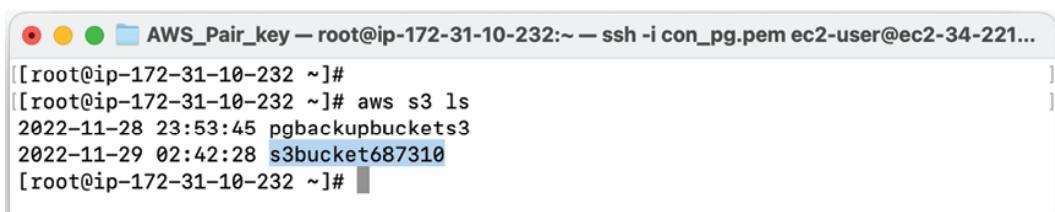
## Recipe 22: Native backup or restore with AWS EC2 instance

This recipe will look at backup and restore operations with utilities such as **pg\_dump** to restore the PostgreSQL database on the AWS EC2 instance. The **pg\_dump** program designed to perform a backup, as this utility performs an interaction with PostgreSQL instance, which is in

execution, makes **pg\_dump** an excellent choice for database backup and restore option.

In the following steps, let us look at PostgreSQL database cluster backup and restore on an AWS EC2 instance:

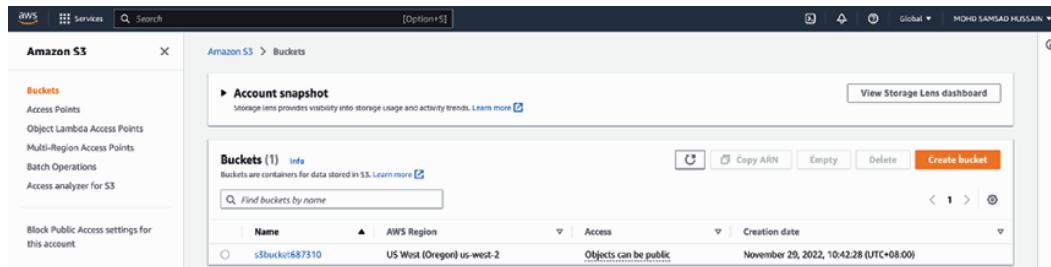
1. First, we will SSH to the AWS EC2 instance from local machine terminal, Refer to the recipe *Create PostgreSQL Cloud Instance and Manage Database Connection with AWS EC2* in [Chapter 3](#), Cloud Provisioning.
2. In reference to [Figure 3.27](#) and [Figure 3.28](#), for this backup setup, we have configured S3 bucket named **s3bucket687310** and granted access to S3 Bucket from EC2 instance named **pgec2cluster**:



```
AWS_Pair_key — root@ip-172-31-10-232:~— ssh -i con_pg.pem ec2-user@ec2-34-221...
[root@ip-172-31-10-232 ~]# [root@ip-172-31-10-232 ~]# aws s3 ls
2022-11-28 23:53:45 pgbackupbuckets3
2022-11-29 02:42:28 s3bucket687310
[root@ip-172-31-10-232 ~]#
```

**Figure 3.27:** AWS EC2 – List S3 bucket

Refer to [Figure 3.28](#), to view S3 Bucket dashboard:



**Figure 3.28:** AWS EC2 –S3 Bucket dashboard

**Note: The method of configuring AWS S3 Bucket is beyond the scope of this book, however, the backup steps on S3 bucket given here are based on the actual deployment in AWS EC2 Instance.**

3. In this step we have written a database backup shell script, the underlying methodology of this backup script is as follows:
  - a. The first section of the script will execute the backup of the

PostgreSQL database cluster hosted on **AWS EC2**, and dump the backup image to the local mount point named **/PG BACKUP**.

- b. Once the database has been successfully backed up, the backup image will be copied onto the S3 bucket named **s3buckand687310** from the / **PG\_BACKUP** mount point.

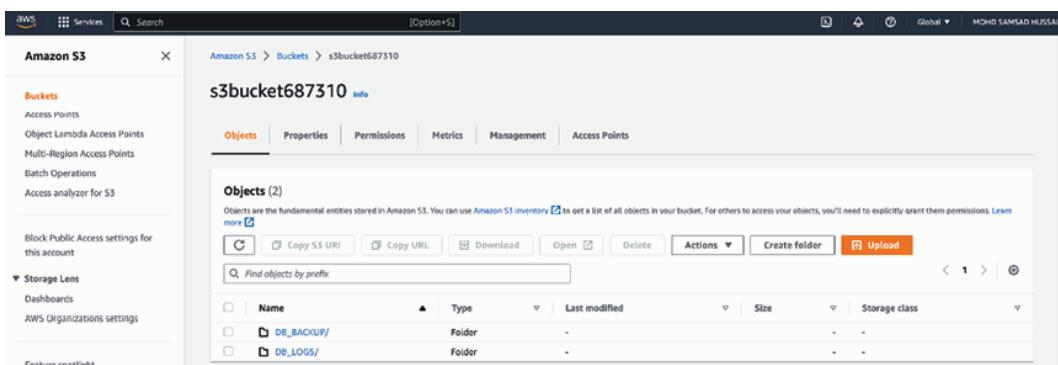
```

for database in $DB_NAME; do
    PG_PASSWORD=${PG_DB_PASS} pg_dump \
        --host=${PG_DB_HOST} \
        --port=${PG_DB_PORT} \
        --username=${PG_DB_USER} \
        ${database} | gzip > ${LOCAL_BACKUP_PATH}/${database}-$DATE.sql.gz
    ##### Upload Database Backup
    To S3 Bucket #####
    aws s3 cp ${LOCAL_BACKUP_PATH}/${database}-$DATE.sql.gz
    s3://${S3_BUCKET_NAME}/${S3_BUCKET_PATH}/
done

```

**Note: For this recipe, we've used gzip compression for backup. However, it is worth noting that there exists a built-in compression method for backup and restore, which we will explore in detail in Chapter 10, Backup and Chapter 11, Recovery later in this book.**

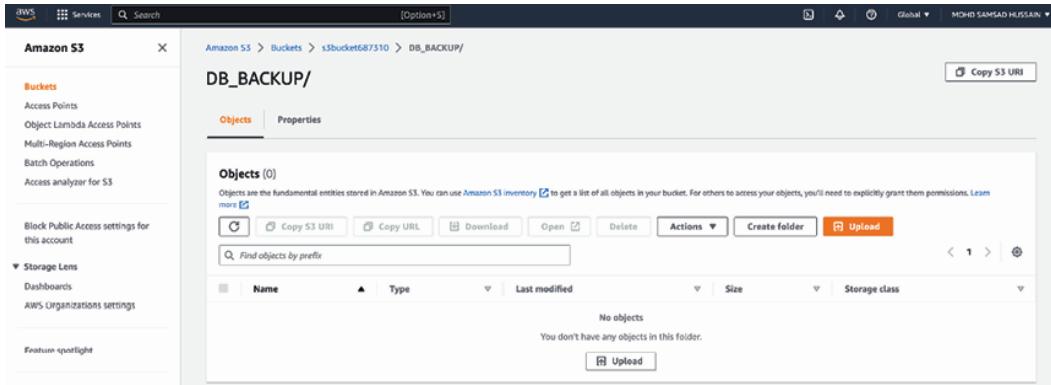
- Once the backup script is successfully prepared you are good to move to the next step of backup execution, but before that verify the S3 bucket directory as shown in the *Figure 3.29*:



*Figure 3.29: AWS EC2 -S3 Bucket object list*

- Refer to *Figure 3.30*, the URI for our S3 Bucket is **s3://s3bucket687310/DB\_BACKUP** where we store our

PostgreSQL database backup Image. Click on **copy S3 URI** tab to obtain the URI of your S3 object storage path.



**Figure 3.30:** AWS EC2 -S3 Bucket Backup directory object list

- Now we are ready to perform the backup of PostgreSQL database cluster on EC2 instance using the shell script created in step 2 of this recipe using following command:

```
# Execute PostgreSQL Backup on EC2 Instance
```

```
./pg_backup_to_s3.sh
```

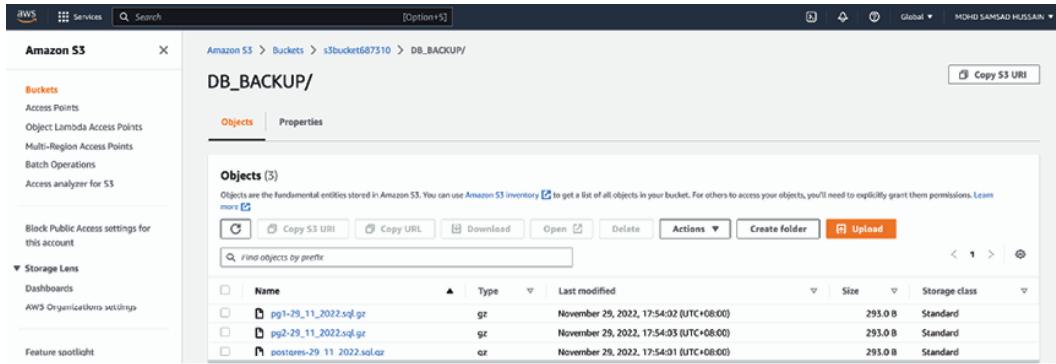
- Figure 3.31** shows the PostgreSQL database backup completed in the EC2 instance and the backup image uploaded into S3 Bucket:

```
[root@ip-172-31-10-232 Backup_Script]# ./pg_backup_to_s3.sh
upload: ./PG_BACKUP/postgres-29_11_2022.sql.gz to s3://s3bucket687310/DB_BACKUP/postgres-29_11_2022.sql.gz
upload: ./PG_BACKUP/pg1-29_11_2022.sql.gz to s3://s3bucket687310/DB_BACKUP/pg1-29_11_2022.sql.gz
upload: ./PG_BACKUP/pg2-29_11_2022.sql.gz to s3://s3bucket687310/DB_BACKUP/pg2-29_11_2022.sql.gz
[root@ip-172-31-10-232 Backup_Script]# 11 /PG_BACKUP/
total 12
-rw-r--r--. 1 root root 293 Nov 29 09:54 pg1-29_11_2022.sql.gz
-rw-r--r--. 1 root root 293 Nov 29 09:54 pg2-29_11_2022.sql.gz
-rw-r--r--. 1 root root 293 Nov 29 09:54 postgres-29_11_2022.sql.gz
[root@ip-172-31-10-232 Backup_Script]#
```

**Figure 3.31:** AWS EC2 - Database backup with S3 Bucket

**Note: Besides these options, you can also carry out backup operations using the AWS backup, which provides the EC2 instance's scheduled and on-demand backup.**

- Verify the backup image on the S3 bucket up-loaded in step 5 of this recipe as shown in **Figure 3.32**:



**Figure 3.32:** AWS EC2 – Database backup upload to S3 backup

9. We can recover the database at this stage with the backup image generated at step 5 of this recipe. Let us see how recovery works in EC2 instance for PostgreSQL database. But before that, we create a **schema** and **table** in the **postgres** database using the following script:

```
# Create Schema on postgres database
create schema book;
# Create Table under book schema.
CREATE TABLE book.ec2_recovery (recovery_id integer, recovery_name text);
# Verify the table with its corresponding schema on postgres
database
\dt *.*
```

The following figure is the output of the above script execution:

```
postgres=# create schema book;
CREATE SCHEMA
postgres=# CREATE TABLE book.ec2_recovery (recovery_id integer, recovery_name text );
CREATE TABLE
postgres=#
postgres=# \dt *.*
          List of relations
   Schema   |      Name       |   Type   | Owner
   book     | ec2_recovery  | table    | postgres
 information_schema | sql_features  | table    | postgres
 information_schema | sql_implementation_info | table    | postgres
 information_schema | sql_parts    | table    | postgres
--More--
```

**Figure 3.33:** AWS EC2 – Create table

10. Database backup and recovery methods are almost identical in on-premise and cloud environments unless any cloud-specific tools are used. In this step, we first download the backup image

from S3 Bucket to the **/PG\_BACKUP** directory on the EC2 instance and unzip the backup image using the following command (Assume backup is not available at mount point **/PG\_BACKUP**):

```
# Download PostgreSQL database backup image from S3 to  
EC2 Instance.
```

```
aws s3 cp s3://s3bucket687310/DB_BACKUP/postgres-29_11_2022.sql.gz  
/PG_BACKUP/  
# Unzip the backup image.  
gunzip postgres-29_11_2022.sql.gz
```

The execution output of the above script is shown in following figure:

```
AWS_Pair_key - root@ip-172-31-10-232:~/PG_BACKUP - ssh -i con_pg.pem ec2-user@ec2-54-149-29-91.us-west-2.compute.amazonaws.com - 179x28  
[root@ip-172-31-10-232 PG_BACKUP]#  
[root@ip-172-31-10-232 PG_BACKUP]# aws s3 cp s3://s3bucket687310/DB_BACKUP/postgres-29_11_2022.sql.gz /PG_BACKUP/  
download: s3://s3bucket687310/DB_BACKUP/postgres-29_11_2022.sql.gz to ./postgres-29_11_2022.sql.gz  
[root@ip-172-31-10-232 PG_BACKUP]#  
[root@ip-172-31-10-232 PG_BACKUP]# gunzip postgres-29_11_2022.sql.gz  
[root@ip-172-31-10-232 PG_BACKUP]#  
[root@ip-172-31-10-232 PG_BACKUP]#
```

**Figure 3.34:** AWS EC2 – Download file from S3 Bucket

The backup image is now ready for restoration, we use the **pg\_dump** tool to restore the database, but other tools are also available for the restore operation that we go through in detail in [Chapter 10, Backup](#). For now, execute the following command to initiate the restore of **postgres** database:

```
# Restore backup to postgres database.
```

```
pg_dump --host=ip-172-31-10-232.us-west-2.compute.internal --port=5432 --  
username=postgres -Fp --clean -d postgres < postgres-29_11_2022.sql
```

The execution output of the above script is shown in following figure:

```
AWS_Pair_key - root@ip-172-31-10-232:~/PG_BACKUP - ssh -i con_pg.pem ec2-user@ec2-54-149-29-91.us-west-2.compute.amazonaws.com - 179x28  
[root@ip-172-31-10-232 PG_BACKUP]# 11  
total 4  
-rw-r-- 1 root root 489 Nov 29 09:54 postgres-29_11_2022.sql  
[root@ip-172-31-10-232 PG_BACKUP]# pg_dump --host=ip-172-31-10-232.us-west-2.compute.internal --port=5432 --username=postgres -Fp --clean -d postgres < postgres-29_11_2022.sql  
--  
-- PostgreSQL database dump  
--  
-- Dumped from database version 16.1  
-- Dumped by pg_dump version 15.1  
  
SET statement_timeout = 0;  
SET lock_timeout = 0;  
SET idle_in_transaction_session_timeout = 0;  
SET client_encoding = 'UTF8';  
SET standard_conforming_strings = on;  
SELECT pg_catalog.set_config('search_path', '', false);  
SET check_function_bodies = false;  
SET xmloption = content;  
SET client_min_messages = warning;  
SET row_security = off;  
  
--  
-- PostgreSQL database dump complete  
--  
[root@ip-172-31-10-232 PG_BACKUP]#
```

**Figure 3.35:** AWS EC2 – Database restore

After the database recovery, as expected, we can determine that the **schema & table** created after the backup operation is not available in the database:

```
# Verify schema & table on postgres database post restoration.
```

```
select * from pg_catalog.pg_namespace where nspname='book' ORDER BY  
nspname;
```

The execution output of the above script is shown in following figure:



```
AWS_Pair_key — postgres@ip-172-31-10-232:~ — ssh -i con_pg.pem ec2-user@ec2-54-149-29-91.us-...  
[postgres@ip-172-31-10-232 ~]$ psql  
psql (15.1)  
Type "help" for help.  
  
[postgres=# SELECT * FROM pg_catalog.pg_namespace where nspname='book' ORDER BY nspname;  
oid | nspname | nspowner | nspacl  
-----+-----+-----+-----  
(0 rows)  
  
postgres=# ]
```

**Figure 3.36:** AWS EC2 – Verify table post restore

## Recipe 23: Backup or restore with AWS RDS instance

Backing up and restoring the PostgreSQL database hosted in an EC2 instance is almost identical to the on-premise backup method. With PostgreSQL on AWS RDS, the database backup and restore operation can be simplified with a click.

RDS creates a storage volume snapshot of your DB instance, these backups are accomplished using on-demand or automated backup. However, backup is stored in S3 bucket but not managed in consumer bucket.

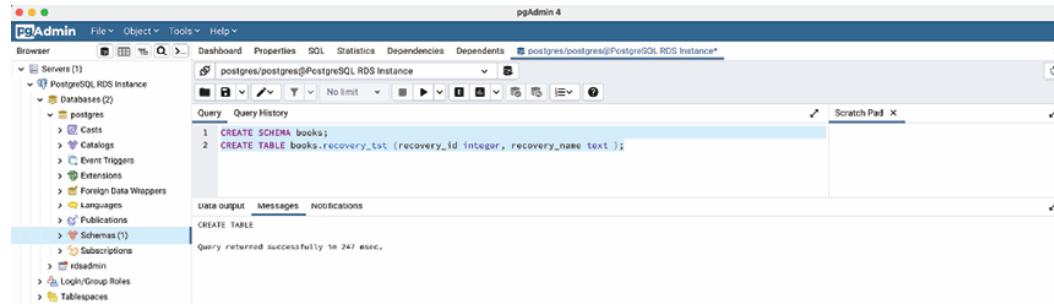
This recipe will look at backup and restore operations with AWS RDS snapshot backup or restore option. So let us start the recipe to backup and restore PostgreSQL database cluster on an AWS RDS instance in the following steps:

1. At First, select the **Root user** at the AWS account login prompt and click **Next** to continue the login process. Root user credential provides full access to all resources, whereas IAM credential can secure access to AWS services and resources for users of your AWS account.

- Now we are in the **AWS Management console** home page, click the **Service** tab that list all of the services. We must select the RDS in the **Database** service category, which will take us to the Amazon RDS Console.
- For this recipe, we created a **table books.recovery\_tbs**. To test recovery, we drop this table post successful snapshot backup:

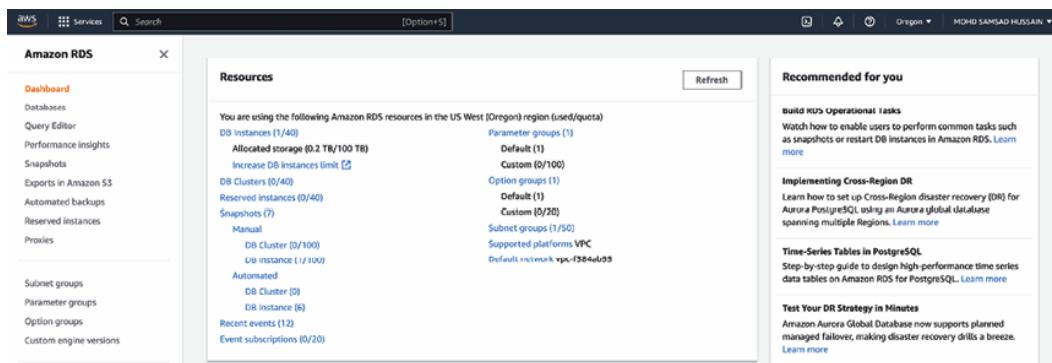
```
# Create Schema on postgres database
create schema books;
# Create Table under books schema.
CREATE TABLE books.ec2_recovery_tst (recovery_id integer, recovery_name text);
```

The execution output of the above script is shown in following figure:



**Figure 3.37:** AWS RDS – Create schema and table

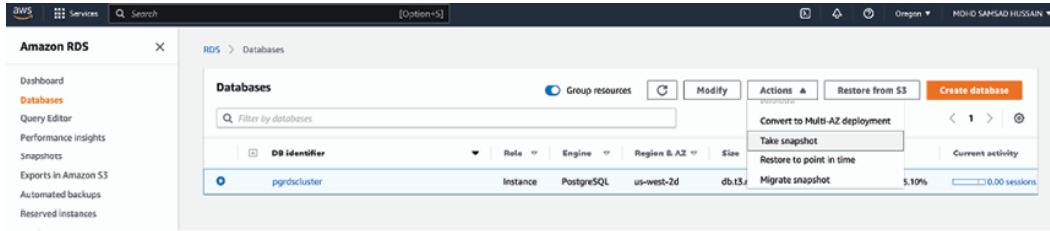
- Refer to [Figure 3.38](#), now we are AWS RDS Dashboard page, click on the **Db instances** under the **Resource** tab which will take us to the **Database console**:



**Figure 3.38:** AWS RDS Dashboard

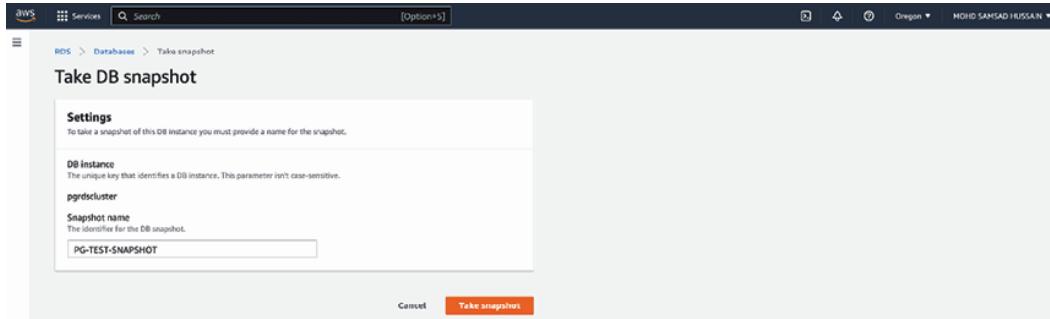
- Here select the **pgrdscluster** under the **Database** tab then select **Action** tab and choose **Take snapshot** option as shown in

the following figure:



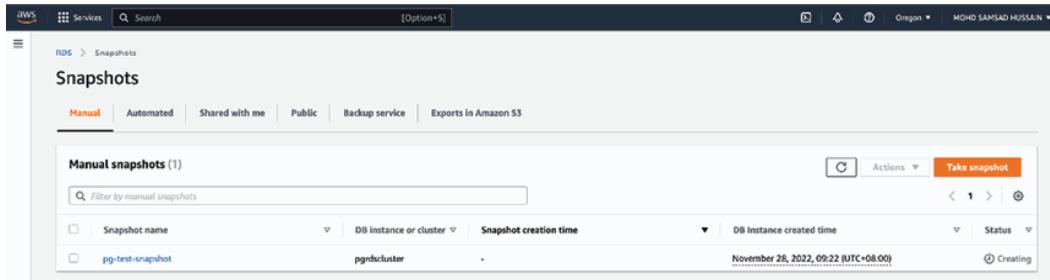
**Figure 3.39:** AWS RDS - Database dashboard

3. Refer to [Figure 3.40](#), the **Take snapshot** option opens a **Take DB snapshot** page where we need to enter the **Snapshot name**, here we choose **PG-TEST-SNAPSHOT** as the **Snapshot name** and click **Take snapshot** to start the snapshot backup of our RDS DB instance:



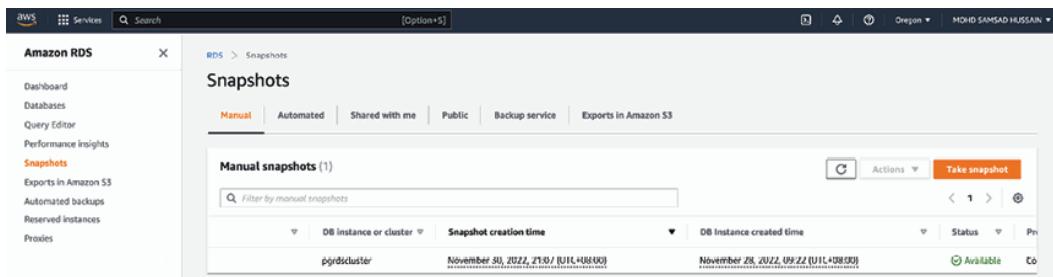
**Figure 3.40:** AWS RDS - Take DB snapshot

4. To verify the status of the Snapshot backup executed in the previous step, navigate to the **Snapshots** page under AWS RDS **Dashboard**. Refer to [Figure 3.41](#), the **Creating** entry in the **Status** column indicates that our snapshot task is in progress:



**Figure 3.41:** AWS RDS - Snapshot page

5. As the **Snapshot job** completed as shown in the following [Figure 3.42](#). Now we drop the table created on step 3 of this recipe:



**Figure 3.42:** AWS RDS - Snapshot job

Now we drop the table **books.recovery\_tst** from **postgres** database using following script:

```
# drop table
```

```
drop table books.ec2_recovery_tst;
```

```
# Verify the table under schema books
```

```
SELECT * FROM information_schema.tables where table_schema='books';
```

The execution output of the above script is shown in following figure:

The screenshot displays two pgAdmin 4 windows. The top window is connected to 'postgres/postgres@PostgreSQL RDS Instance' and shows a query editor with the following content:

```
1 drop table books.recovery_tst;
2
```

The message area below the editor shows: 'DROP TABLE' and 'Query returned successfully in 237 msec.' The bottom window is connected to 'postgres/postgres@PostgreSQL RDS Instance' and shows a query editor with the following content:

```
1
2 SELECT * FROM information_schema.tables where table_schema='books';
3
```

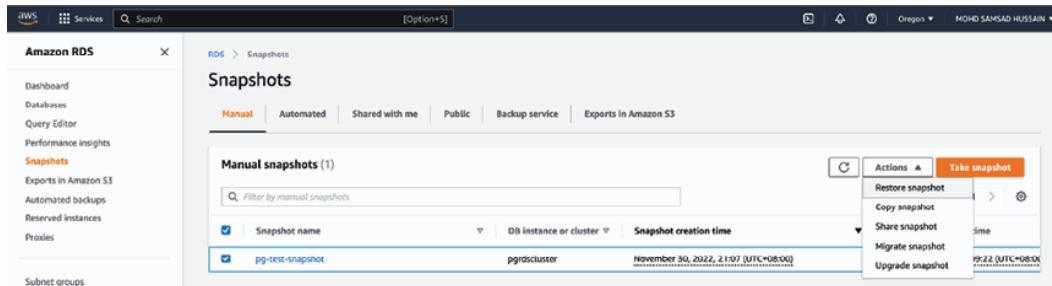
The message area below the editor shows a table with the following data:

table_catalog	table_schema	table_name	table_type	self_referencing_column_name	reference_generation	user_define
name	name	name	character varying	name	character varying	name

**Figure 3.43:** AWS RDS - Drop table

- Now we are in **Snapshots** page under **AWS RDS** to restore backup taken at step 6 of this recipe. Select **PG-TEST-SNAPSHOT** under the **Manual snapshot** tab and click **Actions**

tab and choose **Restore snapshot** from the dropdown list as shown in *Figure 3.44* that opens a **Restore snapshot** page:



*Figure 3.44: AWS RDS - Snapshot restore*

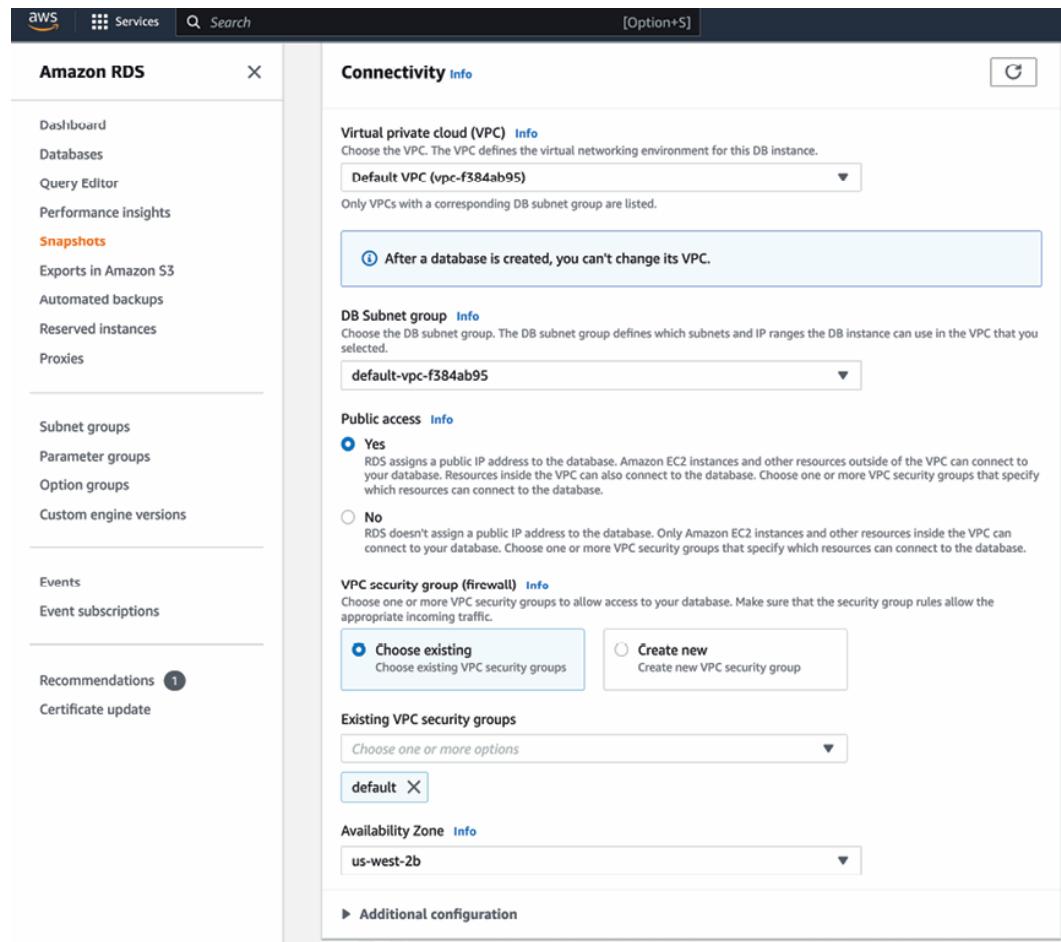
7. The **Restore snapshot** page allows you to input the following:

- a. In reference to *Figure 3.45*, select **DB engine** as **PostgreSQL** under the **DB instance settings** tab, select **Single DB instances** in the **Deployment options** section under the **Availability and durability** tab and we are choosing **pgrdscluster-snap-restore** in the **Instance identifier name** section under the **Settings** tab:

The screenshot shows the AWS RDS interface for restoring a database snapshot. The left sidebar lists various RDS management options like Dashboard, Databases, Query Editor, Performance insights, and Snapshots. Under Snapshots, 'Exports in Amazon S3' is selected. The main content area is titled 'Restore snapshot' and contains three tabs: 'DB instance settings', 'Availability and durability', and 'Settings'. The 'DB engine' dropdown is set to 'PostgreSQL'. In the 'Availability and durability' tab, the 'Single DB instance' option is selected. The 'Settings' tab shows the 'DB snapshot ID' as 'pg-test-snapshot' and the 'DB instance identifier' as 'prdrdscluster-snap-restore'.

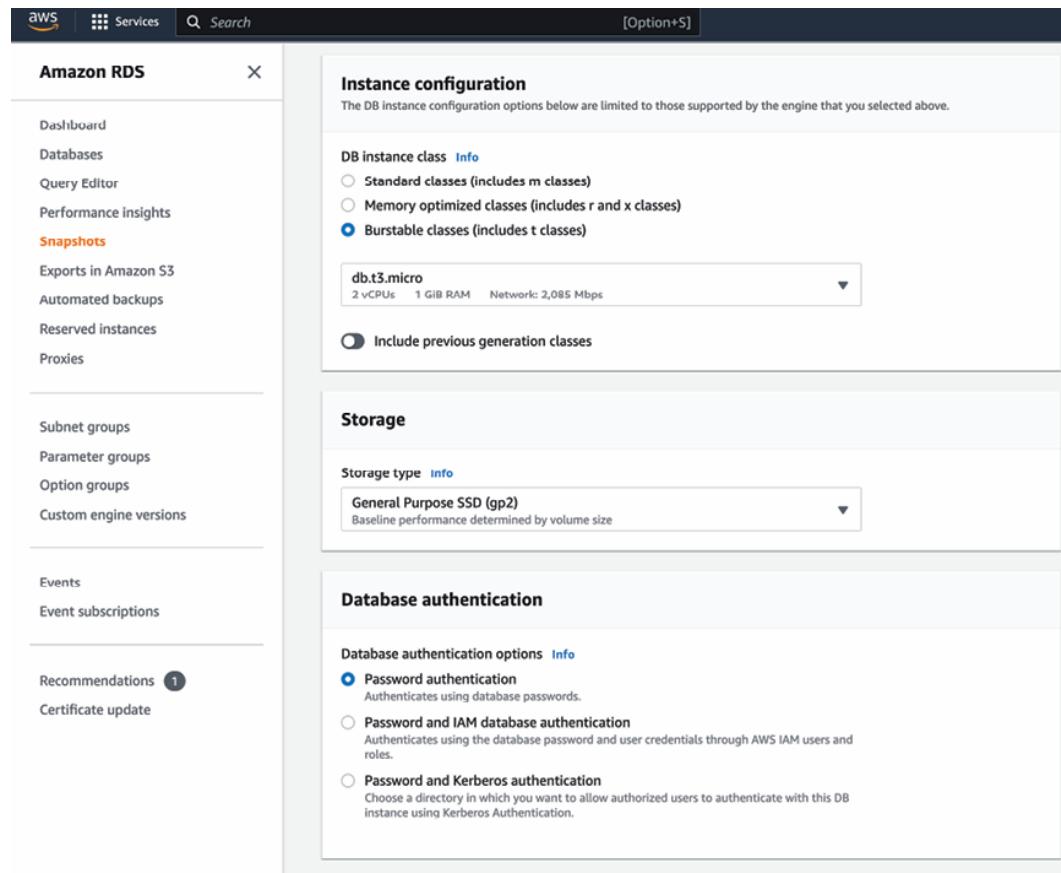
**Figure 3.45:** AWS RDS – Snapshot restore DB instance settings

- b. In reference to [Figure 3.46](#), we are keeping all the setting as **Default** under the **Connectivity** tab:



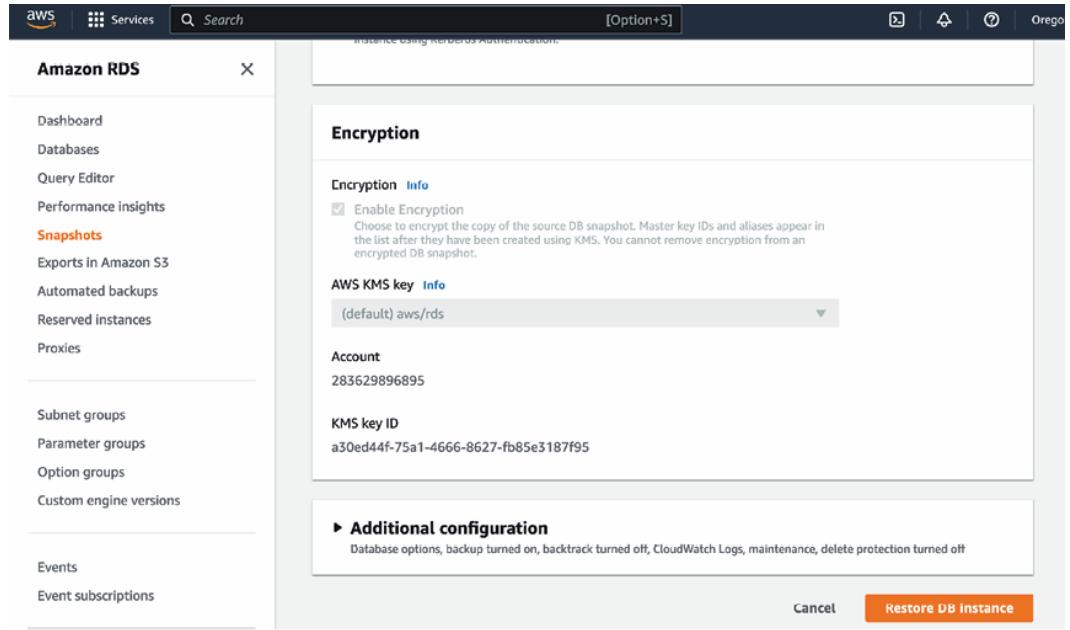
**Figure 3.46:** AWS RDS – Snapshot restore connectivity settings

- c. Refer to [Figure 3.47](#), select **db.t3.micro** in the **Burstable classes** section under the instance configuration and keeping the storage and database authentication as default selection:



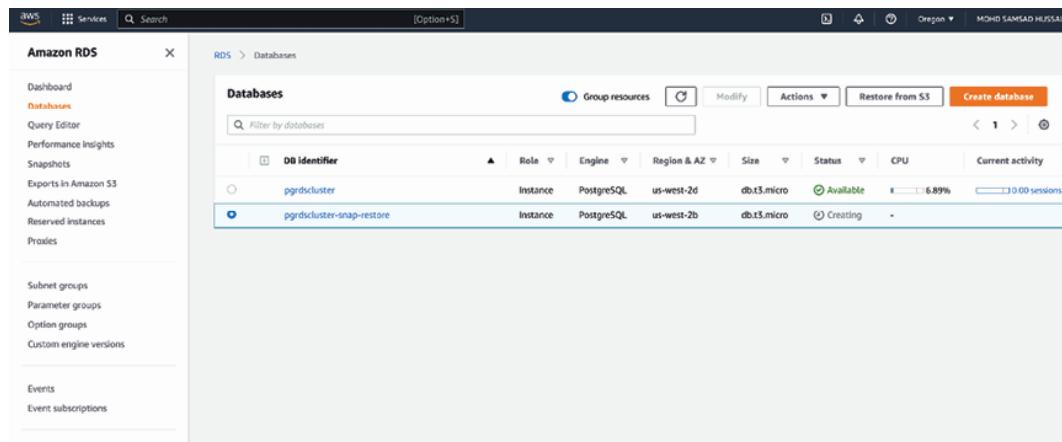
**Figure 3.47:** AWS RDS – Snapshot restore instance configuration and authentication settings

- d. At this point, we keep the **Encryption** tab option as the default selection and ready to restore our RDS instance named **pgrdscluster-snap-restore** by clicking on the **Restore DB instance** as show in [Figure 3.48](#):



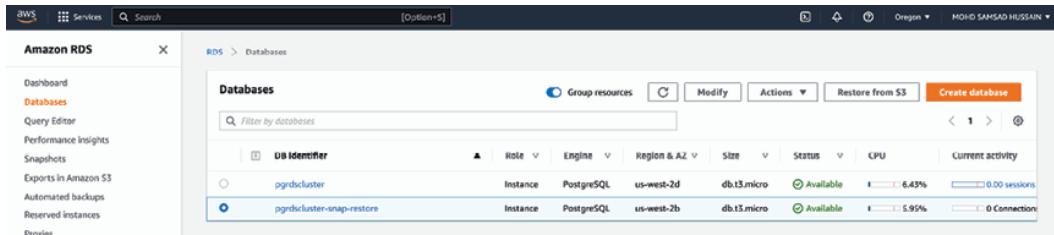
**Figure 3.48:** AWS RDS – Snapshot restore encryption setting

Refer to [Figure 3.49](#), to verify the status of the restore from RDS **Databases** dashboard:



**Figure 3.49:** AWS RDS – Snapshot restore progress

8. Reference to [Figure 3.50](#), instance is now being restored and we are now ready to connect to the PostgreSQL RDS instance named **pgrdscluster-snap-restore**. Adding a connection to the instance need a client tool, refer to the recipe *Install and configure pgAdmin client tool* of [Chapter 1](#) for detail on how to install pgAdmin tool.



**Figure 3.50:** AWS RDS – Snapshot restore succeed

- Check the tables in the **pgrdscluster-snap-restore** RDS instance, after you set-up the connection to the **pgrdscluster-snap-restore** instance using the pgAdmin tool. Execute the following script to verify the tables:

```
# Verify the table under schema books
```

```
SELECT * FROM information_schema.tables where table_schema='books';
```

The execution output of the above script is shown in following figure:

table_catalog_name	table_schema_name	table_name_name	table_type	self_referencing_column_name	reference_generation	user_defined_name
postgres	books	recovery_log	BASE TABLE	[null]	[null]	[null]

**Figure 3.51:** AWS RDS – Verify table after restore

## Recipe 24: Working with replication on AWS for PostgreSQL

PostgreSQL replication is a process of copying data from one database server to another, in order to achieve this, there should be at least two servers that can communicate with one another. The concept of replication is explained in detail in [Chapter 7, Replication and High Availability](#). Refer to the [Be introduced to Replication and High Availability](#) section.

This recipe is all about setting-up a PostgreSQL streaming replication on AWS EC2 instance. The main benefit of using streaming replication, it enables real-time replication between the master and the standby host, making the standby server more up-to-date.

The following are the prerequisite for setting-up replication:

- One Master server on AWS EC2
- One Standby server on AWS EC2

Let us begin the recipe for setting up Streaming replication with the following steps:

1. Create the first PostgreSQL database cluster on the AWS EC2 instance that we have already named **pgec2cluster**, we count this as master node of our streaming replication. Refer to the recipe *Create PostgreSQL Cloud Instance and Manage Database Connection with AWS EC2* in this chapter for detailed steps to create AWS EC2 instance.
2. Create the second PostgreSQL database cluster on the AWS EC2 instance, named **pgec2cluster\_slave**, we count this as streaming replication standby node. Refer to the recipe *Create PostgreSQL Cloud Instance and Manage Database Connection with AWS EC2* in this chapter for detailed steps to create AWS EC2 instance.
3. Now we initiate the connection to the master node using its public DNS and create the Replication user **prepuser** with a **REPLICATION** privilege using the following command:

```
# Connect to EC2 instance(Master) using SSH.
```

```
ssh -i "con_pg.pem" ec2-user@ec2-34-214-169-163.us-west-2.compute.amazonaws.com
```

```
# Enter psql terminal
```

```
psql
```

```
# Create user with REPLICATION Privileges.
```

```
CREATE USER prepuser WITH REPLICATION PASSWORD 'prepuser#5432'  
LOGIN;
```

The execution output of the above script is shown in following figure:



A screenshot of a terminal window titled 'AWS\_Pair\_key — postgres@ip-172-31-10-232:~'. The window shows the following command being run:

```
postgres=# CREATE USER prepuser WITH REPLICATION PASSWORD 'prepuser#5432' LOGIN;
CREATE ROLE
postgres=#
```

The terminal also displays system information such as the host name (mohammad@MOHAMMADS-MacBook-Pro), the ssh connection details, and the PostgreSQL version (psql (15.1)).

**Figure 3.52: AWS EC2 Replication – Create replication user**

- For the master node to accept connection from the standby node with the replication user **repuser**, can create an entry of standby node IP in the host-based authentication file **pg\_hba.conf** on **Master node**.

```
# Syntax for the entry in pg_hba.conf file for replication
host replication <replication_user> <ip_addr,netmask> <auth_method>
# Replication user entry in pg_hba.conf file
host replication repuser      172.31.8.253/32      md5
```

The execution output of the above script is shown in following figure:

A screenshot of an AWS EC2 terminal window titled "AWS\_Pair\_key". The terminal shows the command: "cat /var/lib/pgsql/15/data/pg\_hba.conf |grep -i repuser". The output displays a single line: "host replication repuser 172.31.8.253/32 md5". The terminal prompt is "[root@ip-172-31-10-232 ~]#".

**Figure 3.53:** AWS EC2 Replication - Config host-based authentication

**Note: If multiple standby nodes are used in the replication architecture, then each standby node IP entry must be made to the pg\_hba.conf file on the master node.**

- Restart the Postgres service on the **Master node** to ensure configuration changes to take effect:

```
# Restart PostgreSQL instance
systemctl restart postgresql-15
```

- At this point, the Master node has been successfully configured for replication. Now, we start configuring the standby node, but prior to that, we first initiate a connection to the Standby node and stop the Postgres service with the following command:

```
# Connect to EC2 instance (Standby) using SSH.
ssh -i "con_pg.pem" ec2-user@ec2-35-88-200-182.us-west-
2.compute.amazonaws.com
# Stop PostgreSQL instance on Standby node
systemctl stop postgresql-15
```

The execution output of the above script is shown in following figure:

```

moharram(MOHAMMAD)-MacBook-Pro AWS_Pair_key % ssh -i pgec2cluster_standby.pem ec2-user@ec2-35-88-200-182.us-west-2.compute.amazonaws.com -l 147x23
moharram(MOHAMMAD)-MacBook-Pro AWS_Pair_key % ssh -i "pgec2cluster_standby.pem" ec2-user@ec2-35-88-200-182.us-west-2.compute.amazonaws.com
Register this system with Red Hat Insights: insights-client --register
Create an account or view all your systems at https://red.ht/insights-dashboard
Last login: Wed Nov 30 09:47:11 2022 from 202.186.240.125
[ec2-user@ip-172-31-8-253 ~]$ 
[ec2-user@ip-172-31-8-253 ~]$ sudo systemctl stop postgresql-15
[ec2-user@ip-172-31-8-253 ~]$ 
[ec2-user@ip-172-31-8-253 ~]$ ls
[ec2-user@ip-172-31-8-253 ~]$ sudo systemctl status postgresql-15
● postgresql-15.service - PostgreSQL 15 database server
    Loaded: loaded (/usr/lib/systemd/system/postgresql-15.service; disabled; vendor preset: disabled)
      Active: inactive (dead)
        Docs: https://www.postgresql.org/docs/15/static/
[ec2-user@ip-172-31-8-253 ~]$ 
Nov 30 09:45:28 ip-172-31-8-253.us-west-2.compute.internal systemd[1]: Starting PostgreSQL 15 database server...
Nov 30 09:45:28 ip-172-31-8-253.us-west-2.compute.internal postmaster[18883]: 2022-11-30 09:45:27.838 UTC [18883] LOG: redirecting log output to log file
Nov 30 09:45:28 ip-172-31-8-253.us-west-2.compute.internal postmaster[18883]: 2022-11-30 09:45:27.838 UTC [18883] HINT: Future log output will be in log file
Nov 30 09:45:28 ip-172-31-8-253.us-west-2.compute.internal systemd[1]: Started PostgreSQL 15 database server.
Nov 30 09:45:30 ip-172-31-8-253.us-west-2.compute.internal systemd[1]: Stopping PostgreSQL 15 database server...
Nov 30 09:45:34 ip-172-31-8-253.us-west-2.compute.internal postgresql-15.service: Deactivated successfully.
Nov 30 09:45:34 ip-172-31-8-253.us-west-2.compute.internal systemd[1]: Stopped PostgreSQL 15 database server.
[ec2-user@ip-172-31-8-253 ~]$ 
[ec2-user@ip-172-31-8-253 ~]$ 

```

**Figure 3.54:** AWS EC2 Replication – Standby node connection

- Before moving to the next step, ensure that the data directory of the standby node is empty. Otherwise, list the data directory path and empty the directory as shown in [Figure 3.55](#):

```

[root@ip-172-31-8-253 ~]# ls /var/lib/pgsql/15/data/
base          log          pg_hba.conf   pg_multixact  pg_serial    pg_stat_tmp  pg_twophase  pg_xact           postmaster.opts
current_logfiles pg_commit_ts pg_ident.conf pg_notify    pg_snapshots pg_subtrans PG_VERSION  postgresql.auto.conf
global         pg_dynshmem  pg_logical   pg_replslot pg_stat      pg_tblspc   pg_wal        postgresql.conf
[root@ip-172-31-8-253 ~]# 
[root@ip-172-31-8-253 ~]# rm -rf /var/lib/pgsql/15/data/*
[root@ip-172-31-8-253 ~]# 
[root@ip-172-31-8-253 ~]# ls /var/lib/pgsql/15/data/
[root@ip-172-31-8-253 ~]# 
[root@ip-172-31-8-253 ~]# 

```

**Figure 3.55:** AWS EC2 Replication – Empty data directory on Standby node

- Now we initiate the **pg\_basebackup** from Standby node, this will make the exact of the Master node PostgreSQL cluster files to the Standby node.

Log into the standby node with postgres user and execute the following script to sync the database files from the primary node to the Standby node:

```

# Syntax for the pg_basebackup execution
pg_basebackup -h <Master_Node_IP> -p <Master_Port>5432 -U
<Replication_user> -D <Standby_Node_Data_Directory> -Fp -Xs -R
# Execute pg_basebackup on the Standby node
pg_basebackup -h 172.31.10.232 -p 5432 -U prepuser -D /var/lib/pgsql/15/data
-Fp -Xs -R

```

The execution output of the above script is shown in following figure:

```

AWS_Pair_key - postgres@ip-172-31-8-253:~ ssh -i pgec2cluster_standby.pem ec2-user@ec2-35-88-200-182.us-west-2.compute.amazonaws.com
[root@ip-172-31-8-253 ~]# su - postgres
Last login: Wed Nov 30 09:57:15 UTC 2022 on pts/0
[postgres@ip-172-31-8-253 ~]#
[postgres@ip-172-31-8-253 ~]$ pg_basebackup -h 172.31.10.232 -p 5432 -U prepuser -D /var/lib/pgsql/15/data/ -Fp -Xs -R
>Password:
[postgres@ip-172-31-8-253 ~]#

```

**Figure 3.56:** AWS EC2 Replication – pg\_basebackup execution on Standby node

- Verify the data directory of the Standby node once **pg\_basebackup** executed successfully as shown in [Figure 3.56](#). **pg\_basebackup** also creates a **standby.signal** file in the data directory path of the standby node, indicating that the node started as standby as shown in [Figure 3.57](#):

```

AWS_Pair_key - postgres@ip-172-31-8-253:~ ssh -i pgec2cluster_standby.pem ec2-user@ec2-35-88-200-182.us-west-2.compute.amazonaws.com
[postgres@ip-172-31-8-253 ~]#
[postgres@ip-172-31-8-253 ~]$ ls /var/lib/pgsql/15/data/
backup_label    global      pg_hba.conf   pg_notify     pg_stat      pg_twophase  postgresql.auto.conf
backup_manifest log        pg_ident.conf  pg_replslot  pg_stat_tmp  PG_VERSION  postgresql.conf
base           pg_commit_ts pg_logical    pg_serial    pg_subtrans pg_wal      standby.signal
current_logfiles pg_dynshmem pg_multixact pg_snapshots pg_tblspc  pg_xact
[postgres@ip-172-31-8-253 ~]#

```

**Figure 3.57:** AWS EC2 Replication – Verify data directory sync

- Finally, get ready to start the replication by starting the PostgreSQL database cluster service using the following script:

# Start the PostgreSQL instance on Standby node

systemctl restart postgresql-15

The execution output of the above script is shown in following figure:

```

AWS_Pair_key - root@ip-172-31-8-253:~ ssh -i pgec2cluster_standby.pem ec2-user@ec2-35-88-200-182.us-west-2.compute.amazonaws.com - 1...
[root@ip-172-31-8-253 ~]#
[root@ip-172-31-8-253 ~]# systemctl start postgresql-15
[root@ip-172-31-8-253 ~]#
[root@ip-172-31-8-253 ~]# systemctl status postgresql-15
● postgresql-15.service - PostgreSQL 15 database server
   Loaded: loaded (/usr/lib/systemd/system/postgresql-15.service; disabled; vendor preset: disabled)
   Active: active (running) since Wed 2022-11-30 10:07:17 UTC; 6s ago
     Docs: https://www.postgresql.org/docs/15/static/
  Process: 11573 ExecStartPre=/usr/pgsql-15/bin/postgresql-15-check-db-dir ${PGDATA} (code=exited, status=0/SUCCESS)
 Main PID: 11578 (postmaster)
   Tasks: 6 (limit: 6719)
    Memory: 16.9M
      CPU: 52ms
     CGroup: /system.slice/postgresql-15.service
             └─11578 /usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/15/data/
                  ├─11579 "postgres: logger"
                  ├─11580 "postgres: checkpointer"
                  ├─11581 "postgres: background writer"
                  ├─11582 "postgres: startup recovering 000000010000000000000003"
                  └─11583 "postgres: walreceiver streaming 0/3000148"

Nov 30 10:07:17 ip-172-31-8-253.us-west-2.compute.internal systemd[1]: Starting PostgreSQL 15 database server...
Nov 30 10:07:17 ip-172-31-8-253.us-west-2.compute.internal postmaster[11578]: 2022-11-30 10:07:17.217 UTC [11578] LOG:  redirecting log output to logging collector process
Nov 30 10:07:17 ip-172-31-8-253.us-west-2.compute.internal postmaster[11578]: 2022-11-30 10:07:17.217 UTC [11578] HINT: Future log
Nov 30 10:07:17 ip-172-31-8-253.us-west-2.compute.internal systemd[1]: Started PostgreSQL 15 database server.
[root@ip-172-31-8-253 ~]#

```

**Figure 3.58:** AWS EC2 Replication – Start database service

11. With this, we have successfully configure the streaming replication for PostgreSQL database cluster on AWS EC2 instance, the following script will get the replication status from Master and Standby node:

```
# Check Replication Status from Master node from psql shell
```

```
select username as REPLICATION_USER, application_name as APP_NAME,
client_addr as STANDBY_IP, state, sync_state from pg_stat_replication;
```

The execution output of the above script is shown in following figure:

```
AWS_Pair_Key - postgres@ip-172-31-10-232:~ ssh -i con_pg.pem ec2-user@ec2-34-214-169-163.us-west-2.compute.amazonaws.com -t
[postgres@ip-172-31-10-232 ~]$ psql (15.1)
Type "help" for help.

postgres=# select username as REPLICATION_USER, application_name as APP_NAME, client_addr as STANDBY_IP, state, sync_state from pg_stat_replication;
 replication_user | app_name | standby_ip | state | sync_state
 prepuser          | walreceiver | 172.31.8.253 | streaming | async
(1 row)

postgres=#

```

**Figure 3.59:** AWS EC2 Replication – Verify replication status on Master node

12. The subsequent step involves verifying the replication status directly from the standby node's **psql** shell. By executing the following SQL query:

```
# Check Replication Status from Standby node from psql shell
```

```
SELECT STATUS, last_msg_send_time, last_msg_receipt_time, slot_name,
sender_host as master_ip FROM pg_stat_wal_receiver;
```

In [Figure 3.60](#), the execution of above query provides information about the replication status, message transmission and receipt times, designated slot name, and the IP address of the master node. Monitoring these details ensures a real-time assessment of the replication's health and progress, enabling informed decision-making to ensure data integrity and system reliability.

```
AWS_Pair_Key - postgres@ip-172-31-8-253:~ ssh -i pgec2cluster_standby.pem ec2-user@ec2-35-88-200-182.us-west-2.compute.amazonaws.com...
[postgres@ip-172-31-8-253 ~]$ psql (15.1)
Type "help" for help.

postgres=# SELECT status, last_msg_send_time, last_msg_receipt_time, slot_name, sender_host as master_ip FROM pg_stat_wal_receiver;
 status | last_msg_send_time | last_msg_receipt_time | slot_name | master_ip
 streaming | 2022-11-30 10:39:54.218915+00 | 2022-11-30 10:39:54.21912+00 | | 172.31.10.232
(1 row)

postgres=#

```

**Figure 3.60:** AWS EC2 Replication – Verify replication status on Standby node

## Conclusion

In conclusion, this chapter has provided an in-depth exploration of cloud provisioning for PostgreSQL 15 within the AWS landscape. With an understanding of various deployment options, native backup strategies, administration techniques, and advanced replication mechanisms, you are now equipped to embark on a journey of effectively managing PostgreSQL instances in the cloud. By adhering to best practices and leveraging the power of the cloud, you can ensure optimal performance, scalability, and data integrity while reaping the rewards of a seamlessly integrated PostgreSQL ecosystem. As you delve further into the realm of cloud provisioning, you will discover the limitless possibilities and transformative potential that PostgreSQL on the cloud has to offer.

In the next chapter, we will look at the methodology for migrating databases and its implementation between on-premises and cloud infrastructure.

### **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

# Migration

## Introduction

Migrating databases is about moving your data from one platform to another. These platforms can be an operating system, a database, or a combination thereof. There are several reasons why an organization may choose to migrate databases. For instance, an organization may feel that a particular database provides more benefits than its existing database, besides cutting costs by moving its existing system onsite into the cloud.

Generally, database migration in PostgreSQL is a multi-phase procedure that requires preparing the source instance and databases, setting source and target instances, and network connectivity, along with any extra parameters required to set up a migration.

## Structure

In this chapter, we will cover the following topics:

- Introducing migration for PostgreSQL
- Exploring migration methodology

- Database migration roadmap
- Understanding PostgreSQL database system migration types

## Objectives

This chapter aims to explain how to migrate a database from multiple distributions to the PostgreSQL instance. This chapter also covers the detailed implementation of the migration tool and the scenario where it can be used.

## Introduction to migration for PostgreSQL

Database migration is the process of moving data from the source to a target database system or systems. Based on operational requirements, database migration may be partial or at the database level with each granular change.

The migration methodology and its complexity depend on the source and target database, but it should also be adapted to the new system's objective. Essentially, PostgreSQL database migration refers to the movement or migration of the multi-source database to the PostgreSQL database system.

Furthermore, data migration sometimes refers to the upgrading of data to the same data source, this often involves data validation, fixing issues in the source or during transport, and converting data formats.

Additionally, there are several database migration methods that may or may not be suitable for migration purposes. The selection of an appropriate migration method depends upon numerous factors and requirements.

The selection of migration method depends on elements such as data volume, complexity, downtime tolerance, resource availability, and the desired outcome.

In essence, the selection of an appropriate migration method is a strategic decision that shapes the overall success of the migration process. Taking into account the multifaceted considerations and requirements of the migration project ensures a well-planned and effective transition to the PostgreSQL database system.

## Exploring migration methodology

The database migration methodology means an organized phase where the migration task will be performed. Each migration phase is a specific scope that defines the appropriate method and steps for the database migration project. Please refer to the following figure:



**Figure 4.1:** Database migration phase

The database migration methodology includes the following phases, as shown in

**Figure 4.1:**

**Project scope:** A draft plan that sets out migration requirements, deliverables, domain expertise, and budgeting.

**Analysis phase:** Assessment of the database migration process, determination of the proper migration methods, and landscape analysis.

**Planning phase:** Perform advanced analysis and evaluation of both source and target systems and write a flexible schedule for migration.

**Execution phase:** Execute the database migration plan, the data is extracted, transformed, and loaded on the target, from the source system.

**Validation phase:** Validate the transferred data and perform end-to-end tests that assure data quality between the source and target database system.

## **Understanding PostgreSQL database system migration types**

In the realm of database system migration, a comprehensive understanding of its types is crucial. The PostgreSQL ecosystem offers two primary migration categories: Homogeneous database system migration and heterogeneous database system migration.

**Homogeneous database system migration** involves the migration of data between instances of the same database system. This approach is particularly useful when upgrading to a newer version of PostgreSQL or when migrating data within the same version. The compatibility between source and target systems simplifies the process, allowing for a relatively straightforward transfer of schema and data. This type of migration often entails upgrading hardware, optimizing performance, or transitioning to a more robust infrastructure.

On the other hand, **heterogeneous database system migration** revolves around migrating data between instances of different database systems. This type of migration is employed when transitioning from a non-PostgreSQL database, such as Oracle or MySQL, to PostgreSQL. While more complex due to differences in data types, syntax, and functionality between systems, this approach provides an opportunity to harness PostgreSQL's advanced features and capabilities.

Understanding these migration types is paramount in shaping an effective migration strategy. Tailoring the approach to the unique characteristics of each type ensures a successful and seamless transition, minimizing disruptions

and data loss. By selecting the appropriate migration type based on the specific requirements of the project, organizations can optimize their migration process and harness the full potential of PostgreSQL's capabilities.

## Recipe 25: On-premise to AWS EC2 instance migration

Migrating your on-premise PostgreSQL 15 database to an AWS EC2/RDS instance can provide scalability, reliability, and cost-efficiency benefits. This recipe provides detailed steps to perform the migration successfully.

Prerequisites:

- An AWS account with sufficient permissions.
- PostgreSQL 15 installed on the on-premise server.
- AWS EC2 instance created and configured appropriately.

### 1. Configure target AWS EC2 instance:

- a. The first step is to install and configure PostgreSQL database instance on the destination EC2 server instance, refer to [\*Chapter 3, Manage PostgreSQL Instance with AWS EC2 from Cloud Provisioning\*](#) for instructions on how to setup PostgreSQL instance on EC2.

### 2. Backup on-premise database:

- a. Connect to your on-premise PostgreSQL server using **psql** or any other PostgreSQL client and run the following command to create a full backup of on-premise PostgreSQL database:

```
pg_dump -U postgres -d pgs -F p -f
```

```
Logical_Bkp_202305.sql
```

The above command connects to the database named **pgs** using the username **postgres** and creates a logical backup in a file named **Logical\_Bkp.sql**.

3. Restore database backup to EC2:

- Once your EC2 instance is running, connect to it using its public DNS.

**# Connect to EC2 instance using SSH.**

```
ssh -i "con_pg.pem" ec2-user@ec2-54-244-149-254.us-west-  
2.compute.amazonaws.com
```

**Note: Refer to the recipe manage PostgreSQL instance with AWS EC2 from Chapter 3, Cloud Provisioning, for instructions on how to access the PostgreSQL instance on EC2.**

- Transfer the database backup file from your on-premise server to the EC2 instance using **scp** or any other file transfer method.
- Use the **pg\_restore** command to restore the backup on the EC2 instance; Before initiating the restore process, stop the PostgreSQL service to prevent conflicts or data corruption during the restore operation. Execute the following command to stop the PostgreSQL service:

```
systemctl stop postgresql-15.service
```

- Open a terminal or command prompt and run the following command to restore the database from the logical backup file.

```
pg_restore -U postgres -Fc -d pgs Logical_Bkp_202305.sql
```

The **pg\_restore** utility will execute the SQL commands from the backup file and restore the database to its original state.

- e. Start the PostgreSQL service using the following command and this will bring the database back online after the restore process.

```
systemctl start postgresql-15.service
```

**Note: During the migration of PostgreSQL from an on-premise server to an EC2 instance, it is common practice to use a logical backup for the restoration process. Logical backups offer a flexible and consistent way to transfer data, schema, and configurations. However, it is important to consider the impact of in-flight transactions that may occur on the on-premise PostgreSQL database after the logical backup has been taken.**

Any transactions initiated on the source (on-premise) database after the logical backup but before the final cutover to the target (EC2 instance) database will not be captured in the backup. These in-flight transactions might result in data inconsistencies between the source and target databases. To minimize this impact, it is advisable to plan the migration during a low-activity period or during the complete downtime window, notify users about the migration, and take necessary precautions to minimize new transactions during the migration cutover window. Additionally, consider using replication or other strategies to synchronize any missed transactions before making the switch to the new database.

4. Test the migration:
  - a. Connect to the EC2 instance using any PostgreSQL client to ensure it is accessible and data is intact.
5. Decommission on-premise database:
  - a. Once confident in the EC2 instance's stability

- and performance, decommission on-premise PostgreSQL server.
- b. Ensure to have all necessary backups and archived data as needed before decommissioning.

## Database migration roadmap

The database migration roadmap for PostgreSQL 15 serves as a vital compass for organizations looking to transition to this powerful open-source database system. This roadmap offers a structured approach, beginning with thorough assessment and planning, followed by the selection of appropriate migration tools and strategies tailored to PostgreSQL 15's unique features and improvements. It covers data schema conversion, data transfer, and application adjustments, all while ensuring minimal downtime and data integrity. This invaluable resource helps businesses and database administrators navigate the migration process seamlessly, unlocking the full potential of PostgreSQL 15 for their data management needs.

## Recipe 26: Migrating PostgreSQL from EC2 to RDS instance on AWS

Migrating a PostgreSQL database from an **Amazon Elastic Compute Cloud (EC2)** instance to an Amazon **Relational Database Service (RDS)** instance can offer benefits such as managed operations, automated backups, and scalability. In this recipe, we will guide you through the process of migrating a PostgreSQL 15 database from an EC2 instance to an RDS instance on AWS. This recipe assumes you have a PostgreSQL database running on an EC2 instance and an RDS instance provisioned and ready.

Here is the architecture of the source and target systems used in this recipe.

Detail	Source Instance	Target Instance
Hosted environment	AWS EC2 instance	AWS RDS instance
Operating system	RHEL-9	RHEL-9
Database version	PostgreSQL 14.5	PostgreSQL 14.5

**Table 4.1:** architecture of the source and target systems

In our previous recipe *Native Backup or Restore with AWS EC2 Instance* from [Chapter 3, Cloud Provisioning](#), we utilized native tools such as **pg\_dump** and **pg\_restore** for the migration of on-premises database. However, in this specific recipe, we are choosing to utilize the AWS database migration service (AWS DMS) to facilitate the migration of our database to the cloud. This will be achieved by following the steps outlined below:

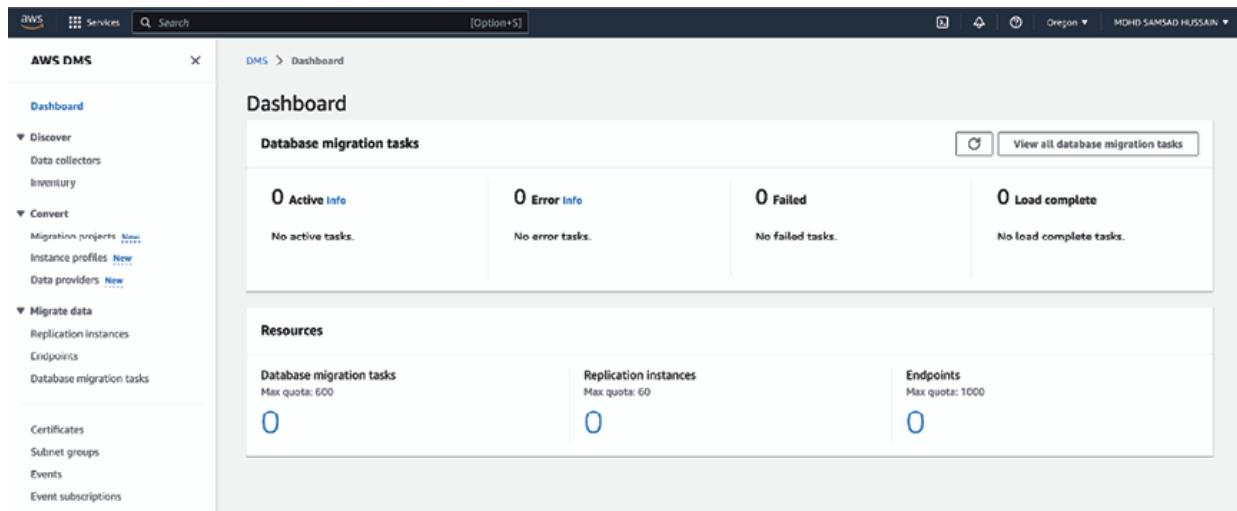
1. First, let us assume that our PostgreSQL cluster instance is running on an AWS EC2 instance on a source system.

```
# Verify PostgreSQL status on source system  
Systemctl status postgresql-14
```

2. The next step is to install and configure the target database system. AWS RDS for PostgreSQL is our target database system. Refer to the recipe *Create PostgreSQL cloud instance and manage database connection with AWS RDS*, [Chapter 3, Cloud Provisioning](#) for instructions on how to set up a PostgreSQL instance on AWS RDS.
3. Since the source and target database system is running, the next step is to configure the AWS DMS. The first step to configure the AWS DMS is to create a replication instance that is responsible to host the

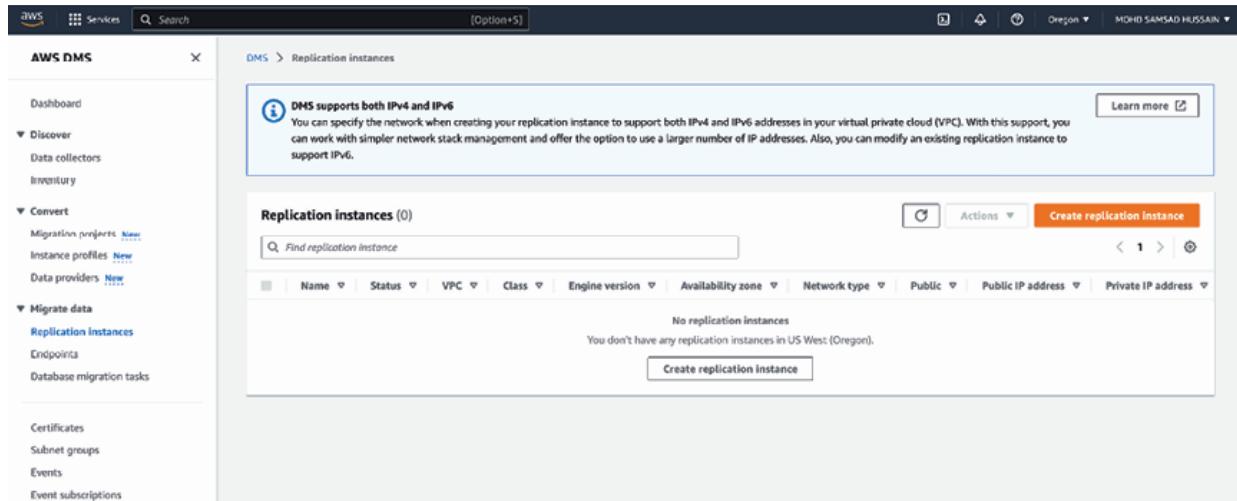
replication task in the AWS DMS, by following the step below:

- a. From the AWS console home page, click on the **Services** tab which lists all services. All we need to do is select the **Database migration service** within the **Migration & transfer** service category, which will take us to the **AWS DMS** console dashboard, as seen in *Figure 4.2*:



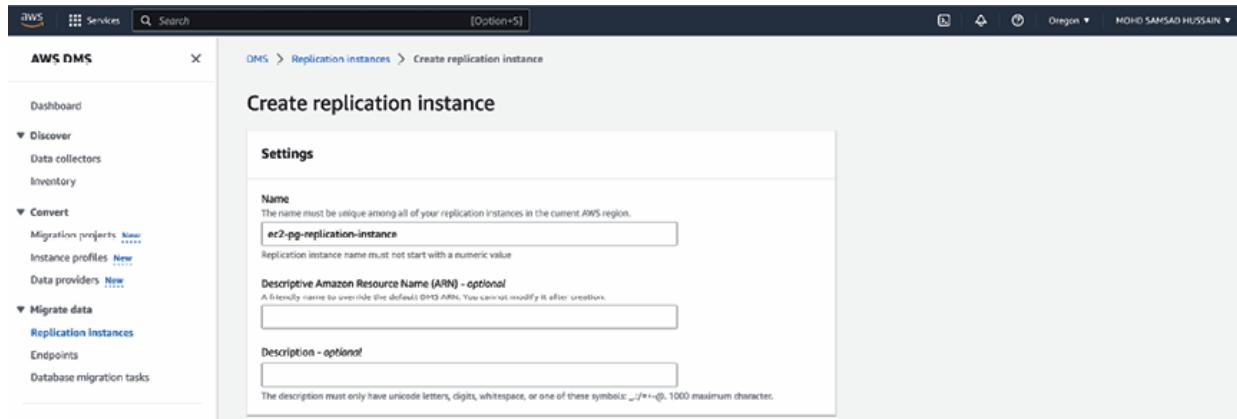
**Figure 4.2:** AWS DMS – Dashboard

- b. From the AWS DMS **Dashboard**, under the **Migrate** data tab, choose **Replication instances** and then click **Create replication instance** from the **Replication instances** tab (*Figure 4.3*):



**Figure 4.3:** AWS DMS – Replication instance dashboard

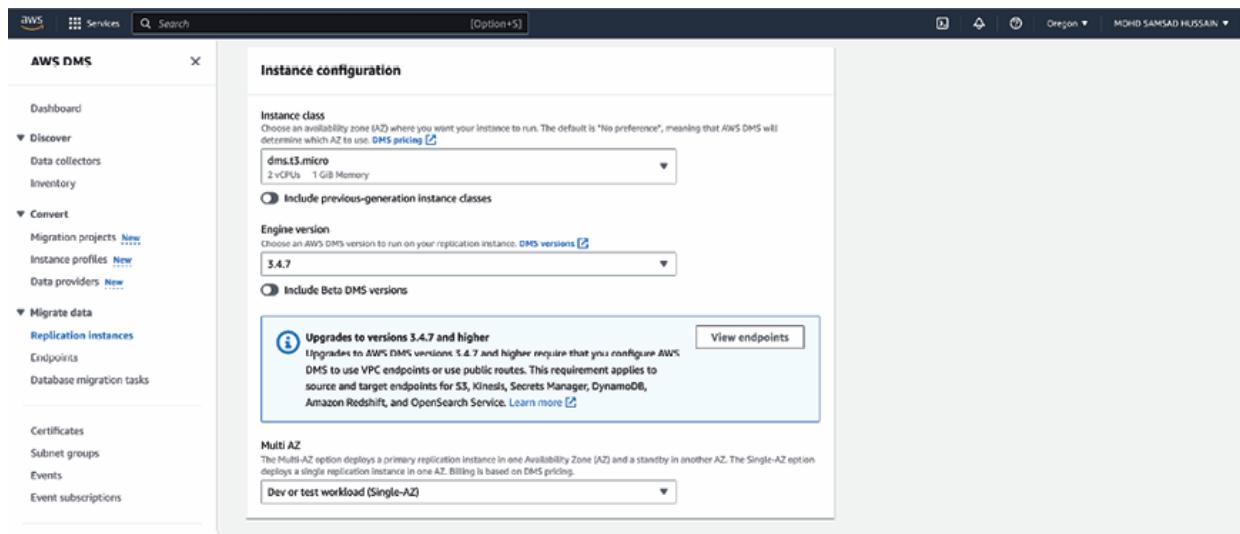
- c. The **Create replication instances** opens a new page that lets you type the following:
  - i. Enter the name of your replication instance into the **Name** field under the **Settings** section, for this recipe we use the name **ec2-pg-replication-instance** as seen in [\*\*Figure 4.4\*\*](#):



**Figure 4.4:** AWS DMS – Create replication instance

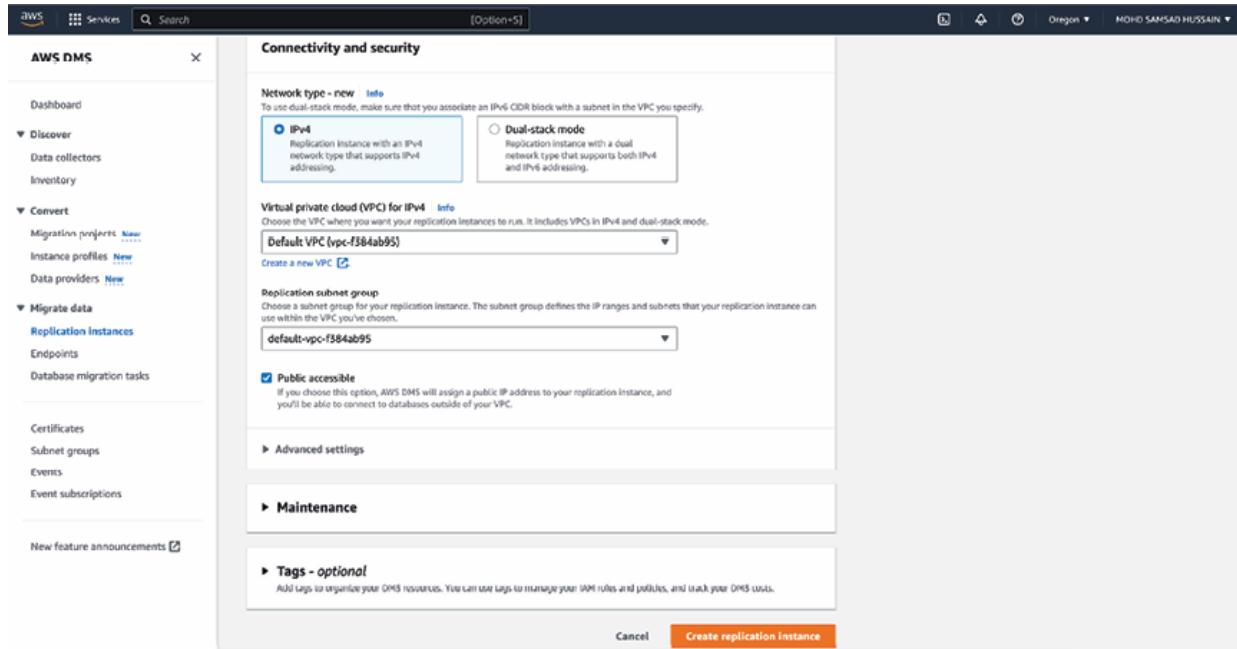
- ii. In the **Instance configuration** section, select the appropriate options for the **Instance**

**class, engine version** and **Multi AZ (availability zone)** according to the infrastructure requirements. For this recipe, we used **Instance class** as **dms.t3.micro**, **Engine version** as 3.4.7 and **Multi AZ** as **Dev or test load (single-AZ)** as shown in *Figure 4.5*:



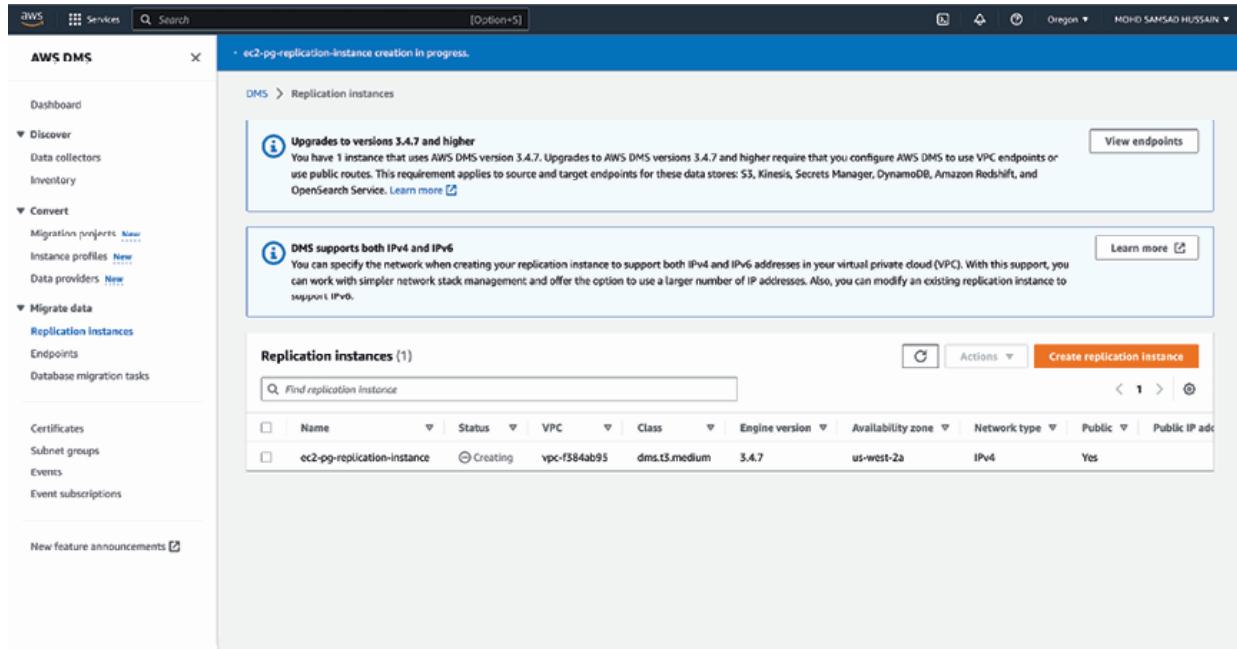
**Figure 4.5:** AWS DMS – Replication instance configuration

- iii. In the **Storage** section, we selected 50GB as the size for allocated storage.
- iv. In the **Connectivity and security** section, we keep the default settings and click the **Create replication instance** as seen in *Figure 4.6*:



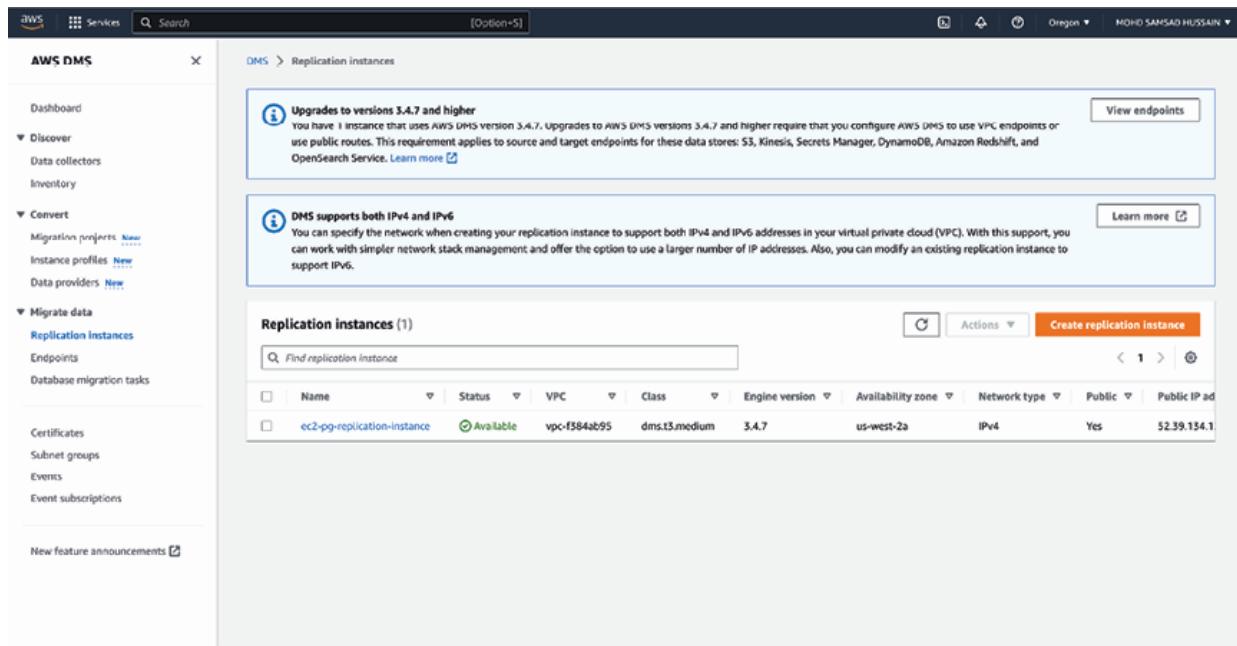
**Figure 4.6:** AWS DMS – Replication instance connectivity & security

v. Finally, click on the **Create replication instance** that will begin with creating a replication instance. As shown in *Figure 4.7*, the **Status** column displays **Creating**, indicating that the creation of the instance is in progress. Please refer to the following figure:



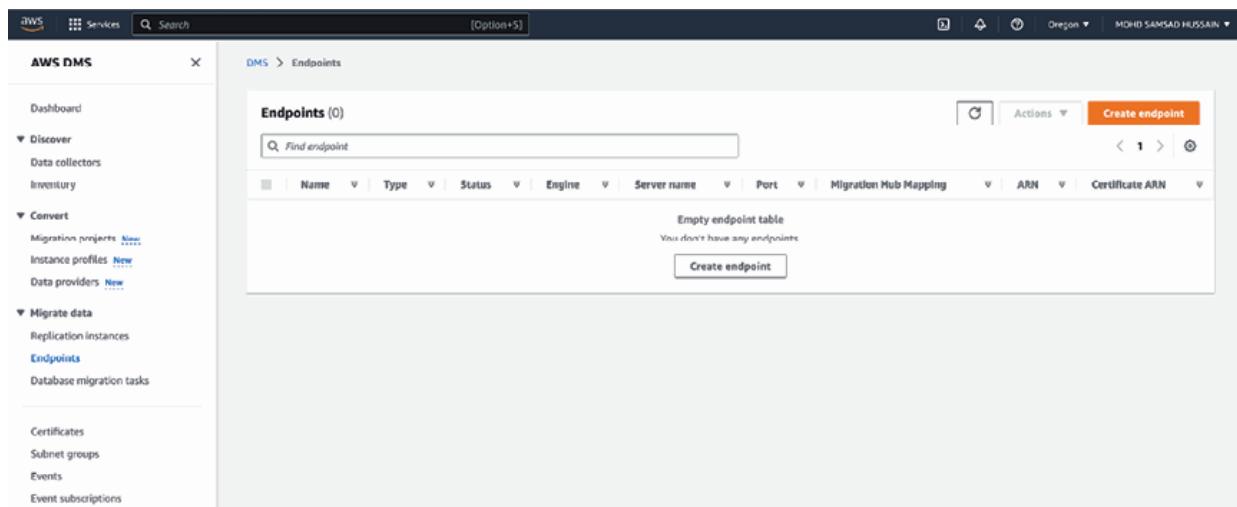
**Figure 4.7: AWS DMS – Replication instance create status**

vi. At that stage, the replication instance was successfully created, as shown in [Figure 4.8](#):



**Figure 4.8: AWS DMS – Active replication instance**

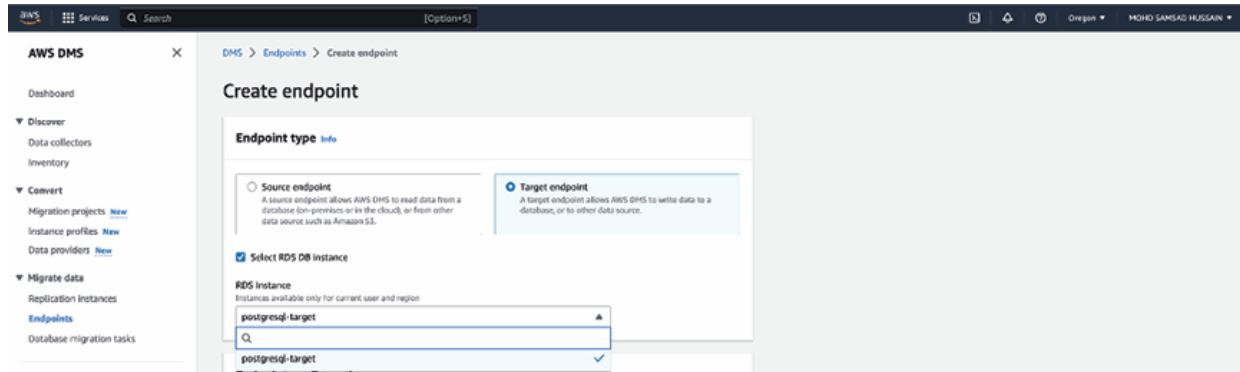
4. Since we have successfully created the replication instance for AWS DMS. The next step in the AWS DMS configuration is to configure the source server and target server endpoints. The configuration of endpoints is essentially a source and target database connection information, which enables you to connect the source and target database from AWS DMS. The following steps will setup the endpoint for the target system.
- a. In the **AWS DMS** console, select **Endpoints** under the **Migrate data** tab and then click **Create endpoint** under the **Endpoints** tab as shown in *Figure 4.9*:



*Figure 4.9: AWS DMS - Endpoint dashboard*

- b. The **Create endpoint** option opens a new page which allows you to enter the following:
- Select the **Target endpoint**, check the box to select the **RDS DB instance**, and choose the target **RDS database** from the drop-down list beneath the RDS instance section. As shown in *Figure 4.10*, for our recipe, we have created a target RDS instance named

## postgresql-target:



**Figure 4.10:** AWS DMS - Create target endpoint

ii. As shown in [Figure 4.11](#), under the **Endpoint configuration tab**:

Type the name of the target database endpoint into the **Endpoint identifier** section.

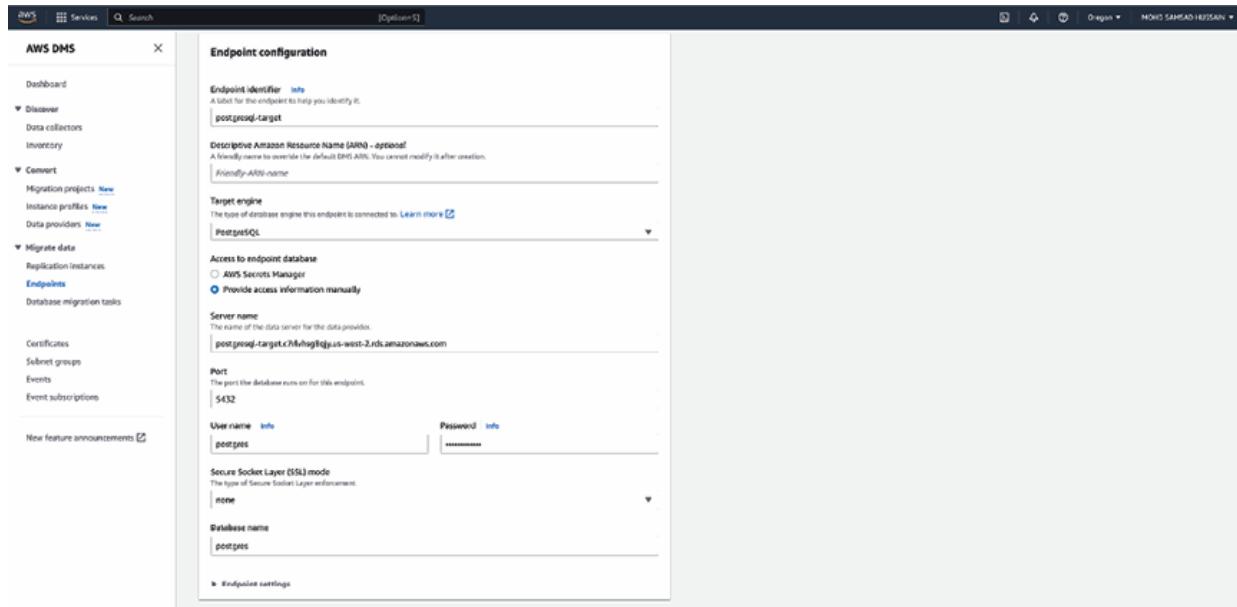
Select the engine as **PostgreSQL** from the drop-down list under the **Target engine** section.

Select the option **Provide access information manually** under the **Access to endpoint database** section.

Under the **Server name** section, enter the target database endpoint address where our PostgreSQL database is running.

Enter the **Port no**, **User name** and **Password** for the target database.

Enter the **target database name** under the **Database name** section.



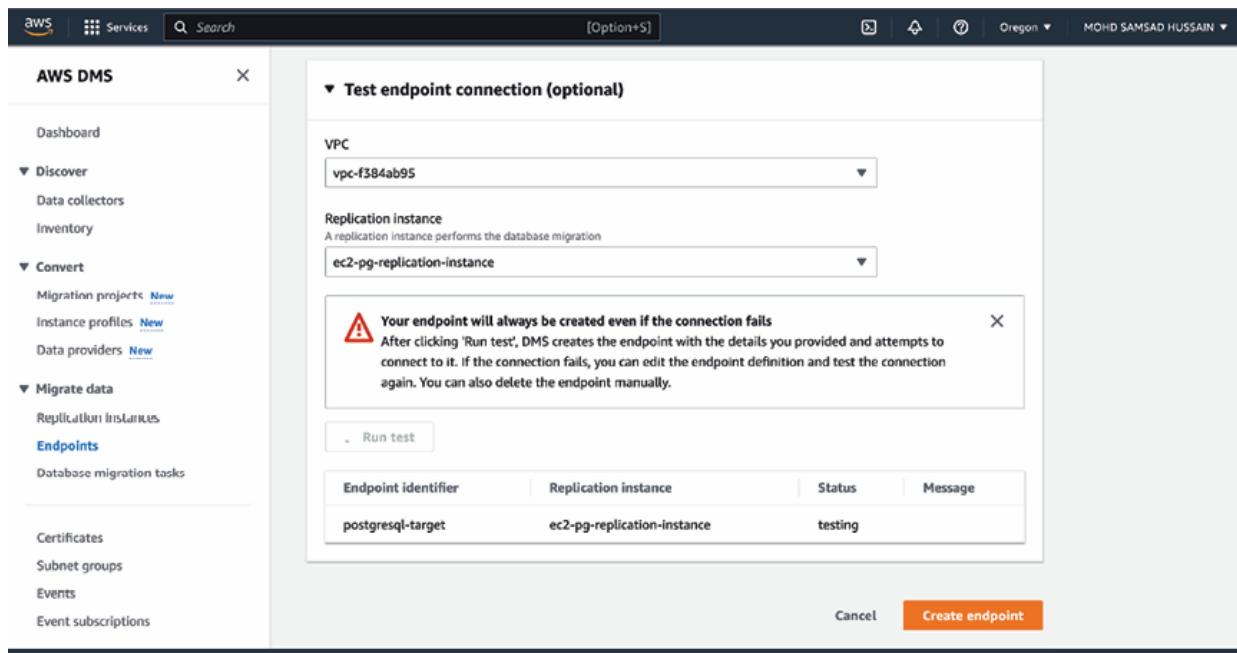
**Figure 4.11:** AWS DMS – Target endpoint configuration

We keep the **KMS key** and **Tag** tab to the default configuration.

As shown in [Figure 4.12](#), under the **Test endpoint connection** tab, click **Run test** setup with the following options.

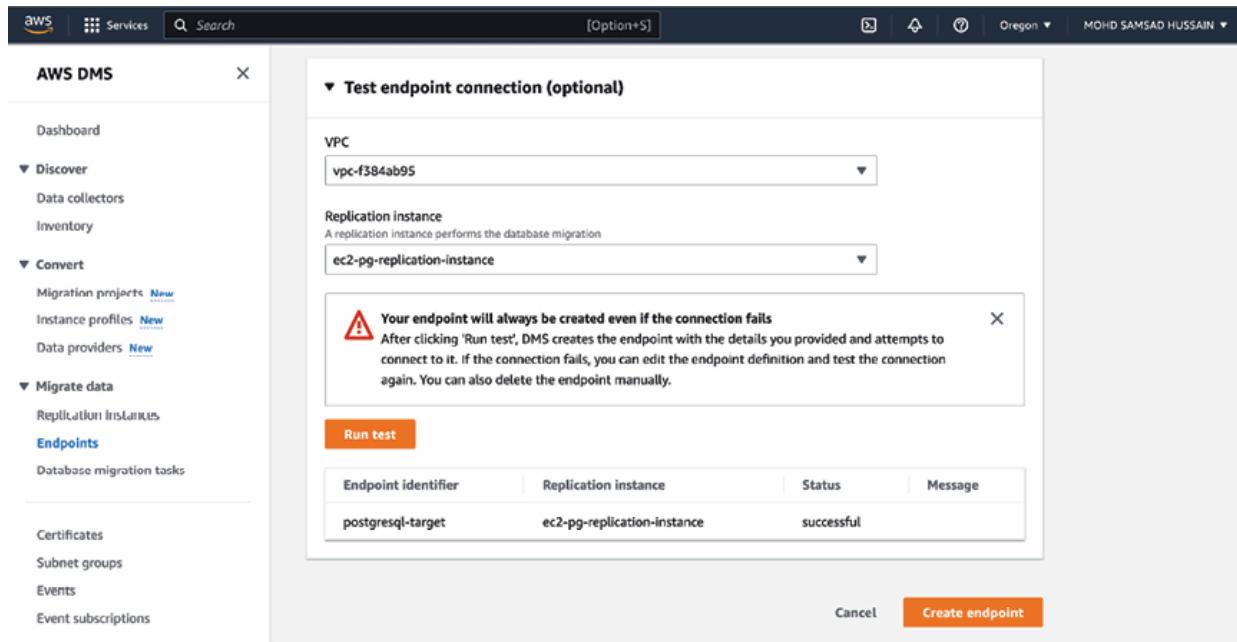
Select the appropriate **VPC** from the drop-down list under the **VPC** section.

Choose the replication instance from the drop-down list under the **Replication instance** section. (We chose our replicate instance name as **ec2-pg-replication-instance** which we created in a previous step of this recipe). Please refer to the following figure:



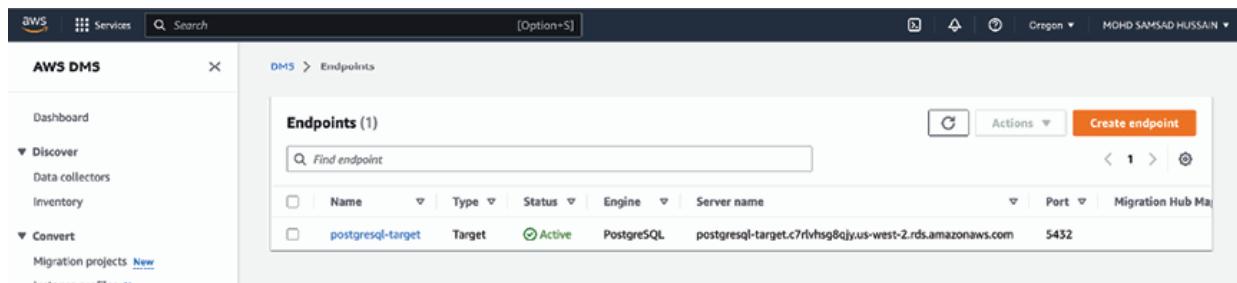
**Figure 4.12:** AWS DMS – Target endpoint connection test

- a. As shown in [Figure 4.13](#), test result is successful. It means that our replication instance is qualified to connect to the target database instance. Please refer to the following figure:



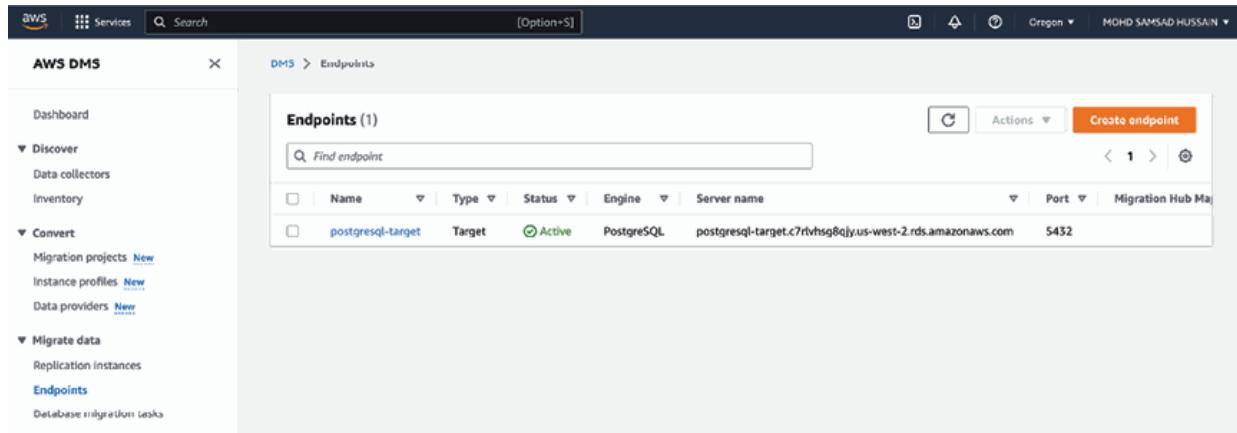
**Figure 4.13:** AWS DMS – Target endpoint connection test result

- b. In the next step, click **Create endpoint** to save the endpoint settings of the target database. Referring to the *Figure 4.14*, **Target** database endpoint has been created successfully.



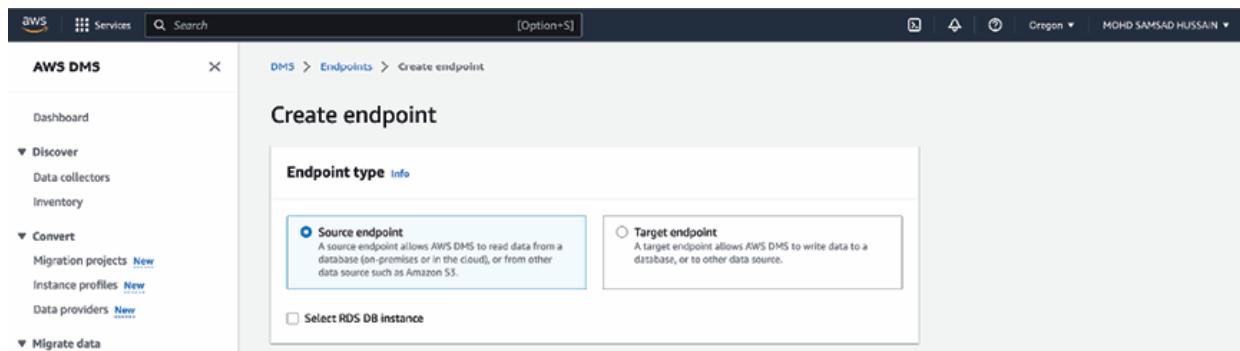
**Figure 4.14:** AWS DMS – Endpoint dashboard

5. Since we have successfully created the endpoints for the target database system. The next step in the AWS DMS configuration is to configure the source server endpoints. The following steps will setup the endpoint for the source system:
- a. In the **AWS DMS** console, select Endpoints under the **Migrate data** tab and then click **Create endpoint** under the **Endpoints** tab as seen in *Figure 4.15*:



**Figure 4.15:** AWS DMS - List endpoint

- b. The **Create endpoint** option opens a new page which allows you to enter the following:
  - i. Select the **Source endpoint**, under the **Endpoint type** section (*Figure 4.16*):



**Figure 4.16:** AWS DMS - Create source endpoint

- ii. As shown in *Figure 4.17*, under the Endpoint configuration tab:

Type the name of the source database endpoint into the **Endpoint identifier** section.

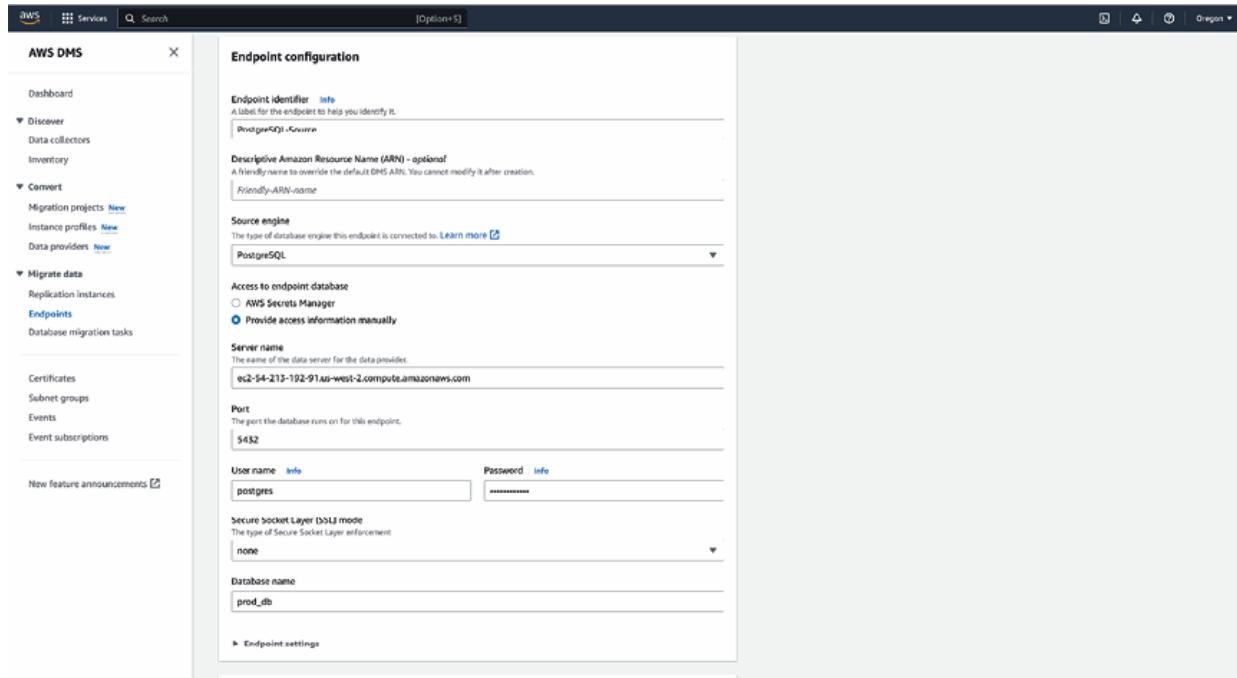
Select the engine as **PostgreSQL** from the drop-down list under the **Source engine** section.

Select the option **Provide access information manually** under the **Access to endpoint database** section.

Under the **Server name** section, enter the source database endpoint address where our PostgreSQL database is running.

Enter the **Port no**, **User name** and **Password** for the source database.

Enter the **Source database name** under the **Database name** section.

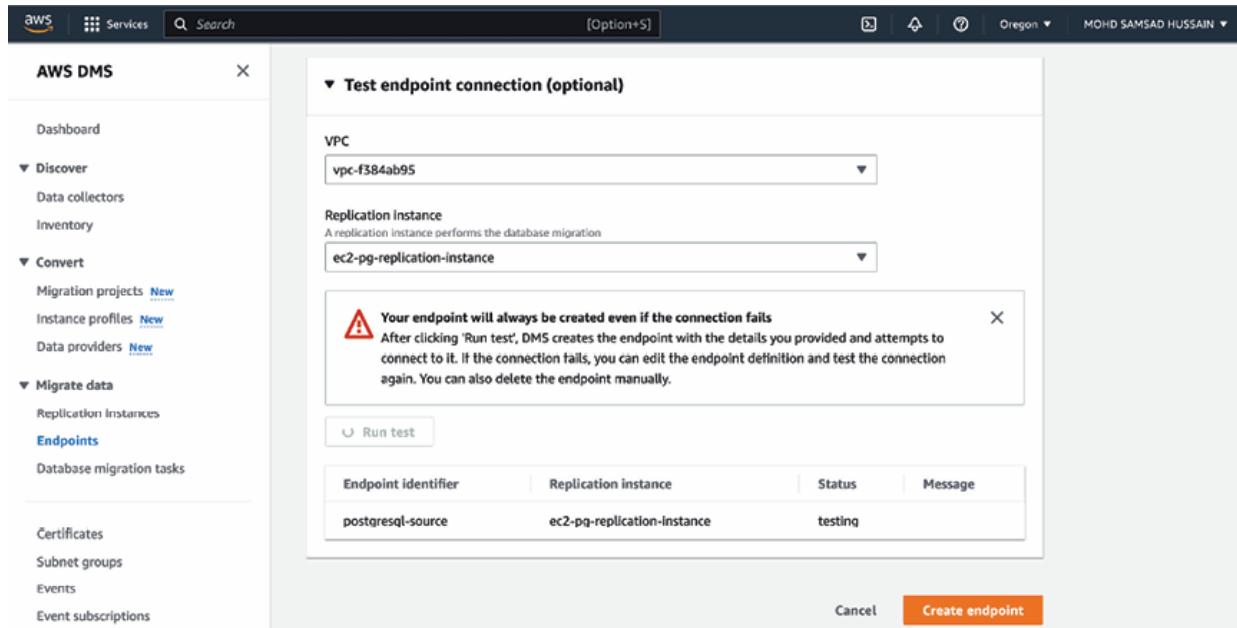


**Figure 4.17:** Source endpoint configuration

- iii. We keep the **KMS** key and **Tag** tab to the default configuration.
- iv. As shown in [Figure 4.18](#), under the **Test endpoint connection** tab, click **Run test** setup with the following options.

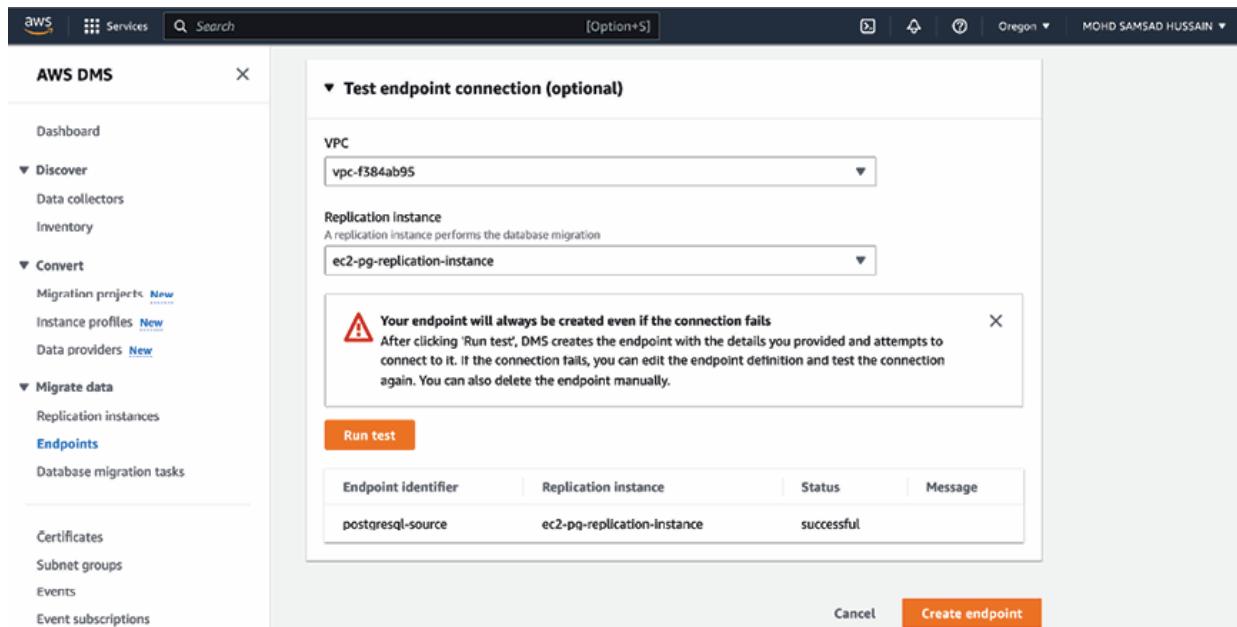
Select the appropriate **VPC** from the drop-down list under the **VPC** section.

Choose the replication instance from the drop-down list under the **Replication instance** section. (We chose our replicate instance name as **ec2-pg-replication-instance** which we created in an earlier step of this recipe). Please refer to the following figure:



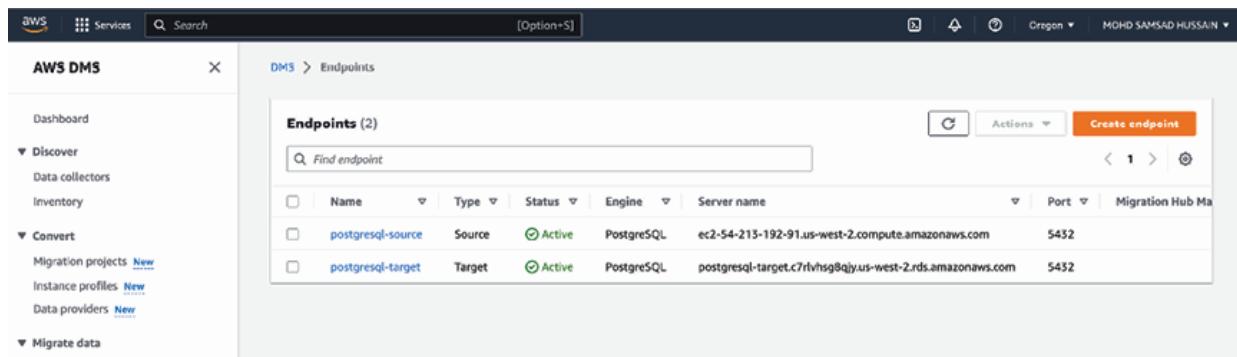
**Figure 4.18:** AWS DMS – Source endpoint connection test

- As shown in [Figure 4.19](#), test result is successful. It means that our replication instance is qualified to connect to the source database instance. In the next step, select **Create endpoint** to create the source endpoint.



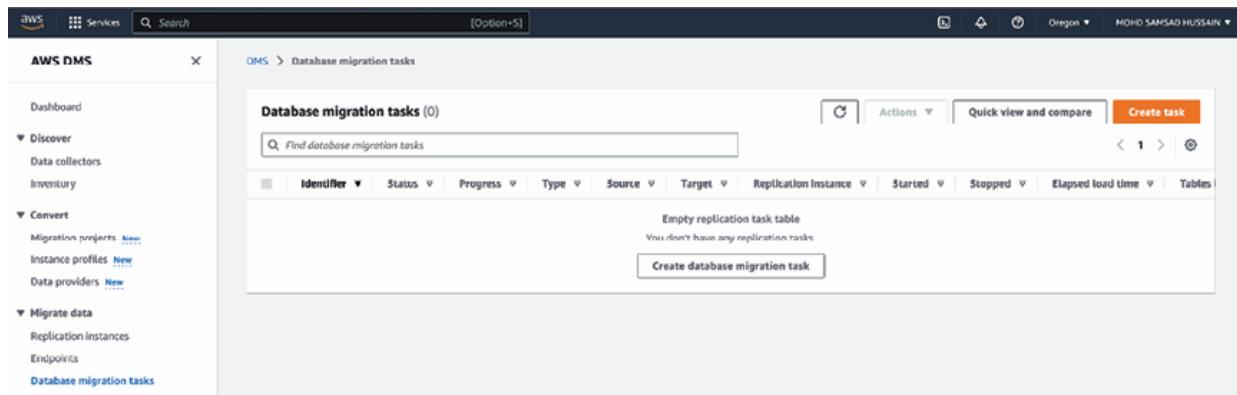
**Figure 4.19:** AWS DMS - Source endpoint connection test result

- b. In the next step, click **Create endpoint** to save the endpoint settings of the source database. Refer to the *Figure 4.20*, Source database endpoint created successfully.



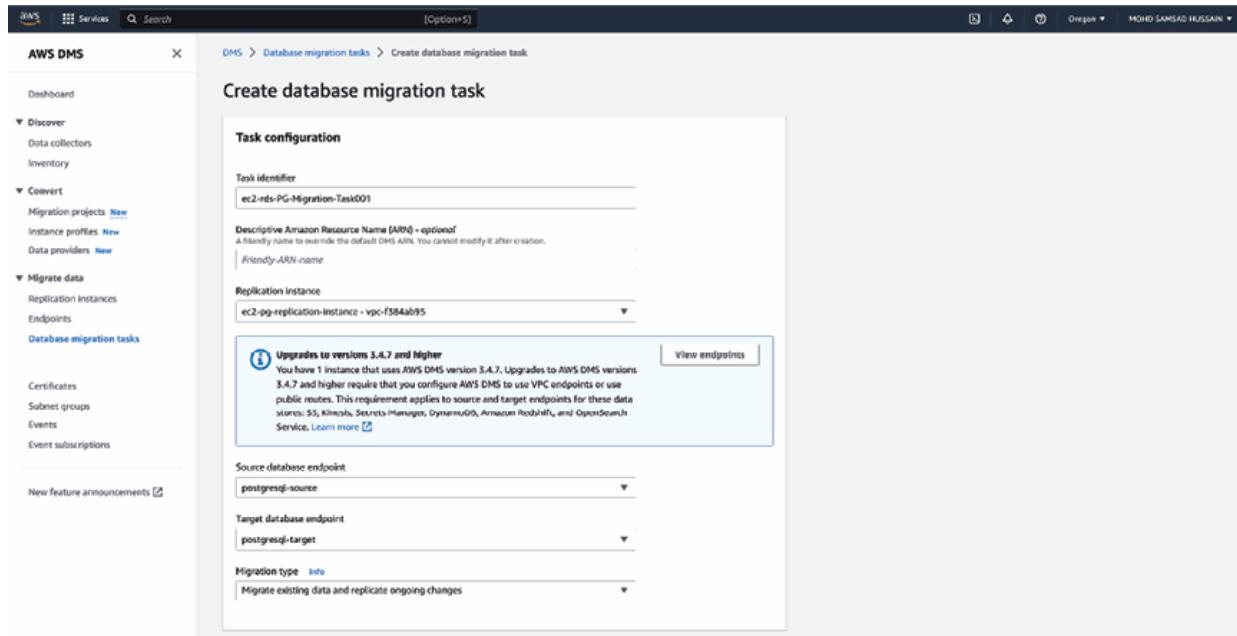
**Figure 4.20:** AWS DMS – endpoint list

6. The next step is to create and configure a database migration task, which is responsible for the database migration between the source and target database system.
    - a. In the **AWS DMS** console, under the **Migrate data** tab, select **Database migration tasks** and click **Create task** from the **Database migration tasks** tab (*Figure 4.21*):



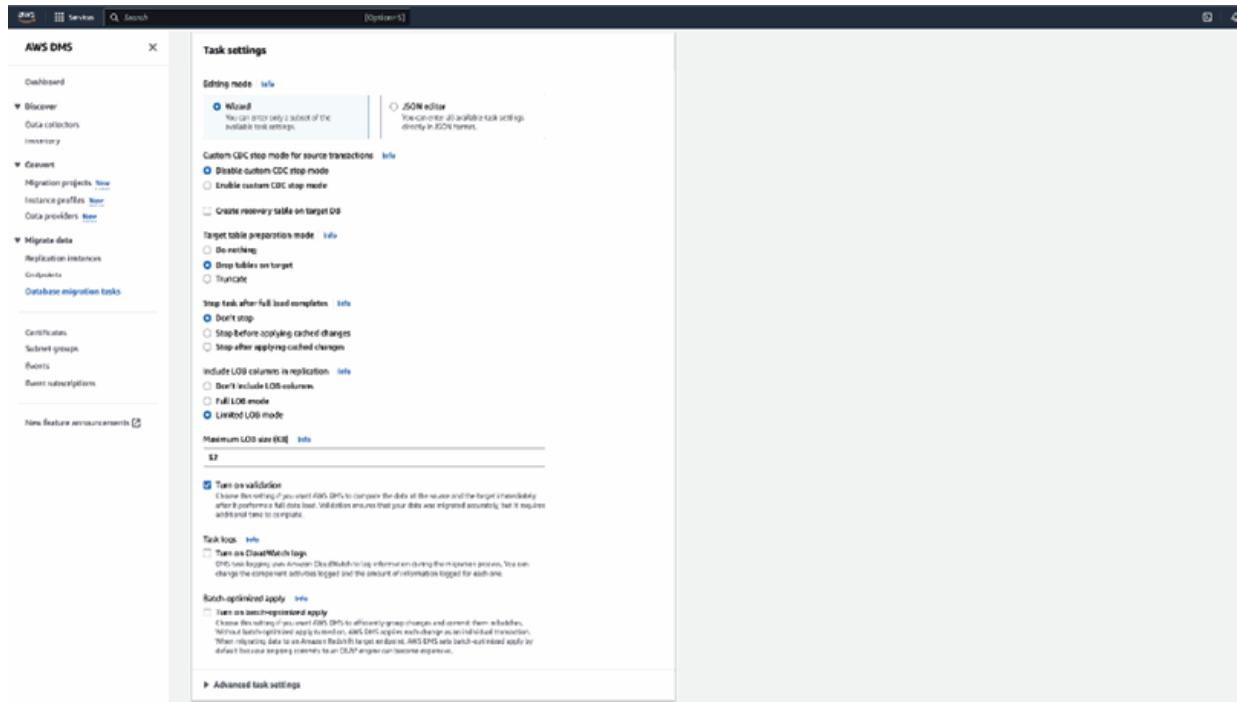
**Figure 4.21:** AWS DMS - Database migration task dashboard

- b. The **Create task** option opens a new page that lets you enter the following on the **Task configuration** tab.
  - i. Type the name of the task in the **Task identifier** section. As shown in [Figure 4.22](#), we selected the task identifier name **ec2-rds-pg-migration-task001**.
  - ii. Select the replication instance like **ec2-pg-replication-instance** from the drop-down list under the **Replication instance** section, that replication instance we created in **step 2** of this recipe.
  - iii. In the **Source database endpoint** section, choose the source database from the drop-down list.
  - iv. In the **Target database endpoint** section, choose the target database from the drop-down list.
  - v. In the **Migration type** section, choose **Migrate existing data and replicate ongoing changes** from the drop-down list.
  - vi. Please refer to the following figure:



**Figure 4.22:** AWS DMS – Create database migration task

- c. Under the **Task setting** tab, we keep the default settings, except for the following:
  - i. Check the **Turn on validation** checkbox to allow the **AWS DMS** to perform source and target data validation when migrating data sets (*Figure 4.23*):

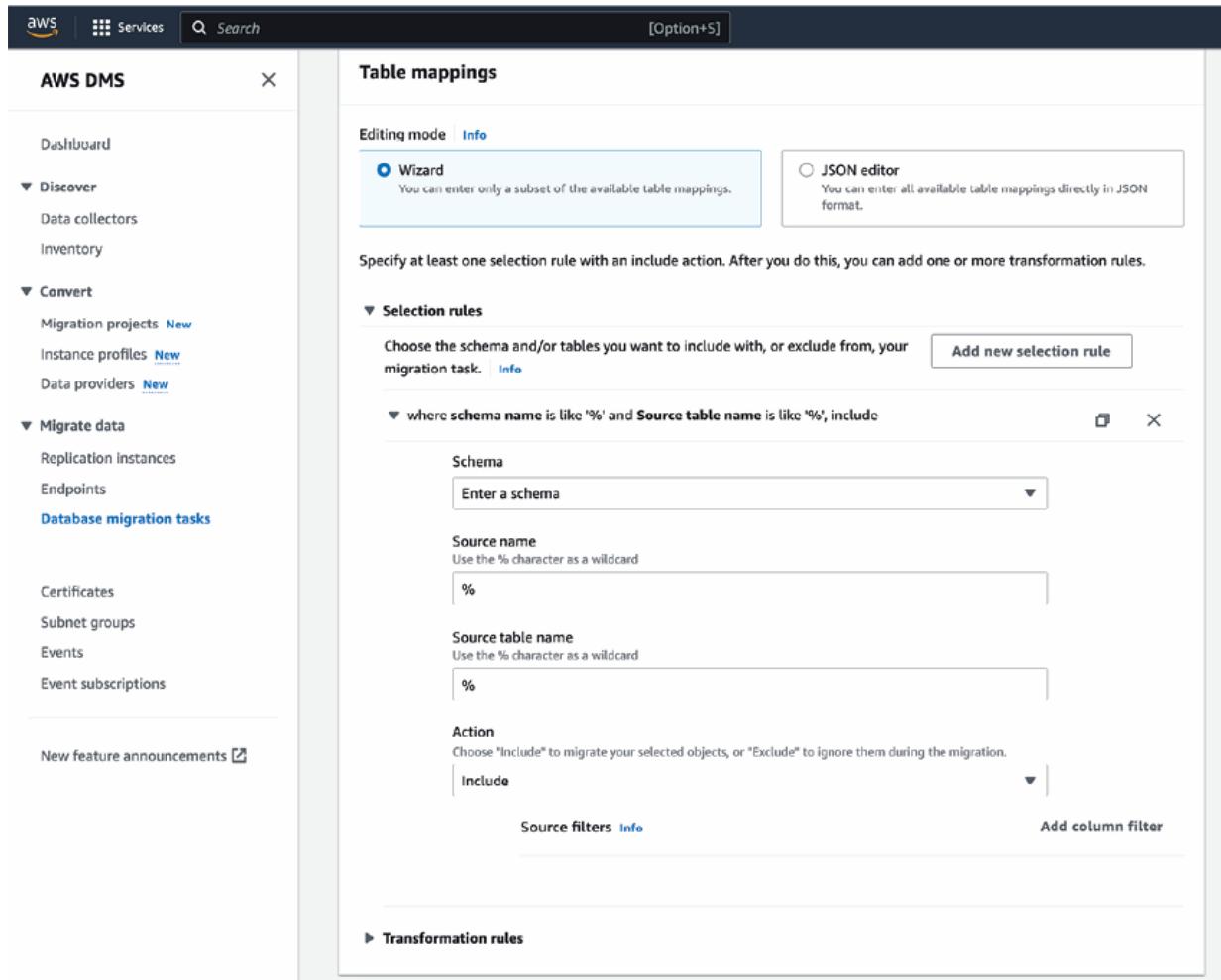


**Figure 4.23:** AWS DMS – Database migration task settings

- d. Referring to *Figure 4.24*, on the **Table mappings** tab, make the following settings:
  - i. In the **Selection rules** section, click on **Add new selection rules** and select the following settings:

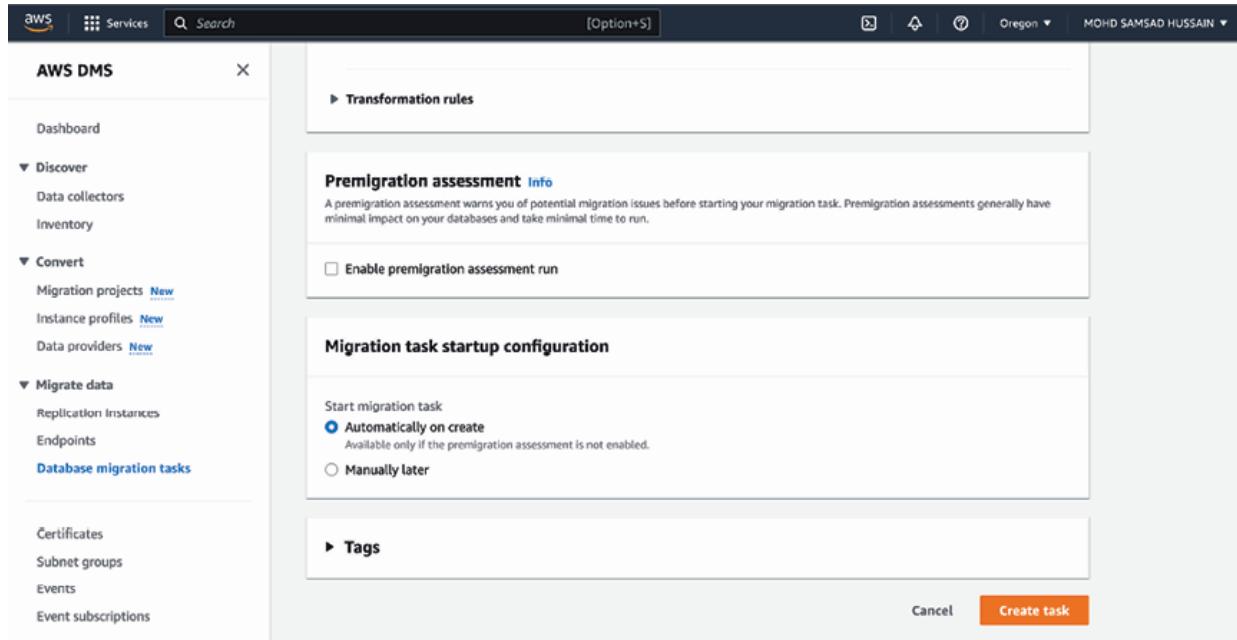
In the **Schema** section, choose **Enter a schema** from the drop-down list.

Keep the wildcard entry under the **Source name** and **Source table name** section.



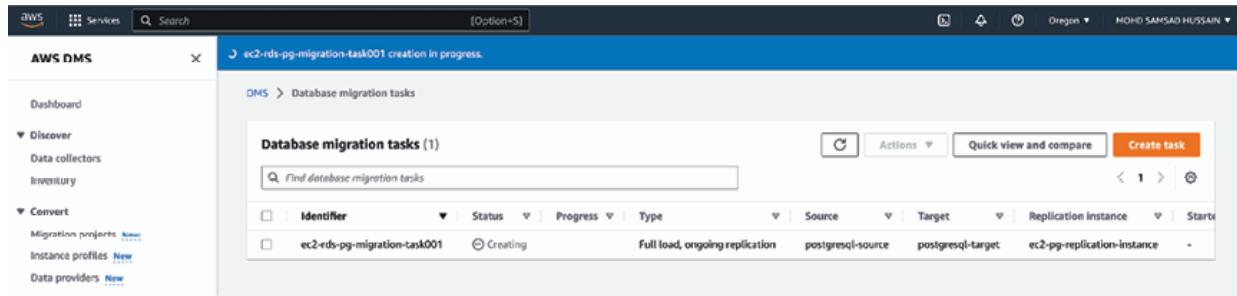
**Figure 4.24:** AWS DMS – Database migration task table mapping

- e. In the **Migration task startup configuration**, select the **Automatically on create** option and then click **Create task**, as seen in [Figure 4.25](#).



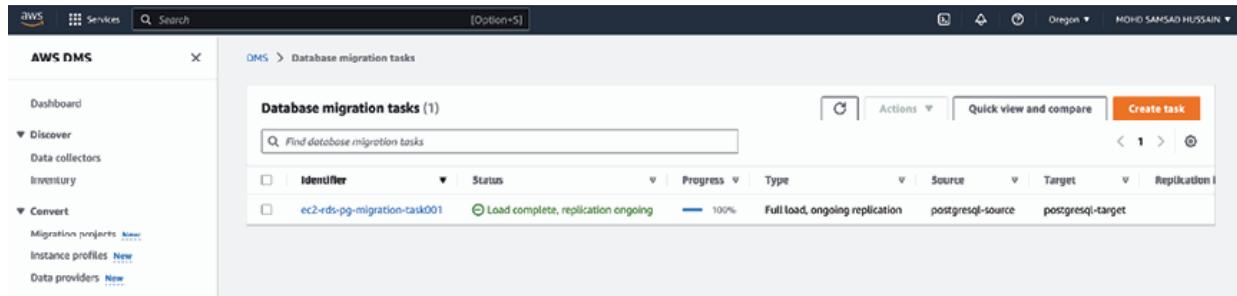
**Figure 4.25:** AWS DMS – Database migration task start-up configuration

f. As shown in *Figure 4.26*, the **Create task** option has started to create a database migration task. Please refer to the following figure:



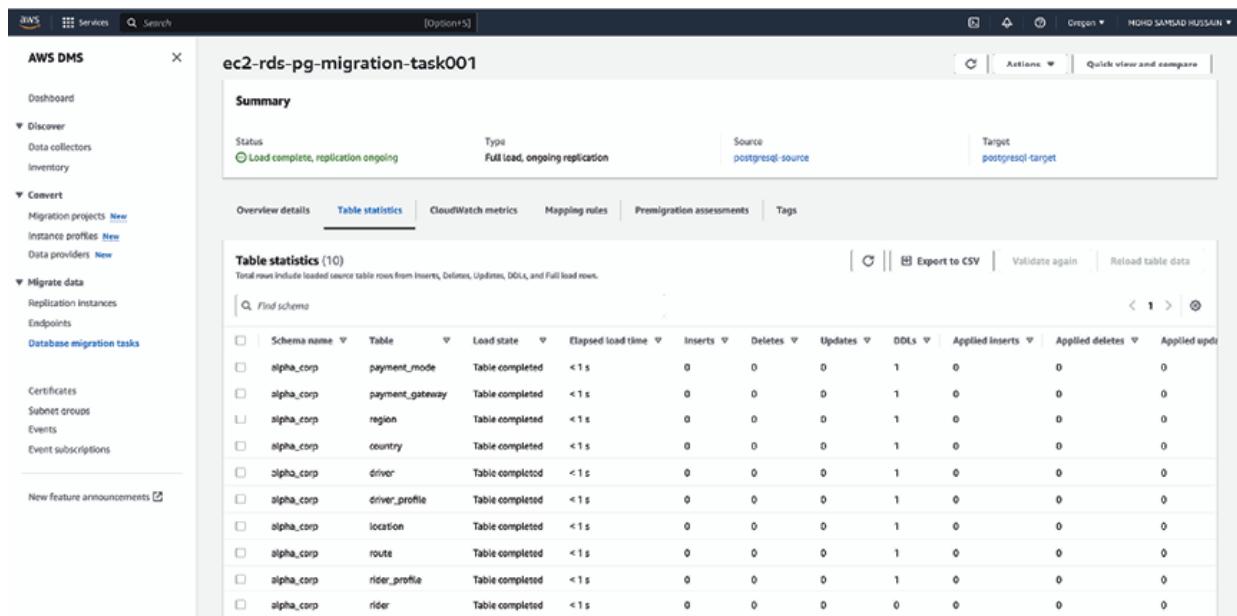
**Figure 4.26:** AWS DMS – Database migration task list

g. At this point, we can see from the **Status** column, as shown in *Figure 4.27* that our migration task configuration has finished and started replication of data from the source to the target PostgreSQL database. Please refer to the following figure:



**Figure 4.27:** AWS DMS – Database migration task list status

h. As shown in *Figure 4.27*, Under the database migration tasks tab, select the task from the list of available tasks. Click on the task name **ec2-rds-pg-migration-task001** as shown in the identifier column that will open a task page, (refer *Figure 4.28*) select the **Table statistics** option under the task page to obtain the summary detail of the migration from source to target database. Please refer to the following figure:



**Figure 4.28:** AWS DMS – Database migration statistics

7. The next step is to verify the compare the migrated database between the source and target database.

a. Since we have chosen **prod\_db** as our source database, in this step, we first verify the source system's data set. As indicated in *Figure 4.29*, below is a set of tables in the **prod\_db** source database.

```
AWS_Pair_key - root@ip-172-31-17-188:~ ssh -i con_pg.pem ec2-user@ec2-18-236-102-69.us-west-2.compute.amazonaws.com
[root@ip-172-31-17-188 ~]# psql --username=postgres --host=ec2-18-236-102-69.us-west-2.compute.amazonaws.com --dbname=prod_db --port=5432
Password for user postgres:
psql (14.6)
Type "help" for help.

prod_db=# SELECT table_catalog, table_schema, table_name FROM information_schema.tables WHERE table_schema = 'alpha_corp';
table_catalog | table_schema | table_name
-----+-----+-----+
prod_db     | alpha_corp | rider
prod_db     | alpha_corp | driver
prod_db     | alpha_corp | location
prod_db     | alpha_corp | region
prod_db     | alpha_corp | country
prod_db     | alpha_corp | route
prod_db     | alpha_corp | payment_gateway
prod_db     | alpha_corp | payment_mode
prod_db     | alpha_corp | rider_profile
prod_db     | alpha_corp | driver_profile
(10 rows)

prod_db=#
```

**Figure 4.29:** AWS DMS - Database migration verification at Source system

b. Since we have chosen **postgres** as our target database, in this step, we first verify the target system's data set. As indicated in *Figure 4.30*, below is a set of tables in the **postgres** target database that is replicated from the source system. Please refer to the following figure:

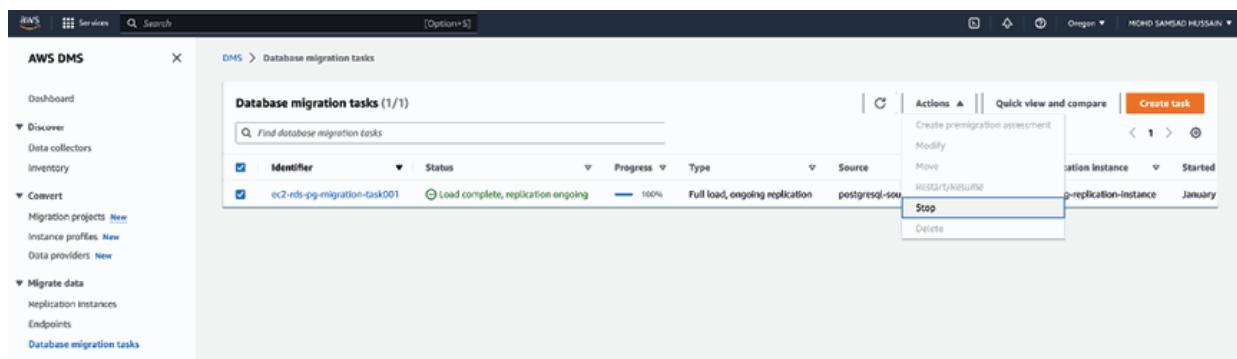
The screenshot shows the pgAdmin 4 interface with the title bar 'pgAdmin 4'. The left sidebar shows 'Servers [2]' with 'ec2\_mig1' and 'rds\_mig2' selected. Under 'rds\_mig2', the 'Databases [2]' section is expanded, showing 'postgres' selected. The main pane displays the SQL query results for the 'postgres' database:

```
postgres=> SELECT table_catalog, table_schema, table_name FROM information_schema.tables WHERE table_schema = 'alpha_corp';
table_catalog | table_schema | table_name
-----+-----+-----+
postgres     | alpha_corp | rider_profile
postgres     | alpha_corp | rider
postgres     | alpha_corp | driver_profile
postgres     | alpha_corp | driver
postgres     | alpha_corp | location
postgres     | alpha_corp | region
postgres     | alpha_corp | country
postgres     | alpha_corp | route
postgres     | alpha_corp | payment_gateway
postgres     | alpha_corp | payment_mode
(10 rows)
```

**Figure 4.30:** AWS DMS – Database migration verification at Target system

**Note: For representation purposes, we compared the list of tables between source and target at step 7 of this recipe only. In actual production, point 6.8 of Step 6 is for the validation of the migrated data.**

8. By referring to [Figure 4.31](#), at this stage, we have successfully migrated the data from the PostgreSQL database cluster hosted on EC2 to PostgreSQL hosted on the RDS instance, and all data is synchronized to your targeted database.
9. The next step is to use the target database as the main database by deploying the app on the target database, but prior to that perform the following steps to AWS DMS configuration.
  - a. Under the **Database migration tasks** tab, select the task from the checkbox that we have created in step 6 of this recipe, and select **Stop** from the drop-down list in the **Action** tab to stop the selected migration task. (Ensure that no other transactions are performed on the source server, otherwise, this transactional data cannot be synchronized with the target server). Please refer to the following figure:



**Figure 4.31:** AWS DMS – Database migration task status

10. In conclusion, the application can be pointed and use a target database as the primary database.

## Getting ready with pgloader

Database migration can be overwhelming since it involves migrating data from one database management system to another. The right migrate tool should easily deliver the scalable and efficient method of migrating the database.

**pgloader** is an open-source cross-platform database migration tool that facilitates the process to migrate a database between the database system. It is designed to migrate the entire database into a single command line allowing the implementation of continuous migration, which uses **COPY PostgreSQL** protocol to stream the data into the server.

Furthermore, pgloader operates in two modes: Importing from files and migrating databases. The details of each mode are explained in the following table:

Importing from file features	Description
File base format support	pgloader supports a large number of source file formats include. <ul style="list-style-type: none"><li>• CSV</li><li>• Fixed width flat files</li><li>• dBase</li><li>• IBM IXF</li></ul>
Archive file support	pgloader support the reading source file from an archive. <ul style="list-style-type: none"><li>• zip</li><li>• tar</li><li>• gzip</li></ul>

<b>Importing from file features</b>	<b>Description</b>
Target schema discovery	pgloader can automatically discover the target schema of your PostgreSQL database, simplifying the data loading process.
Full field projection	Ensures that the complete set of fields from the source is projected onto the target.

**Table 4.2:** Importing from file mode

In the following table migrating database mode is shown:

<b>Migrating database features</b>	<b>Description</b>
One-command migration	Migration process by executing a single command for the entire migration, ensuring efficiency and ease of use.
Schema discovery	pgloader automatically discovers the schema of the source database, reducing the manual effort required during migration.
ORM compatibility	Supports <b>object-relational mapping (ORM)</b> compatibility, ensuring a seamless migration process for applications using ORM frameworks.
Online ALTER schema	Facilitates online schema alteration during migration, allowing for real-time adjustments to the schema to accommodate data differences.
Encoding overrides	Enables encoding overrides, allowing you to customize character encoding settings during migration to ensure data integrity.

**Table 4.3:** Migrating database mode

While pgloader eases the migration procedure, it is still important to understand the migration procedure. We assume that you already have a general understanding of migration features. In the next recipe, we will get our hands dirty with pgloader setup and database migration.

## Recipe 27: Setting up pgloader

This recipe is simple and allows us to setup the pgloader for our migration infrastructure. For this recipe, you need one server for setting up the pgloader, we will use this server in our subsequent recipe as well for pgloader-based migration.

First, we have downloaded the pgloader RPM binaries from the download site. If not then you can download the latest version of pgloader from their official website at

<https://ftp.postgresql.org/pub/repos/yum/common/redhat/> after selecting your host operating system and version, follow the installation instructions.

When you have finished downloading pgloader RPM, follow these steps:

1. Begin the installation by executing the following command from the RPM downloaded location:

```
# Install pgLoader from RPM  
yum localinstall pgloader-3.6.9-1.rhel8.x86_64.rpm -y
```

Following the execution of the above command, the output displays the installation process of the **pgloader-3.6.9-1.rhel8.x86\_64.rpm** package through the usage of the **yum localinstall** command.

```
[root@postgresdev pgloader]# ll  
total 21388  
-rwxrwxrwx 1 root root 21099920 Dec 31 12:57 pgloader-3.6.9-1.rhel8.x86_64.rpm  
[root@postgresdev pgloader]# yum localinstall pgloader-3.6.9-1.rhel8.x86_64.rpm -y  
Updating Subscription Management repositories.  
Unable to read consumer identity  
  
This system is not registered to Red Hat Subscription Management. You can use subscription-manager to register.  
Last metadata expiration check: 9:18:41 ago on Sat 31 Dec 2022 01:00:55 PM +08.  
Dependencies resolved.  
----  
Package           Arch    Lecture   Version      Repository      Size  
Installing:  
  pgloader          x86_64          3.6.9-1.rhel8      @CommandLine      21 M  
Installing dependencies:  
  keyutils-libs-devel  x86_64          1.5.19-6.el8     InstallMedia-BaseOS      48 k  
  krb5-devel        x86_64          1.18.2-8.el8     InstallMedia-BaseOS      559 k
```

**Figure 4.32:** pgloader – installation on PostgreSQL server

2. Verify the installed pgloader version by executing the following command:

```
# Verify pgLoader installed version
$ pgloader --version
# Following the execution of the above query, the
# output provides the version information
pgloader version "3.6.7~devel"
    compiled with SBCL 2.2.10.1.rhel8
```

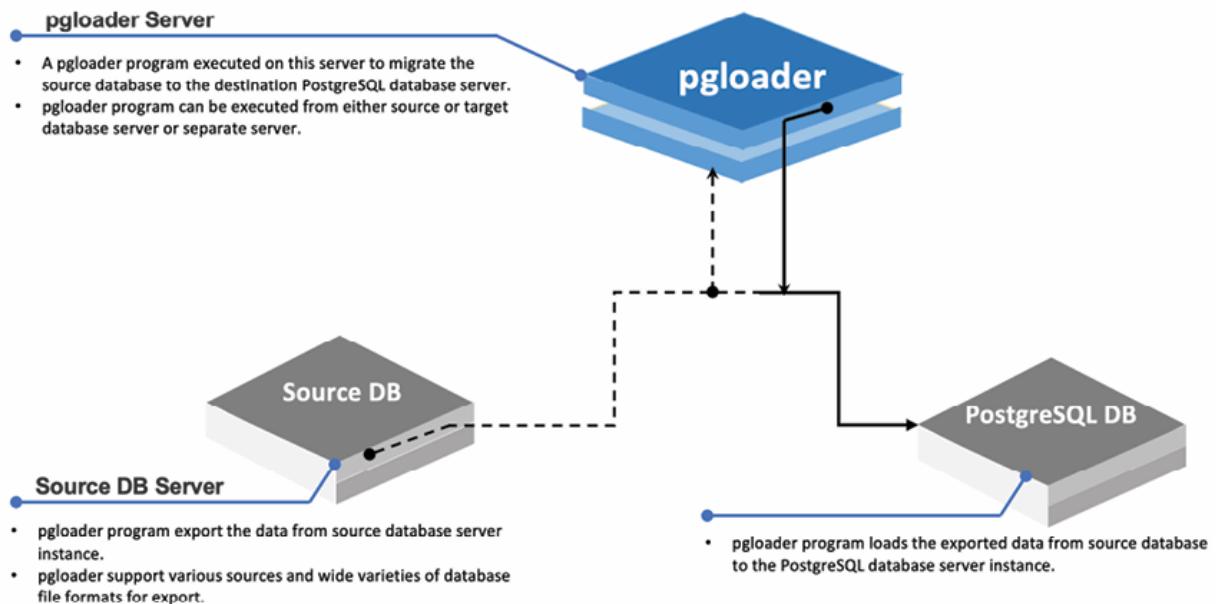
At this stage, you are all set up with the pgloader installation, and we are now ready to migrate any database instance to the PostgreSQL database server instance, which is what will be demonstrated in the next recipe, *Getting insight to migrate MariaDB and MySQL to PostgreSQL*.

## Recipe 28: Getting insight to migrate MariaDB to PostgreSQL on EC2 instance

While data warehouses are best suited to requests that require a higher level of analysis, small atomic transactions are the foundation of databases. Database is the basis for almost all the business, but situation sometimes demand moving a database to a different environment. This change in environment might be a change in infrastructure, hardware, OS, database, and so on.

This recipe is all about migrating to a PostgreSQL database where the source database distribution is **MariaDB** and **MySQL**. In this recipe, will get into a comprehensive understanding of how to migrate databases of different distributions to PostgreSQL database.

For this recipe, we use the following architecture, as shown in *Figure 4.33* to perform this migration:



**Figure 4.33:** pgloader – migration outline

Before starting the recipe, we assume the instance (MariaDB database instance) on the source database already configured and running. The following are the migration steps:

1. The first step is to install and configure PostgreSQL database instance on the destination server, refer to the *Working with Installation from Binaries* recipe from [Chapter 1, Up and running with PostgreSQL 15](#) for instructions on how to setup PostgreSQL instance.
2. Now we will create a user on the source and destination database server for Pgloader to use that user for the migration.

```
# Create a user on source MariaDB database
Server
```

```
CREATE USER 'mymigusr'@'%' IDENTIFIED BY
'Welcome#2023';
```

```
# Create a user on target PostgreSQL database
Server
```

```
CREATE USER pgmigusr WITH PASSWORD 'Welcome#2023';
```

**Note:** For this recipe, We use the `mysql_native_password` authentication method within the MariaDB database by which Pgloader is authenticated.

3. Now that we have already configured the pgloader on a separate server, Refer to the *Setting up pgloader* recipe from [Chapter 4, Migration](#), for instructions on how to setup pgloader.
4. Verify the connection between the pgloader server and the source and destination database server. The connection verification is as follows:
  - a. Verify that the source and target database server service port is open for the server from which the pgloader program gets executed.
  - b. Validate that the remote connection is open on the source and target database server for the server from which pgloader program gets executed. (Refer to the recipe *Working with PostgreSQL remote access* from [Chapter 1, Up and running with PostgreSQL 15](#), for instructions on how to allow remote access for PostgreSQL database)
5. At this stage, we are good to start the migration MariaDB database to the PostgreSQL database hosted on AWS EC2 instance. But prior to that, verify the source database dataset that needs to be migrated as shown in [Figure 4.37](#).

Following is the database migration syntax for pgloader:

**pgloader**

```
--type <csv|fixed|db3|ixf|sqlite|mysql|mssql> # Source file format
```

*type*

```
--field "field1,field2,field3,..."      # Source field definition  
--with <"Option1":"Option2"> # specific options used in  
command line execution  
--encoding <encoding> # Set the encoding of the source file  
to load data from  
-          # Standard input file
```

```
postgres://<DB_USER>:<password>@<DB_IP>:  
<DB_PORT>/<DB_NAME>?<TABLE_NAME>
```

Execute the following query to confirm the status of the database objects on the source MariaDB database server instance, ensuring they are prepared for migration.

```
# List the tables under manufacturing_src schema  
select TABLE_SCHEMA, TABLE_NAME, TABLE_ROWS from  
information_schema.tables where TABLE_SCHEMA='manufacturing_src';
```

*# Output of the above executed query*

TABLE_SCHEMA	TABLE_NAME	TABLE_ROWS
manufacturing_src	action	200
manufacturing_src	address	603
manufacturing_src	category	16
manufacturing_src	city	600
manufacturing_src	country	109
manufacturing_src	customer	599
manufacturing_src	inventory	4581
manufacturing_src	language	6
manufacturing_src	payment	16125
manufacturing_src	rental	16067
manufacturing_src	staff	2
manufacturing_src	store	2

manufacturing_src   customer_list	NULL
manufacturing_src   staff_list	NULL
manufacturing_src   sales_by_store	NULL
+-----+-----+-----+	

**Note: This step serves an illustrative purpose to comprehend the recipe. In real-world scenarios, the specific steps and outcomes may vary based on individual systems and environments.**

6. Since we already verified our source database dataset that is MariaDB in our previous step, now execute the following command from pgloader server to start the MariaDB to PostgreSQL database migration process:

```
[root@postgresdev ~]#
[root@postgresdev ~]# pgloader mysql://mymigusr:Welcome#2023@192.168.100.24:3306/manufacturing_src postgresql://pgmigusr:Welcome#2023@192.168.100.21:54
32/postgres
2023-01-09T08:15:23.014000Z LOG pgloader version "3.6.7-devel"
2023-01-09T08:15:23.016000Z LOG Data errors in '/tmp/pgloader/'
2023-01-09T08:15:23.071001Z LOG Migrating from #<MYSQL-CONNECTION mysql://mymigusr@192.168.100.24:3306/manufacturing_src {1005A4E0A3}>
2023-01-09T08:15:23.071001Z LOG Migrating into #<PGSQL-CONNECTION pgsql://pgmigusr@192.168.100.21:5432/postgres {1005D32E23}>
2023-01-09T08:15:24.456018Z ERROR PostgreSQL Database error 42501: must be owner of database postgres
QUERY: ALTER DATABASE "postgres" SET search_path TO public, manufacturing_src;
2023-01-09T08:15:24.460018Z LOG report summary reset
      table name    errors    rows    bytes   total time
-----+-----+-----+-----+-----+
      fetch meta data    0     58          0.095s
      Create Schemas    0      0          0.003s
      Create SQL Types  0      0          0.005s
      Create tables     0     24          0.059s
      Set Table IDs     0     12          0.005s
-----+-----+-----+-----+
manufacturing_src.payment    0    16044    962.6 kB  0.151s
manufacturing_src.inventory  0    4581   137.1 kB  0.043s
manufacturing_src.city       0     609    21.4 kB  0.170s
manufacturing_src.action     0     200     7.2 kB  0.367s
manufacturing_src.category   0      16     0.5 kB  0.371s
manufacturing_src.staff      0      2    71.2 kB  0.484s
manufacturing_src.rental     0    16044    1.2 MB  0.254s
manufacturing_src.address    0     603    45.7 kB  0.049s
manufacturing_src.customer   0     599    59.3 kB  0.185s
manufacturing_src.country    0     109     3.5 kB  0.167s
manufacturing_src.language   0      6     0.2 kB  0.275s
manufacturing_src.store      0      2     0.1 kB  0.264s
-----+-----+-----+-----+
COPY Threads Completion    0      4          0.628s
      Create Indexes    0     31          0.734s
Index Build Completion     0     31          0.284s
      Reset Sequences  0     12          0.037s
      Primary Keys    0     12          0.026s
Create Foreign Keys        0     15          0.051s
      Create Triggers   0     24          0.038s
      Set Search Path  1      0          0.003s
      Install Comments 0      0          0.000s
-----+-----+-----+-----+
Total import time           ✓    38806    2.4 MB  1.801s
[root@postgresdev ~]#
```

**Figure 4.34:** pgloader - database migration report on PostgreSQL server

7. Referring to the [Figure 4.35](#), we have successfully migrated to PostgreSQL database from MariaDB

database, the next step is to validate the migrated dataset on PostgreSQL database cluster. Please refer to the following figure:

```
postgres=# select TABLE_SCHEMA, TABLE_NAME from information_schema.tables where TABLE_SCHEMA='manufacturing_src';
   table schema | table name
-----+-----
manufacturing_src | city
manufacturing_src | address
manufacturing_src | country
manufacturing_src | customer
manufacturing_src | inventory
manufacturing_src | payment
manufacturing_src | rental
manufacturing_src | action
manufacturing_src | category
manufacturing_src | language
manufacturing_src | store
manufacturing_src | staff
(12 rows)
```

**Figure 4.35:** pgloader – Target PostgreSQL system data verification

Following validation of the migrated data to the target system, three tables were skipped to migrate because they contain zero rows.

## Recipe 29: Using db2topg for migration to PostgreSQL

**db2topg** is a versatile and powerful database migration tool designed to facilitate the seamless transition of data from IBM Db2 to PostgreSQL. It plays a pivotal role in simplifying what can be a complex process, offering a user-friendly interface and robust features. With Db2topg, users can efficiently generate PostgreSQL schema from their existing Db2 databases and migrate data with ease, ensuring data integrity and consistency. This tool's flexibility allows it to handle various migration scenarios, making it an invaluable asset for organizations looking to leverage the capabilities of PostgreSQL while transitioning away from Db2. Its ability to automate many migration tasks, coupled with extensive configuration options, makes Db2topg a go-to solution for database administrators and developers seeking a reliable and efficient way to move their data to PostgreSQL. This recipe will guide through the process of using Db2topg for a successful migration.

1. Begin by downloading and install **db2topg** package on PostgreSQL server instance:
  - a. Download db2topg tool from official GitHub repository <https://github.com/dalibo/db2topg> and extract the download file. Please refer to the following figure:

```
[root@pgsrvdev Software]# unzip master.zip
Archive: master.zip
012b9edde8c5c0962214869762d9acf3e6f44802
  creating: db2topg-master/
  inflating: db2topg-master/LICENSE
  inflating: db2topg-master/README.md
  inflating: db2topg-master/db2topg.pl
  inflating: db2topg-master/deltocopy.pl
  inflating: db2topg-master/parallel_unload.pl
[root@pgsrvdev Software]# |
```

**Figure 4.36:** db2topg - Download extract on PostgreSQL server

2. Configure PostgreSQL instance:
  - a. Install and configure PostgreSQL database instance on the destination PostgreSQL server, refer to the *Working with Installation from Binaries* recipe from [Chapter 1, Up and running with PostgreSQL 15](#), for instructions on how to setup PostgreSQL instance.
3. Generate Db2 schema:
  - a. Use the **db2look** command to generate the schema for IBM Db2 database by. executing following command on the source Db2 server.

```
db2look -d PRD -z customer -e -l -xd -o db2_schema.sql
```

The above script execution using the **db2look** command to generate the schema for an IBM Db2 database. Here is a breakdown of what each part of the script does:

```
db2look -d PRD -z customer -e -l -xd -o db2_schema.sql
```

```
-- No userid was specified, db2look tries to use  
Environment variable USER  
-- USER is: DB2INST1  
-- Specified SCHEMA is: CUSTOMER  
-- Creating DDL for table(s)  
-- Schema name is ignored for the Federated  
Section  
-- Output is sent to file: db2_schema.sql
```

**db2look:** This is the command to run the Db2 look utility.

**-d PRD:** Specifies the target Db2 database with the name PRD.

**-z customer:** Specifies the schema to generate DDL for, in this case, the customer schema.

**-e:** Indicates that you want to generate DDL for all objects in the specified schema.

**-l:** Generates a script that includes authorization statements.

**-xd:** Excludes the generation of XML type information.

**-o db2\_schema.sql:** Specifies the output file where the generated schema DDL will be saved, which is named **db2\_schema.sql** in this case.

#### 4. Migration processing:

- a. Navigate to the **db2topg** extracted directory and execute **db2topg.pl** to process the Db2 schema SQL and migrate it to PostgreSQL-compatible files.

```
db2topg.pl -f db2_schema.sql -o out_dir -d PRD -u db2inst1 -p  
p@ssw0rd
```

Here is a breakdown of what each part of the script does:

**-f db2\_schema.sql:** Specifies the path to the SQL file generated by **db2look**, which contains the schema definition for your Db2 database.

**-o out\_dir:** Specifies the output directory where **db2topg.pl** will write the PostgreSQL-compatible schema and data files.

**-d PRD:** Specifies the name of the source Db2 database, in this case, PRD.

**-u db2inst1:** Specifies the username for connecting to the Db2 database.

**-p p@ssw0rd:** Specifies the password for connecting to the Db2 database.

Inside the specified **out\_dir**, the successful execution of **db2topg.pl** generates the files **after.sql**, **before.sql**, **export.db2**, **TABLEDESC** and **unsure.sql** for the exported schema object. These files will contain the schema definitions and data migration commands.

## 5. Import schema into PostgreSQL instance:

- Load the data into the PostgreSQL database instance using the following command:

```
psql -e --set=ON_ERROR_STOP=1 --single-transaction -f  
before.sql postgres
```

Referring to [Figure 4.37](#), The output indicates that the SQL statements from the **before.sql** script were executed successfully, resulting in the creation of roles, a schema, and several tables within the PostgreSQL database. These schema elements are typically created to prepare the database for data migration or to establish the necessary structure for your application's data. Please refer to the following figure:

```
[postgres@pgsrdev ~]$ psql -e --set=ON_ERROR_STOP=1 --single-transaction -f /Software/db2topg-master/Migr_out/before.sql postgres
set client_encoding to UTF8;
SET
CREATE role db2inst1;
CREATE ROLE
CREATE SCHEMA customer AUTHORIZATION DB2INST1;
CREATE SCHEMA
CREATE TABLE customer.address (
    id INTEGER);
CREATE TABLE
CREATE TABLE customer.action (
    id INTEGER);
CREATE TABLE
CREATE TABLE customer.finance (
    id INTEGER);
CREATE TABLE
```

**Figure 4.37:** db2topg - import on PostgreSQL database instance

## 6. Verify schema:

- Connect to your PostgreSQL database and verify that the schema has been successfully imported. You can use SQL queries and commands to examine the schema and confirm its correctness.

## Recipe 30: Setting up foreign data wrapper for PostgreSQL migration

Although, the data is the topmost concern for any business model, it is almost inevitable that different segments of the organization will utilize different systems to produce, store and search for their essential information. Even so, it is only by combining or migrating the informational data of these diverse systems. **PostgreSQL foreign data wrapper** provides an approach for accessing data from multiple data sources in the local PostgreSQL database.

The PostgreSQL foreign data wrap method facilitates the following scenarios:

- Data analysis
- Query parallelism
- Data migration

In this recipe, we will take a closer look at how to migrate a MySQL database to the PostgreSQL database using the

foreign data wrapper. Let us start the recipe by configuring a foreign data wrapper and then migrate the database with the following steps:

1. The first step for this recipe, setting up foreign data wrapper in a PostgreSQL target server, but prior to that, make sure that our PostgreSQL cluster instance is running on the target system.
2. Since the PostgreSQL database cluster instance is active on the target system currently, the next step is to install the foreign data wrapper on the target system by following the steps below:
  - a. Download and install the MySQL foreign data wrapper RPM on the target PostgreSQL system.

*# Download the MySQL FDW RPM for  
PostgreSQL database server*

```
wget
```

```
https://download.postgresql.org/pub/repos/yum/15/redhat/rhel-8-x86\_64/mysql\_fdw\_15-2.8.0-2.rhel8.x86\_64.rpm
```

*# Install the MySQL FDW RPM to PostgreSQL  
database server*

```
rpm -ivh mysql_fdw_15-2.8.0-2.rhel8.x86_64.rpm
```

```
[root@pgdevsrv FDW_Setup]#  
[root@pgdevsrv FDW_Setup]# wget https://download.postgresql.org/pub/repos/yum/15/redhat/rhel-8-x86_64/mysql_fdw_15-2.8.0-2.rhel8.x86_64.rpm  
--2023-01-07 22:01:23-- https://download.postgresql.org/pub/repos/yum/15/redhat/rhel-8-x86_64/mysql_fdw_15-2.8.0-2.rhel8.x86_64.rpm  
Resolving download.postgresql.org (download.postgresql.org) ... 217.196.149.55, 72.32.157.246, 147.75.85.69, ...  
Connecting to download.postgresql.org (download.postgresql.org)|217.196.149.55|:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 142140 [application/x-redhat-package-manager]  
Saving to: 'mysql_fdw_15-2.8.0-2.rhel8.x86_64.rpm'  
  
mysql_fdw_15-2.8.0-2.rhel8.x86_64.rpm 100%[=====] 138.81K 262KB/s in 0.5s  
2023-01-07 22:01:25 (262 KB/s) - 'mysql_fdw_15-2.8.0-2.rhel8.x86_64.rpm' saved [142140/142140]  
  
[root@pgdevsrv FDW_Setup]#  
[root@pgdevsrv FDW_Setup]# ll  
total 140  
-rw-r--r--. 1 root root 142140 Oct 12 13:16 mysql_fdw_15-2.8.0-2.rhel8.x86_64.rpm  
[root@pgdevsrv FDW_Setup]#  
[root@pgdevsrv FDW_Setup]# rpm -ivh mysql_fdw_15-2.8.0-2.rhel8.x86_64.rpm  
warning: mysql_fdw_15-2.8.0-2.rhel8.x86_64.rpm: Header V4 DSA/SHA1 Signature, key ID 442df0f8: NOKEY  
Verifying... ##### [100%]  
Preparing... ##### [100%]  
Updating / installing...  
1:mysql_fdw-15-2.8.0-2.rhel8 ##### [100%]  
[root@pgdevsrv FDW_Setup]#  
[root@pgdevsrv FDW_Setup]#
```

**Figure 4.38:** FDW – Setup on target PostgreSQL instance

- b. Verify the installed extension on the target system by executing the following command:

```
# Verify the installed extension in PostgreSQL database
```

select \* from pg\_extension

The above query queries the PostgreSQL instance to check for the presence of the **mysql\_fdw** extension. Refer to [Figure 4.39](#), the **mysql\_fdw** extension is not present in the current list of extension:

```
[postgres@pgdevsrv ~]$ psql
psql: /usr/pgsql-15/lib/libpq.so.5: no version information available (required by psql)
psql: /usr/pgsql-15/lib/libpq.so.5: no version information available (required by psql)
psql (10.15, server 15.0)
WARNING: psql major version 10, server major version 15.
         Some psql features might not work.
Type "help" for help.

postgres=# select * from pg_extension;
   oid   | extname | extowner | extnamespace | extrelocatable | extversion | extconfig | extcondition
-----+-----+-----+-----+-----+-----+-----+-----+
 13527 | plpgsql |     10 |        11 | f             | 1.0       |           |           |
(1 row)
```

**Figure 4.39:** FDW - Install extension on Target PostgreSQL instance

- c. The next step is creating the **mysql\_fdw** extension and checking the extension that was created.

```
# Create mysql_fdw extension I PostgreSQL database
```

```
CREATE EXTENSION mysql_fdw
```

```
# Verify the above installed extension
```

```
select * from pg_extension
```

Refer to [Figure 4.40](#), The above query executes the creation of a PostgreSQL extension named **mysql\_fdw** on. The PostgreSQL server instance. This extension enables the PostgreSQL server to interact with a remote MySQL server using the **foreign data wrapper (FDW)** functionality. Please refer to the following figure:

```

postgres=# CREATE EXTENSION mysql_fdw;
CREATE EXTENSION
postgres=# select * from pg_extension;
   oid  | extname | extowner | extnamespace | extrelocatable | extversion | extconfig | extcondition
-----+-----+-----+-----+-----+-----+-----+-----+
 13527 | plpgsql |      10 |          11 | f            | 1.0        |           |
 24686 | mysql_fdw |      10 |         2200 | t            | 1.1        |           |
(2 rows)

postgres=#

```

**Figure 4.40:** FDW – Verify extension on target PostgreSQL instance

- d. As of this point, foreign data wrapper is installed on the target server. The next step is to build a foreign server with the detail of our MySQL database source server.

*# Create Foreign Data Wrapper server on PostgreSQL database server*

```

CREATE SERVER SOURCE_MYSQL_SRV FOREIGN DATA
WRAPPER mysql_fdw options (host 'mysql_srv.local', port '3306');

# Verify Foreign Data Wrapper server
select * from pg_foreign_server;

```

```

postgres=#
postgres=# CREATE SERVER SOURCE_MYSQL_SRV FOREIGN DATA WRAPPER mysql_fdw options (host 'mysql_srv.local', port '3306');
CREATE SERVER
postgres=#
postgres=# select * from pg_foreign_server;
   oid  |  srvid  | srvid  | srvid  | srvid  | srvid  | srvid
-----+-----+-----+-----+-----+-----+-----+
 24691 | source_mysql_srv |      10 | 24689 |          |          | {host=mysql_srv.local.port=3306}
(1 row)

postgres=#

```

**Figure 4.41:** FDW – Create server on target PostgreSQL instance

- e. This step involves mapping users between the source and target database server on the target server.

*# Configure User mapping for Foreign Data Wrapper server on the PostgreSQL database server*

```
CREATE USER MAPPING FOR pgfdwusr server
```

```
source_mysql_srv OPTIONS (username 'myfdwusr', password  
'Welcome@2023');
```

*# Verify user mapping for Foreign Data  
Wrapper server*

```
select * from pg_user_mappings;
```

Refer to [Figure 4.42](#), the above query initiates the creation of a user mapping connecting the source and target servers. This user mapping serves as the link facilitating communication between the PostgreSQL server and the external MySQL server. This connection enables the foreign data wrapper to function seamlessly across both systems. Please refer to the following figure:

```
postgres=# CREATE USER MAPPING FOR pgfdwusr server source_mysql_srv OPTIONS (username 'myfdwusr', password 'Welcome@2023');  
CREATE USER MAPPING  
postgres=#  
postgres=#  
postgres=#  
postgres=# SELECT * FROM pg_user_mappings;  
 umid | srvid | srvname | umuser | username | umoptions  
-----+-----+-----+-----+-----+  
 24693 | 24691 | source_mysql_srv | 24692 | pgfdwusr | {username=myfdwusr,password=Welcome@2023}  
(1 row)  
postgres=#
```

**Figure 4.42:** FDW – User mapping on target PostgreSQL instance

- f. Having established a user mapping between the source and target in the foreign server configuration, the subsequent step is to grant the user access to the foreign server on the target PostgreSQL server. This is accomplished by executing the provided statement on the target PostgreSQL instance.

*# Authorize user for Foreign server access*

```
GRANT USAGE ON FOREIGN SERVER source_mysql_src TO  
pgfdwusr;
```

- g. In this step we can migrate the MySQL database, but before that we create a schema on our target PostgreSQL database as shown in

*Figure 4.43:*

```
postgres=> \c
You are now connected to database "postgres" as user "pgfdwusr".
postgres=>
postgres=> CREATE SCHEMA workorder_SRC;
CREATE SCHEMA
postgres=>
postgres=>
postgres=> SELECT schema_name FROM information_schema.schemata;
   schema_name
-----
information_schema
pg_catalog
public
workorder_src
org
(5 rows)

postgres=>
```

*Figure 4.43: FDW - Create schema*

- h. The next step is to import the target schema **workorder** from MySQL to the PostgreSQL database schema **workorder\_src**.

```
# Import the target schema from source
schema
```

```
IMPORT FOREIGN SCHEMA workorder FROM SERVER
source_mysql_srv INTO workorder_src;
```

Refer to *Figure 4.44*, The above query start importing a target PostgreSQL database schema from a source MySQL database schema using PostgreSQL's foreign data wrapper functionality. Please refer to the following figure:

```
postgres=>
postgres=> IMPORT FOREIGN SCHEMA workorder FROM SERVER source_mysql_srv INTO workorder_src;
IMPORT FOREIGN SCHEMA
postgres=>
```

*Figure 4.44: FDW - Import foreign schema*

- i. Since the import successfully done on the **workorder\_src** schema of the PostgreSQL database instance, at this stage, any DML operation on the source schema object **workorder** will be reflected in the target schema objects **workorder\_src**. Refer to the *Figure 4.45* and *Figure 4.46* for the source and target schema the objects list:

```
mysql> select TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE from information_schema.tables where TABLE_SCHEMA='WORKORDER';
+-----+-----+-----+-----+
| TABLE_CATALOG | TABLE_SCHEMA | TABLE_NAME | TABLE_TYPE |
+-----+-----+-----+-----+
| def | workorder | action | BASE TABLE |
| def | workorder | address | BASE TABLE |
| def | workorder | category | BASE TABLE |
| def | workorder | city | BASE TABLE |
| def | workorder | country | BASE TABLE |
| def | workorder | customer | BASE TABLE |
| def | workorder | customer_list | VIEW |
| def | workorder | inventory | BASE TABLE |
| def | workorder | language | BASE TABLE |
| def | workorder | payment | BASE TABLE |
| def | workorder | rental | BASE TABLE |
| def | workorder | sales_by_store | VIEW |
| def | workorder | staff | BASE TABLE |
| def | workorder | staff_list | VIEW |
| def | workorder | store | BASE TABLE |
+-----+-----+-----+-----+
15 rows in set (0.00 sec)

mysql> ■
```

**Figure 4.45: FDW – Verify source system**

```
postgres=>
postgres=>
postgres=> SELECT table_catalog, table_schema, table_name, table_type FROM information_schema.tables where table_schema='workorder_src';
+-----+-----+-----+-----+
| table_catalog | table_schema | table_name | table_type |
+-----+-----+-----+-----+
| postgres | workorder_src | action | FOREIGN |
| postgres | workorder_src | address | FOREIGN |
| postgres | workorder_src | category | FOREIGN |
| postgres | workorder_src | city | FOREIGN |
| postgres | workorder_src | country | FOREIGN |
| postgres | workorder_src | customer | FOREIGN |
| postgres | workorder_src | customer_list | FOREIGN |
| postgres | workorder_src | inventory | FOREIGN |
| postgres | workorder_src | language | FOREIGN |
| postgres | workorder_src | payment | FOREIGN |
| postgres | workorder_src | rental | FOREIGN |
| postgres | workorder_src | sales_by_store | FOREIGN |
| postgres | workorder_src | staff | FOREIGN |
| postgres | workorder_src | staff_list | FOREIGN |
| postgres | workorder_src | store | FOREIGN |
+-----+-----+-----+-----+
(15 rows)

postgres=> ■
```

**Figure 4.46: FDW – Verify target system**

- j. Referring *Figure 4.47*, let us do a DML operation on a source system's action table and verify the

data from a PostgreSQL target system using the following command:

```
# Insert sample data in the source MySQL database system
```

```
INSERT INTO workorder.action(action_id, first_name, last_name) VALUES(201, 'Bob', 'Builder');
```

```
mysql>
mysql> INSERT INTO workorder.action(action_id, first_name, last_name) VALUES(201, 'Bob', 'Builder');
Query OK, 1 row affected (0.00 sec)

mysql> select * from workorder.action where action_id=201;
+-----+-----+-----+
| action_id | first_name | last_name | last_update |
+-----+-----+-----+
|     201 | Bob        | Builder   | 2023-01-08 18:07:42 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

**Figure 4.47:** FDW – DML operation on Source

**Figure 4.48** shows the status of the data prior to insertion in the source system **action** table. Please refer to the following figure:

```
mysql>
mysql> select max(action_id) from workorder.action;
+-----+
| max(action_id) |
+-----+
|      200      |
+-----+
1 row in set (0.00 sec)

mysql>
```

**Figure 4.48:** FDW – Source data verification

k. It is time to review the transaction made in step 9 of this recipe from the target PostgreSQL database, as shown in **Figure 4.49**:

```

postgres=>
postgres=> select * from workorder_src.action where action_id=201;
   action_id | first_name | last_name | last_update
-----+-----+-----+-----+
  201      | Bob        | Builder    | 2023-01-08 18:07:42
(1 rows)

postgres=>
```

**Figure 4.49:** FDW - Data verification on the Target PostgreSQL system

**Note:** Any new object, for example a table created under a source scheme, will not be reflected in the target scheme unless the scheme is re-imported.

1. Since the DML change on the source schema reflected on already imported objects in the target PostgreSQL database system, now let us create a table **action\_mod** in the source MySQL database system and reimport the new objects.

# Create table action\_mod on source MySql Database system

```

CREATE TABLE `action_mod` ( `action_id` smallint unsigned NOT NULL
AUTO_INCREMENT, `first_name` varchar(45) NOT NULL, `last_name` varchar(45) NOT NULL,
`last_update` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP, PRIMARY
KEY (`action_id`), KEY `idx_action_mod_last_name`(`last_name`) )
ENGINE=InnoDB AUTO_INCREMENT=202 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci;
```

The above executed query creates a table named **action\_mod** in a MySQL database. To verify the executed query, we can use the following command in the MySQL command-line interface:

# Verify the table action\_mod on the source MySQL

## Database system

```
select TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE  
from information_schema.tables where  
TABLE_SCHEMA='WORKORDER';
```

```
mysql>  
mysql> select TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE from information_schema.tables where TABLE_SCHEMA='WORKORDER'  
+-----+-----+-----+-----+  
| TABLE_CATALOG | TABLE_SCHEMA | TABLE_NAME | TABLE_TYPE |  
+-----+-----+-----+-----+  
| def | workorder | action | BASE TABLE |  
| def | workorder | action_mod | BASE TABLE |  
| def | workorder | address | BASE TABLE |  
| def | workorder | category | BASE TABLE |  
| def | workorder | city | BASE TABLE |  
| def | workorder | country | BASE TABLE |  
| def | workorder | customer | BASE TABLE |  
| def | workorder | customer_list | VIEW |  
| def | workorder | inventory | BASE TABLE |  
| def | workorder | language | BASE TABLE |  
| def | workorder | payment | BASE TABLE |  
| def | workorder | rental | BASE TABLE |  
| def | workorder | sales_by_store | VIEW |  
| def | workorder | staff | BASE TABLE |  
| def | workorder | staff_list | VIEW |  
| def | workorder | store | BASE TABLE |  
+-----+-----+-----+-----+  
16 rows in set (0.00 sec)  
mysql> ■
```

**Figure 4.50:** FDW – Verify new objects at source MySQL database system

- m. At this stage, the new objects are already created on the source system but not reflected on the target system, the target system must be reimported with the parameter associating new objects as shown in [Figure 4.50](#):

# Import the target schema from source schema

```
IMPORT FOREIGN SCHEMA workorder LIMIT TO (action_mod) FROM  
SERVER source_mysql_srv INTO workorder_src;
```

The execution of above query specifically imports the **workorder** schema, with a limitation to the **action\_mod** table, from a remote server named **source\_mysql\_srv** into the local schema named **workorder\_src**.

Verify the imported schema by executing the following SQL command after successfully importing a schema using a **foreign data wrapper (FDW)**.

# Verify the table action\_mod on the Target system post

## *reimport operation*

```
select TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE  
from information_schema.tables where  
TABLE_SCHEMA='workorder_src';
```

Refer to *Figure 4.51*, Execution of the above query retrieve information from the **information\_schema.tables** view and filtering based on the **TABLE\_SCHEMA**, we can efficiently examine the tables imported via the FDW and gain a clearer understanding of their attributes within the target PostgreSQL environment. Please refer to the following figure:

The screenshot shows a pgAdmin interface with a query editor and a results grid. The query is:

```
2 SELECT table_catalog, table_schema, table_name, table_type FROM information_schema.tables WHERE TABLE_SCHEMA='workorder_src';  
3
```

The results grid displays 16 rows of data, each representing a table in the 'workorder\_src' schema. The columns are: table\_catalog, table\_schema, table\_name, and table\_type. The data is as follows:

	table_catalog name	table_schema name	table_name name	table_type character varying
1	postgres	workorder_src	action	BASE TABLE
2	postgres	workorder_src	address	BASE TABLE
3	postgres	workorder_src	category	BASE TABLE
4	postgres	workorder_src	city	BASE TABLE
5	postgres	workorder_src	country	BASE TABLE
6	postgres	workorder_src	customer	BASE TABLE
7	postgres	workorder_src	customer_list	BASE TABLE
8	postgres	workorder_src	inventory	BASE TABLE
9	postgres	workorder_src	language	BASE TABLE
10	postgres	workorder_src	payment	BASE TABLE
11	postgres	workorder_src	rental	BASE TABLE
12	postgres	workorder_src	sales_by_store	BASE TABLE
13	postgres	workorder_src	staff	BASE TABLE
14	postgres	workorder_src	staff_list	BASE TABLE
15	postgres	workorder_src	store	BASE TABLE
16	postgres	workorder_src	action_mod	BASE TABLE

**Figure 4.51:** FDW – Verify new objects at target PostgreSQL instance

At this point, we succeeded in migrating the data from MySQL to the PostgreSQL database instance. However, in addition to **IMPORT FOREIGN SCHEMA**, we can also use **CREATE FOREIGN TABLE** to further granularize the source and target import based on the specific tables.

## Conclusion

In this chapter, we have learned about Postgres database migration techniques and the diverse set of migration methodologies for both on-premises and cloud database systems. This chapter also provides a comprehensive understanding of the tools used to migrate a database from a multi-base distribution to the PostgreSQL database instance.

After going through this chapter in detail, the reader will be familiar with the migration technique and understand the possible advantage applicable to what type of migration technique.

In the next chapter, we will begin to learn about the database logging mechanism and its architecture that helps to build database availability.

## **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

# Transaction Log

## Introduction

PostgreSQL is a powerful and popular open-source relational database management system that is widely used in enterprise applications. One key feature that makes PostgreSQL a robust and reliable database system is its **transaction log**. The transaction log records all changes made to the database, allowing it to be recovered to a known, consistent state in the event of a hardware or software failure. Understanding the transaction log is essential for database administrators and developers who want to ensure data consistency and durability and optimize the performance of their PostgreSQL databases.

## Structure

In this chapter, we will cover the following topics:

- Introducing transaction log
- Understanding WAL
- Exploring the internal layout of the WAL segment
- Exploring the internal layout of XLOG record
- Configuring the Write-Ahead Log
- Managing the WAL writer process
- Optimizing checkpoint processing
- Automating vacuum process

- Implementing continuous archiving and archive logs
- Enhancing performance and benefits of WAL

## Objectives

The objective of a chapter on PostgreSQL transaction log for a book on database management would be to provide an in-depth understanding of the transaction log in PostgreSQL, its importance in ensuring data consistency and durability, and how it can be managed and utilized effectively. The chapter will cover topics such as the overview of the transaction log in PostgreSQL, the importance of transaction log for data consistency and durability, understanding the **Write-Ahead Logging (WAL)** protocol, management and configuration of transaction log settings, analyzing and interpreting transaction log data, utilizing transaction log for point-in-time recovery and replication, best practices for using and maintaining the transaction log, and real-world examples and case studies illustrating the use of transaction log in PostgreSQL.

## Introducing transaction log

A database transaction log is a file (or set of files) that records all changes made to a database. It is used to help ensure data integrity, as any changes made to the database can be traced back to their original source. The log includes records of the changes made to the data, such as the type of operation (for example, delete, insert, or update), and the time the operation was performed.

For example, when an author is drafting a book, the database would record all changes made to the book's data, such as new chapters being added, chapters being removed, or paragraphs being edited. The transaction log would record these changes so that, in the event of a system failure, the data can be recovered, and the book's progress can be restored.

PostgreSQL stores transaction log information in the **pg\_wal** (in PostgreSQL 9.6 and earlier: **pg\_xlog**) directory. The **pg\_wal** directory contains a sequence of log files, each of which is called a WAL segment. The transaction log records the changes to the database in chronological order, including information about each transaction such as the type of transaction, the date and time of the transaction, the user who initiated the transaction, and the data

that was changed.

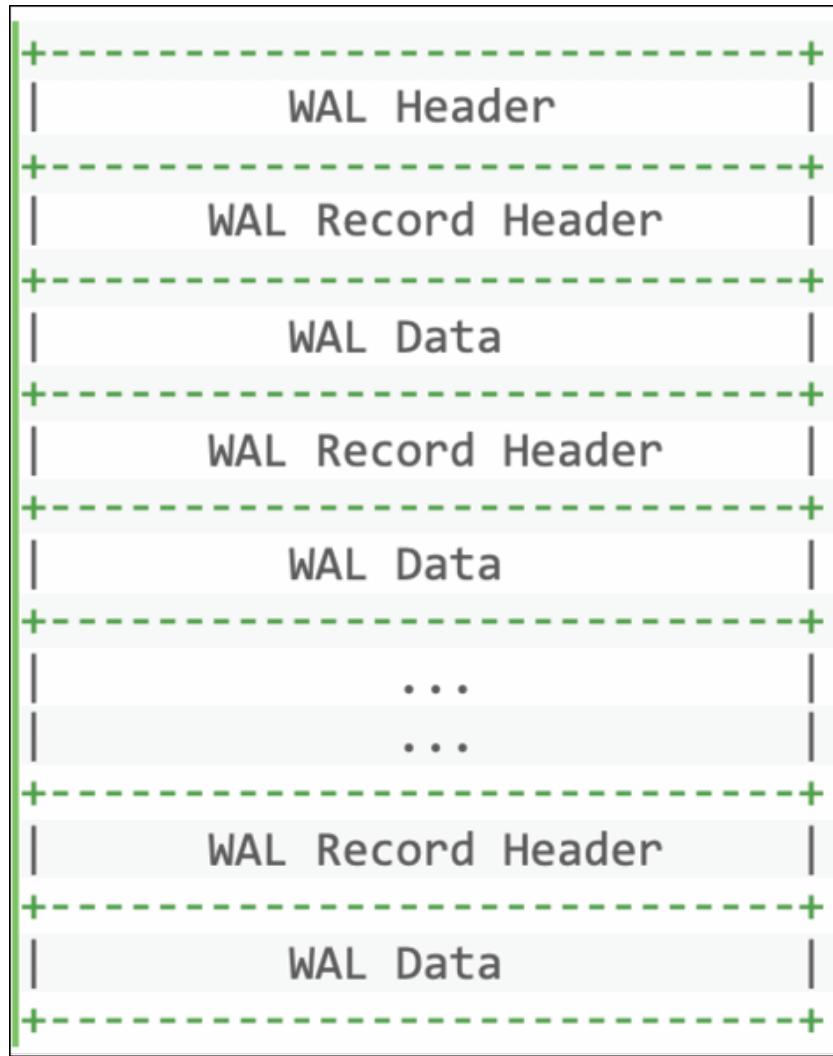
The log is used to roll back any changes that were made to the database in the event of an error. Additionally, it also helps in disaster recovery, allowing administrators to restore the database to its previous state.

## **Understanding WAL**

PostgreSQL Write-Ahead Logging is a feature that provides data durability, consistency, and recoverability in the event of a system crash. It works by keeping a record of all changes made to the database in a special log file. This log file is referred to as the WAL. Whenever a transaction is committed, the changes are written to the WAL before they are written to the database. If the database crashes or is shut down before the transaction is committed, the log records can be used to recover the changes when the database is restarted. WAL also provides a way to replicate data between two or more databases, as the log records can be replayed on the replica databases.

## **Exploring the internal layout of WAL segment**

In PostgreSQL 15, the WAL system uses a segmented architecture for improved performance and reliability. Each WAL segment is a fixed-size file on disk (default is 16 MB), and the WAL files are written sequentially in a circular buffer. The internal layout of a WAL segment is as shown in *Figure 5.1*:



**Figure 5.1:** WAL Segment layout

The internal layout of a WAL segment in PostgreSQL 15 is as follows:

**WAL segment header:** Each WAL segment begins with a 24-byte header that contains information about the segment, such as the timeline ID, the starting position in the WAL stream, and the size of the segment.

**WAL page header:** Each WAL segment is divided into multiple 8 KB pages. Each page begins with a 24-byte header that contains information about the page, such as the page number and the **log sequence number (LSN)** of the last record written on the page.

**WAL records:** Each WAL page can contain multiple WAL records, which are the individual log entries that contain the changes made

to the database. The size of a WAL record varies depending on the nature of the change being logged.

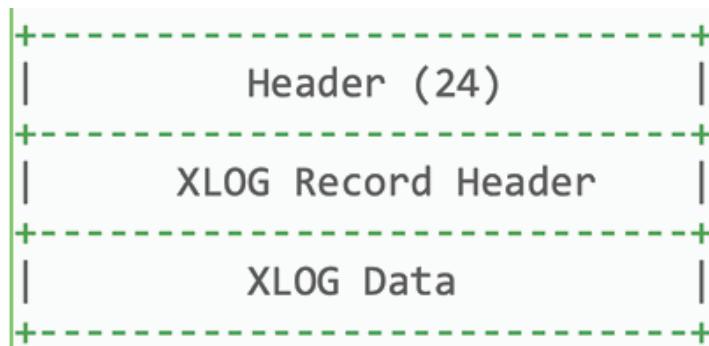
**Checksum:** For improved reliability, each WAL page and WAL record includes a 4-byte **CRC32** checksum that is calculated over the contents of the page or record.

**Control information:** At the end of each WAL segment, there is a block of control information that contains a copy of the most recent checkpoint record and other information needed for recovery.

The internal layout of a WAL segment in PostgreSQL 15 is designed to provide a reliable and efficient way to store and manage transaction log information. By dividing the WAL stream into fixed-size segments and using checksums and control information to ensure consistency, PostgreSQL 15 provides a robust foundation for data recovery and high availability.

## Exploring the internal layout of XLOG record

In PostgreSQL, WAL system uses a file format called **XLOG** (short for transaction log) to write changes made to the database to disk. The XLOG format is designed to be efficient and durable, allowing for fast crash recovery and point-in-time recovery of the database. The internal layout of a XLOG record is depicted in [Figure 5.2](#):



**Figure 5.2: XLOG Layout**

The XLOG (transaction log) files are organized into segments, with each segment containing a fixed number of blocks. The internal layout of each block in an XLOG segment is as follows:

The first 24 bytes of the block are reserved for a header, which contains information such as the location of the block within the **XLOG** file, the length of the block, and a checksum to ensure data

integrity.

Following the header is a variable-length **XLOG record** that contains information about a single transaction or change to the database. Each XLOG record includes a header that contains information such as the transaction ID, the type of record, and the length of the record.

The remainder of the block is used to store the actual data associated with the XLOG record. This data can include information about changes made to the database, as well as metadata such as transaction IDs and timestamps.

The XLOG segments are typically stored in the **pg\_wal** subdirectory of the PostgreSQL data directory and are named according to a sequence of numbers that represent the start and end points of the segment.

The XLOG format in PostgreSQL 15 also includes a feature called **logical decoding**, which allows developers to extract a stream of logical changes from the XLOG for use in other applications. This can be useful for tasks such as data replication and integration with external systems. Logical decoding works by parsing the XLOG records and extracting the relevant data, such as the before and after values of a changed row.

In summary, the XLOG format in PostgreSQL 15 provides a durable and efficient way to store changes to the database, allowing for fast crash recovery and point-in-time recovery. The XLOG format includes a header, a variable-length XLOG record, and the actual data associated with the record.

## Configuring the WAL file

In PostgreSQL 15, the WAL system is configured through the **postgresql.conf** configuration file, which is typically located in the PostgreSQL data directory.

The following are some of the important WAL-related configuration parameters that can be set in the **postgresql.conf** file:

**wal\_level:** This parameter controls the amount of information that is written to the WAL. The valid values are minimal, replica, and logical, with logical providing the most detailed information.

**max\_wal\_size:** This parameter sets the maximum size of a WAL

segment on disk. Once the segment reaches this size, a new segment is created.

**min\_wal\_size:** This parameter sets the minimum size of a WAL segment on disk. If the size of the WAL falls below this value, a new segment is created.

**wal\_keep\_segments:** This parameter sets the number of completed WAL segments that are kept on disk for archival purposes. These segments can be used for point-in-time recovery.

**checkpoint\_completion\_target:** This parameter controls how much of the checkpoint processing time is spent writing dirty buffers to disk. A higher value can reduce the time it takes to recover from a crash but can also impact the performance of the database.

**wal\_writer\_delay:** This parameter sets the amount of time that the WAL Writer process waits between writing WAL buffers to disk. Increasing this value can reduce disk **input/output (I/O)** but can also increase the time it takes to recover from a crash.

**wal\_buffers:** This parameter sets the number of WAL buffers in memory that are used to temporarily store changes before they are written to disk. Increasing this value can reduce disk I/O but can also increase the memory usage of the database.

The **postgresql.conf** file provides a flexible and configurable way to manage the WAL system in PostgreSQL 15. By tuning these and other parameters to suit the needs of the application, developers can ensure optimal performance and reliability of their database systems.

## PostgreSQL WAL benefits

Discover the advantages of PostgreSQL WAL is a powerful feature that offers numerous benefits to the database management landscape. The following are the benefits of WAL:

- WAL is a feature of PostgreSQL that allows for efficient logging of all changes made to the database. This means that a complete history of all changes made to the database is kept, which is especially useful for debugging and data recovery.
- WAL improves the performance of PostgreSQL by reducing the amount of time needed for recovery after an unexpected shutdown. This is because it allows the database to quickly

rollback any changes made before the shutdown, instead of having to manually re-enter the data.

- WAL also increases the durability of the data in the database by allowing for atomic transactions. This means that either all the changes made by a transaction are applied, or none of them are. This prevents data corruption in the event of an unexpected shutdown.
- WAL also allows for point-in-time recovery. This means that you can easily restore a database to a point in time before a certain event occurred. This is especially useful for data recovery in the event of a disaster.
- WAL also helps improve the performance of write operations, as writing can be done in batches, rather than one at a time. This reduces disk overheads as fewer disk writes are needed.
- WAL also improves the performance of PostgreSQL when running multiple databases. This is because it can allow for concurrent access to the database, meaning multiple users can make changes to the database at the same time. This can improve the speed of data processing, as multiple transactions can be processed in parallel.
- Finally, WAL can help to reduce the amount of time needed to perform backups, as the WAL log can be used to quickly determine which data needs to be backed up. This makes it easier and faster to perform a full backup of the database.

## Recipe 31: How to enable/disable archive mode

Now we have deeply understood the WAL and its benefits in a theoretical aspect. In this recipe, we start working with the WAL by enabling and disabling the archive mode. So let us go straight to the archive mode configuration with the following steps:

1. Archiving mode can be enabled by setting the parameter **archive\_mode** to **on** in the **postgresql.conf** configuration file. This can also be done by running the following command:

```
# Enable Archiving
```

```
alter system set archive_mode = on;
```

This parameter can be set to on, off, or always. When set to **on**, the database server will begin archiving WAL files into a

designated archive directory.

2. The next step is to configure the **wal\_level** configuration. The default value for PostgreSQL 15's **wal\_level** configuration setting is **replica**. This setting determines how much information is written to the WAL, which is used for replication and point-in-time recovery.

```
ALTER SYSTEM SET wal_level = '<level>';
```

where **<level>** is one of the following values:

**minimal** (minimal data written to the WAL)

**replica** (default - sufficient information for replication and PITR)

**logical** (sufficient information for logical decoding)

By default, **wal\_level** configuration is set to **replica**. For this recipe, we use the default configuration for archiving as shown in [Figure 5.3](#):

```
postgres=# select name,setting,unit from pg_settings where name in ('wal_level');
   name    | setting | unit
-----+-----+-----
wal_level | replica | 
(1 row)
```

*Figure 5.3: wal\_level configuration*

3. The last step for the archive configurations is to configure the **archive\_command** settings. The parameter **archive\_command** must also be set to a valid command for archiving the WAL file segment.

The command provided in **archive\_command** should be capable of taking two arguments: **%p** is replaced by the pathname of the file to archive, and **%f** is replaced by the file name only. This command should be used in conjunction with **wal\_level** and **archive\_mode** parameters to enable archiving of WAL files as shown in the following configuration:

```
# set archive_command configuration to a valid
directory
```

```
alter system set archive_command = 'test ! -f /wal_archive/%f && cp %p
/wal_archive/%f';
```

```
# set wal_keep_size configuration to a 160 MB
```

```

alter system set wal_keep_size='160 MB';
# Verify WAL Archiving configuration
select name, setting,unit from pg_settings where name in ('archive_mode',
'archive_command', 'wal_keep_size', 'wal_level');

```

In the above archive command, the script is divided into two parts separated by **&&**. The former part of the script **test ! -f /archivedir/%f** is a Unix command used to check if a specific file indicated by **%f** exists in the directory **/archivedir** before proceeding to the later part of the script. If the file does not exist, the latter part of the script copies the file specified with **%p** to the **directory /archivedir/%f**. Refer to *Figure 5.4* to verify the WAL Archiving configuration:

```

postgres=# select name,setting,unit from pg_settings where name in ('archive_mode','archive_command', 'wal_keep_size', 'wal_level');
      name      |           setting           | unit
-----+-----+-----+
archive_command | test ! -f /wal_archive/%f && cp %p /wal_archive/%f |
archive mode    | on
wal keep size   | 160
wal_level       | replica
(4 rows)

```

**Figure 5.4:** WAL Archiving configuration

4. Lastly, restart the PostgreSQL server to apply the changes. After enabling the configuration, execute the following command to verify the archived WAL files, which will be stored in a designated directory and can be utilized for point-in-time recovery, if necessary.

```
# Restart PostgreSQL database cluster service
```

```
systemctl restart postgresql-15.service
```

```
# Verify the Archived WAL
```

```

select archived_count, last_archived_wal, last_archived_time from
pg_stat_archiver;

```

**Note:** By setting the **archive\_command** to '**/bin/true**', enables a more flexible configuration, facilitating changes to archiving settings without requiring a restart of the PostgreSQL service. A configuration reload is deemed sufficient for such modifications.

- set Boolean value to the **archive\_command**
- **alter system set archive\_command TO '/bin/true';**
- **Reload the configuration**
- **select pg\_reload\_conf();**

Verify the generate archive file from the statistics view by executing the above command or directly form the

**/wal\_archive** path as shown in *Figure 5.5*:

```
postgres=# select archived_count, last_archived_wal, last_archived_time from pg_stat_archiver ;
 archived_count | last_archived_wal      | last_archived_time
-----+-----+-----+
        42 | 00000001000000100000089 | 2023-02-12 13:30:07.108527+08
(1 row)

[root@postgresdev wal_archive]# ls -lrt /wal_archive/
total 704512
-rw-----. 1 postgres postgres 16777216 Feb 12 13:19 00000001000000100000008F
-rw-----. 1 postgres postgres 16777216 Feb 12 13:29 000000010000001000000090
-rw-----. 1 postgres postgres 16777216 Feb 12 13:29 000000010000001000000091
-rw-----. 1 postgres postgres 16777216 Feb 12 13:29 000000010000001000000092
-rw-----. 1 postgres postgres 16777216 Feb 12 13:29 000000010000001000000093
-rw-----. 1 postgres postgres 16777216 Feb 12 13:29 000000010000001000000094
-rw-----. 1 postgres postgres 16777216 Feb 12 13:29 000000010000001000000095
```

**Figure 5.5:** WAL archive succeed

- At this stage archiving is configured and generated as shown in the previous step. The final step of this recipe is to disable the archive in PostgreSQL instance. Archiving mode can be disabled by setting the parameter **archive\_mode** to **off** in the **postgresql.conf** configuration file. This can also be done by running the following command:

*# Enable Archiving*

```
ALTER SYSTEM SET archive_mode = off;
```

- Finally, restart the PostgreSQL server for the changes to take effect. Once this command is executed, the **archive\_mode** setting will be set to **off**, which means that PostgreSQL will no longer attempt to create archived WAL files.

*# Restart PostgreSQL database cluster service*

```
systemctl restart postgresql-15.service
```

*# Verify the archive mode status*

```
SELECT name,setting,unit FROM pg_settings WHERE name IN ('archive_mode');
```

*Figure 5.6* is the execution output of the above script to check the **archive\_mode** status:

```
postgres=# SELECT name,setting,unit FROM pg_settings WHERE name IN ('archive_mode');
 name      | setting | unit
-----+-----+-----+
 archive_mode | off     |
(1 row)
```

**Figure 5.6:** Disabled WAL Archive

## Recipe 32: Working with remote WAL Archive options

PostgreSQL WAL remote archiving is a feature that enables users to automatically archive WAL files from their PostgreSQL databases to a remote server. This feature allows users to offload the storage of their WAL files to a separate server, making it easier to maintain and manage backups of their databases. The process involves setting up a connection between the local and the remote server and then configuring the WAL archiving parameters in the local PostgreSQL instance. Once the setup is complete, the WAL files will be automatically sent to the remote server on a regular basis. This way, the local database can continue to operate without interruption, while the WAL files are safely stored on the remote server.

1. First, create an archive directory on a remote host (for example, NAS, S3, and so on) to store the archived files. Make sure that the user running PostgreSQL has read/write permissions on a directory on the remote host. For this recipe we have set-up an infrastructure with the following scenario, as shown in [Table 5.1](#):

Resource	Source server	Target server
Host info	PostgreSQL DB cluster on RHEL	File/backup server on RHEL
WAL path	<b>\$pg_wal</b>	<b>/pgwal_backup</b>
Access	Password-less SSH with remote user <b>dbftp</b>	Password-less SSH with local user <b>dbftp</b>

**Table 5.1:** Remote Archive scenario

2. Make sure that the remote server can be accessed from the source server using SSH keys (password-less SSH).

**Note: The method of configuring password-less SSH is beyond the range of this book.**

3. Next, configure PostgreSQL to enable WAL archiving. This is done by setting the **archive\_command** parameter in the **postgresql.conf** file (located in the Postgres data directory) or from the command line option. The **archive\_command** parameter must also be set to a valid command for archiving the WAL file segment to the remote host.

```
# set archive_command configuration to a remote  
server directory
```

```
alter system set archive_command = 'scp %p
dbftp@dbftpserver:/pgwal_backup/%f';
```

In the above archive command, the script has the following detail. The former part of the script **scp** is a Unix command-line utility used to securely copy files or directories between different locations. The **%p** symbol is essentially a placeholder for the path of the file that is being archived. This placeholder is replaced with the actual file path when the **archive\_command** is run. The latter part **dbftp@dbftpserver:/pgwal\_backup** is the detail for archiving on the target system and **%f** is replaced with the filename.

- Finally, restart PostgreSQL to enable WAL archiving. Once the server has been restarted, the PostgreSQL server will automatically archive the WAL files to the remote host.

*# Restart PostgreSQL database cluster service*

```
systemctl restart postgresql-15.service
```

*# Verify the remote archival configuration*

```
select name,setting,unit from pg_settings where name in
('archive_mode','archive_command','wal_keep_size','wal_level');
```

*Figure 5.7* is the execution output of the above script to check the remote archival configuration:

```
postgres=# select name,setting,unit from pg_settings where name in ('archive_mode','archive_command', 'wal_keep_size', 'wal_level');
          name           | setting          | unit
-----+-----+-----+
archive_command | scp %p dbftp@dbftpserver:/pgwal_backup/%f |
archive_mode    | on               |
wal_keep_size   | 160              | MB
wal_level       | replica          |
(4 rows)
```

*Figure 5.7: Remote archival set-up*

- Test the archiving process by processing some dataset and generating some WAL files. Once the dataset is processed, check the contents of the remote archive directory to verify that the WAL files were copied over successfully as shown in *Figure 5.8*:

```

postgres=# select archived_count, last_archived_wal, last_archived_time from pg_stat_archiver ;
      archived_count |    last_archived_wal    |    last_archived_time
-----+-----+-----+
          85 | 0000000100000010000000E4 | 2023-02-12 14:31:11.726458+08
(1 row)

[root@dbftpserver pgwal_backup]# ls -ltra
total 663492
dr-xr-xr-x. 18 root  root     244 Feb 12 14:08 ..
-rw-----.  1 dbftp dbftp 16777216 Feb 12 14:30 0000000100000010000000BB
-rw-----.  1 dbftp dbftp 16777216 Feb 12 14:30 0000000100000010000000BC
-rw-----.  1 dbftp dbftp 16777216 Feb 12 14:30 0000000100000010000000BD
-rw-----.  1 dbftp dbftp 16777216 Feb 12 14:30 0000000100000010000000BE
-rw-----.  1 dbftp dbftp 16777216 Feb 12 14:30 0000000100000010000000BF
-rw-----.  1 dbftp dbftp 16777216 Feb 12 14:30 0000000100000010000000C0
-rw-----.  1 dbftp dbftp 16777216 Feb 12 14:30 0000000100000010000000C1

```

*Figure 5.8: Remote Archival succeed*

### Recipe 33: Working with WAL compression option for space management

PostgreSQL 15 includes a new feature called **Zstandard (ZSTD)** compression which provides a significant improvement in compression performance compared to the existing PostgreSQL WAL compression algorithms (for example, Gzip). The new ZSTD compression algorithm provides approximately 2X higher compression ratio and up to 8X faster compression and decompression performance compared to Gzip. This makes it possible to reduce the size of WAL files, reduce the time taken for WAL archiving, and improve the overall performance of the PostgreSQL system.

PostgreSQL 15 supports WAL compression to store WAL data more efficiently. Let us start the recipe to demonstrate this in a realistic example:

1. The first step to this recipe is to download the PostgreSQL binary from the official PostgreSQL website (<https://ftp.postgresql.org/pub/source/v15.0/postgresql-15.0.tar.gz>) and extract it.
2. Refer to the recipe *Working with Installation from Source code* from *Chapter 1, Up and running with PostgreSQL 15*, for instructions on how to install PostgreSQL from source.

**Note: The only changes to be made in the Postgres source code installation is to add the configuration option --with-zstd while configuring the downloaded**

**source code binary as shown in the following script./configure --prefix=/pg\_install/pgv150/ --with-zstd. ZCompression can also be installed and configured by directly downloading the desired extension.**

3. Since the database is configured and the ZSTD extension is installed, at this stage we are good to create the ZSTD extension in our PostgreSQL database. This extension provides functions that allow you to compress and decompress data using the ZSTD algorithm. Refer to [Figure 5.9](#), to create an extension, login to the PostgreSQL database Instance with the superuser, we are using here Postgres as our superuser.

*# Create ZSTD Extension*

```
CREATE EXTENSION zstd;
```

*# Verify extension configuration*

```
select extname, extversion from pg_extension;
```

--Following the Output of above command---

extname	extversion
plpgsql	1.0
zstd	1.1.0

Please refer to the following figure:

```
postgres=# select extname, extversion from pg_extension;
 extname | extversion
-----+-----
 plpgsql | 1.0
 zstd    | 1.1.0
(2 rows)

postgres=# \dx
              List of installed extensions
   Name    | Version | Schema |          Description
-----+-----+-----+-----
 plpgsql | 1.0   | pg_catalog | PL/pgSQL procedural language
 zstd    | 1.1.0 | public   | Zstandard compression
(2 rows)
```

**Figure 5.9: ZSTD extension**

4. The default value for PostgreSQL 15's **wal\_compression** configuration setting is **off**. This setting determines that the

WAL compress is disabled. In order to enable **zstd** WAL compression, we need to enable the parameter **wal\_compression** to **on** in the **postgresql.conf** configuration file. The WAL compression setting can also be changed with the command line option. For this recipe we use **ZCompression** that is ZSTD, as shown in the following command:

```
# Enable WAL Compression  
ALTER SYSTEM SET wal_compression=zstd;  
# Verify WAL Compression configuration
```

```
show wal_compression;
```

```
--Following the Output of above command---
```

```
wal_compression
```

```
-----  
off
```

```
(1 row)
```

- Finally, you need to restart the PostgreSQL server for the changes to take effect and verify that the setting is enabled by running the command **SHOW wal\_compression**; again. At this time command output as WAL is enabled as shown in following command output section:

```
# Restart the PostgreSQL Instance  
systemctl restart postgresql-15.service  
# Verify WAL Compression configuration
```

```
SHOW wal_compression;
```

```
--Following the Output of above command---
```

```
wal_compression
```

```
-----  
zstd
```

```
(1 row)
```

## Explore SQL statement transaction sizing

The size of a transaction in Postgres is limited by the amount of memory available. The maximum size of a transaction can range from a few kilobytes to a few gigabytes, depending on the size of the system's memory and the complexity of the transaction.

## Recipe 34: Configuring and managing WAL performance parameter

PostgreSQL 15 provides advanced WAL performance parameters to help you get the most out of your database. These parameters can be configured using the **postgresql.conf** file in the data directory.

WAL performance parameters such as **wal\_buffers**, **wal\_writer\_delay**, **wal\_compression**, and **wal\_sync\_method** can be configured in PostgreSQL 15 to improve WAL performance.

- **wal\_buffers**: This parameter sets the number of shared memory buffers used for WAL operations. Increasing this parameter can help reduce the amount of disk I/O needed when writing WAL records. In general, the **wal\_buffers** setting should be adjusted in conjunction with the **max\_wal\_size** setting to achieve the best balance between write performance and disk usage.

However, it is important to keep in mind that increasing the **wal\_buffers** setting also increases the memory usage of the database, so setting the value based on the available memory resources and the specific needs and constraints of your application.

- **max\_wal\_size**: Increasing the **max\_wal\_size** setting in PostgreSQL 15 can help improve performance by increasing the amount of data that can be written to the WAL without forcing checkpoints. This can result in fewer frequent checkpoints, which can reduce the amount of time required to write the WAL data to disk and help reduce the amount of time queries wait for WAL data to be flushed.

However, setting **max\_wal\_size** too high can cause performance issues if the WAL files become too large. Therefore, it is important to find a balance between performance and disk space usage when setting **max\_wal\_size**.

- **wal\_writer\_delay**: This parameter sets the delay in milliseconds between WAL writes. Increasing this parameter can help reduce the amount of disk I/O needed when writing WAL records. In general, a low value may be appropriate for systems with a high write load, where the objective is to minimize write latency. On the other hand, a higher value may

be appropriate for systems with a low write load, where the objective is to minimize disk I/O overhead.

However, increasing the **wal\_writer\_delay** setting can also increase the amount of time it takes for WAL data to be flushed to disk and written to disk, which can result in longer recovery times in the event of a crash or system failure.

- **wal\_compression**: This parameter enables or disables WAL compression. Compressing WAL records can reduce the amount of disk I/O needed when writing WAL records.
- **wal\_sync\_method**: This parameter sets the method used for synchronizing WAL writes to disk. Setting this parameter to **open\_sync** can improve WAL write performance.
- **wal\_writer\_flush\_after**: This parameter controls how often the WAL writer process flushes the WAL data to disk. Increasing this parameter can improve performance, but it can also increase the risk of data loss in the event of a crash. It is essential to be cautious when adjusting this value, as increasing it too much may raise the risk of data loss in the event of a crash, as more unflushed data may be at risk if a failure occurs. Therefore, it is crucial to strike a balance between performance optimization and data durability while configuring **wal\_writer\_flush\_after**.

For this recipe we have prepared a **RHEL** system with 16 GB of memory and having 8 Core CPU. To configure WAL performance parameters in PostgreSQL 15, you need to edit the **postgresql.conf** configuration file. Following are the steps to configure WAL performance parameter:

1. The first step to this recipe is to locate the **postgresql.conf** file by running the following command from the **psql** prompt.
2. In this step, we will configure the WAL performance parameter for this, we have tried to test the WAL performance with two sets of values as detailed in [Table 5.2](#):

Configuration variable	Previous value	New value
<b>wal_buffers</b>	16MB	32MB
<b>max_wal_size</b>	1GB	4GB
<b>wal_writer_delay</b>	200ms	10000ms

<b>wal_compression</b>	off	on
------------------------	-----	----

**Table 5.2:** WAL performance parameter scenario

Now we are ready to modify the WAL performance parameter as detailed above using the following command option, the value for the WAL performance parameter may vary from case to case.

3. Finally, restart the PostgreSQL server for the changes to take effect and verify that the setting is enabled by running the following command.

## Recipe 35: Administering continuous archiving

Continuous archiving is a feature that allows you to continuously backup your WAL and store it in a remote archive. This feature is especially useful for organizations which needs to keep a copy of their database in a remote location for disaster recovery purposes.

To administer continuous archiving in PostgreSQL 15, first, you must activate the feature. This can be done by setting the **wal\_level** parameter to archive in your **postgresql.conf** configuration file. Additionally, you must configure the **archive\_command** parameter to specify the command used to archive the WAL files to the remote archive.

Once these settings are in place, WAL files will be written to the remote archive location whenever they are generated, allowing you to have a continuous backup of your database in a remote location. Additionally, you can configure additional parameters such as **archive\_timeout** and **archive\_mode** to customize the archiving process.

Finally, you should set up a monitoring system to ensure that the archiving process is running properly. This can be done by using PostgreSQL **log\_archive\_command** parameter to log the archiving process and set up email alerts if any errors occur.

## Managing the WAL writer process

The WAL writer process is a critical component of the PostgreSQL database system that ensures the durability of transactions. In PostgreSQL 15, the WAL Writer process works as follows:

**WAL records:** When a transaction modifies data in a PostgreSQL

database, the changes are first written to the transaction log (WAL) on disk. This ensures that the changes can be replayed in case of a crash or other failure.

**Dirty pages:** The modified pages are also marked as dirty in the shared buffer pool in memory. The dirty pages contain changes that have not yet been written to the data files on the disk.

**WAL buffers:** The WAL writer process periodically writes the contents of the WAL buffers to the WAL on disk. The WAL buffers contain changes that have not yet been written to the data files on the disk.

**Flushing WAL buffers:** When the WAL buffers become full, the WAL writer process flushes them to the WAL on disk, ensuring that all changes are written to the WAL before the buffer is reused.

**Checkpoint trigger:** The WAL writer process is also responsible for triggering the checkpoint process, which ensures that all dirty pages are written to disk and all committed transactions are safely stored in the WAL.

**Performance tuning:** The WAL writer process can be tuned for performance using the `wal_writer_delay` and `max_wal_size` configuration parameters. These parameters control the frequency of WAL writes and the size of the WAL on disk, respectively.

Overall, the WAL writer process is critical to ensuring the durability and reliability of the PostgreSQL database system. By writing changes to the WAL on disk before writing them to the data files on disk, the WAL writer process ensures that the database can be recovered to a consistent state in the event of a crash or other failure.

## PostgreSQL 15 WAL writer process versus checkpoint processing

The PostgreSQL 15 WAL writer process is responsible for writing the WAL to disk. This process is constantly working in the background, writing transaction changes to the WAL as they happen.

The checkpoint process is responsible for ensuring that the data in the database is consistent with the WAL. When a checkpoint occurs, the checkpoint process writes a checkpoint record to the WAL, which tells the WAL writer process to flush all WAL records up to that point to disk. The checkpoint process also writes dirty pages from memory

to disk, so that any changes that were not yet written to the WAL are also persisted. This ensures that the database can be restored to a consistent state in the event of a crash.

The main difference between the WAL writer process and the checkpoint process is their focus. The WAL writer process is focused on writing WAL records to disk to ensure durability, while the checkpoint process is focused on ensuring consistency between the data in memory and the WAL. The WAL writer process writes to the WAL files continuously, while the checkpoint process is triggered periodically or on-demand. Further differences between the WAL writer process and checkpoint process are discussed in [Table 5.3](#):

Features	WAL writer process	Checkpoint process
Responsibility	Write changes to the WAL on disk	Ensure consistency between data and WAL
Frequency of operation	Continuous	Periodic or on-demand
Triggered by	Committing a transaction	Periodic timer or manual command
Writes to WAL files	Yes	Yes
Writes dirty pages to disk	No	Yes
Flushing the WAL	Periodically or on-demand	At the time of a checkpoint
Goal	Ensure durability of the database	Ensure consistency of the database

**Table 5.3: WAL writer versus checkpoint**

Overall, the WAL writer process and the checkpoint process work together to ensure the reliability and consistency of the PostgreSQL database system.

## Optimizing checkpoint processing

The checkpoint process is responsible for ensuring the consistency of the database by synchronizing the data in memory with the WAL on disk. The checkpoint process works in the following way:

1. The checkpoint process is triggered either periodically or on-demand. The default configuration triggers a checkpoint every 5 minutes, but this can be adjusted in the PostgreSQL configuration.

2. The checkpoint process scans the shared buffer pool to identify any dirty pages that have been modified in memory but have not yet been written to the WAL on disk.
3. The checkpoint process writes a checkpoint record to the WAL, which tells the WAL writer process to flush all WAL records up to that point to disk. This ensures that all committed transactions have been written to the WAL and are available for crash recovery.
4. The checkpoint process then writes all the dirty pages to disk, using a technique known as background writer to minimize the impact on system performance. The background writer writes the dirty pages in small batches, rather than all at once, to avoid overwhelming the disk I/O subsystem.
5. Once all the dirty pages have been written to disk, the checkpoint process updates the control file to record the checkpoint location in the WAL. This allows PostgreSQL to know the point in the WAL from which it can start recovery in the event of a crash.

The checkpoint process is critical to ensuring the consistency and reliability of the PostgreSQL database system. By synchronizing the data in memory with the WAL on disk, the checkpoint process ensures that the database can be restored to a consistent state in the event of a crash or other failure. It also helps to minimize the time required for crash recovery, since it ensures that all committed transactions are safely stored in the WAL on disk.

## **Automating vacuum process**

PostgreSQL 15 uses the same vacuum process as the previous versions. The process removes dead tuples from the tables. It also marks the table as needing a full scan when it is next accessed. The process also reclaims storage space by rewriting pages on disk. The process can be done manually, or it can be set to run automatically at regular intervals.

This process is responsible for performing the following operations:

- Marking dead tuples as eligible for removal and removing them.

- Re-ordering tuples within the table to reduce the amount of disk I/O needed to fetch them.
- Analysing tables to update system catalogs with accurate statistics.
- Updating visibility information for rows that were modified or deleted by transactions that have already committed.
- Updating indexes by removing empty index pages.
- Updating visibility map to indicate which pages contain only visible tuples.
- Modifying free-space map to record newly available disk space.
- Updating **relfrozenxid** values to prevent transaction wraparound.
- Upgrading **datfrozenxid** values to prevent database wraparound.
- Updating **relfrozenxid** values to prevent table wraparound.
- Adjusting **relminmxid** values to prevent index wraparound.
- Performing garbage collection operations to physically remove dead rows or tuples.
- Performing table-level locks to prevent concurrent transactions from accessing the table until the **VACUUM** operation is complete.

The vacuum process in PostgreSQL provides several options to run in a specific event, following the vacuum process options:

- **Full vacuum:** This option is used to reclaim disk space and reduce the size of the database. It reclaims disk space by removing deleted or obsolete data in the database.
- **Analyze:** This option is used to update the statistics used by the query planner. It is recommended to use this option after a full vacuum to improve query performance.
- **Freeze:** This option is used to freeze all the tuples in a table. This can help improve performance for read-only operations.
- **Verbose:** This option is used to display the progress of the vacuum process.

- **Truncate:** This option is used to truncate a table. It permanently removes all data from the table and cannot be undone.
- **Index:** This option is used to vacuum indexes. It can improve query performance by removing obsolete index entries.
- **Table:** This option is used to vacuum tables. It can improve query performance by reclaiming disk space and reducing the size of the table.
- **Auto vacuum:** This option is used to enable automatic vacuuming of the database. This can improve query performance and reclaim disk space without manual intervention.

## **Recipe 36: Getting insight to vacuum process in PostgreSQL database**

The PostgreSQL 15 vacuum process can be imagined as a janitorial process that cleans up the database. The process starts by analyzing the database to identify objects that need to be cleaned up or removed. The vacuum process then removes any stale data, clears out any old indexes, and reclaims any unused space. By doing this, the database is optimized for performance. The vacuum process can also detect and fix any inconsistencies in the database, ensuring data integrity. After the vacuum process is complete, the database is ready for use.

This recipe is all about getting an insight into the vacuum process. Suppose we have a PostgreSQL database with a table called **userlogging** which stores user logging information for the last ten years. A batch job is executed on this table which exports data older than 5 years into the flat file and then deletes this data from the **userlogging** table. Consequently, the table size remains the same even after the data is removed from the **userlogging** table. This is because after deleting the data, the deleted rows/tuples remain intact with the data file, indeed it is marked as deleted in the **xmax** field.

Now the vacuum process comes into the picture. It reclaims the space occupied by the deleted tuples and makes the size of the table smaller. The vacuum process works by scanning the table and looking for the tuples that are marked as deleted. It then removes

those tuples from the data file and makes it available to the system for use.

So, let us begin the recipe based on the above dataset information. By following these steps, we can get an insight of the vacuum procedure.

1. The first step to this recipe is to identify the size of the **userlogging** table and the dead tuples using the following script:

```
# Verify the size of the userlogging table
SELECT pg_size.pretty (pg_total_relation_size ('userlogging'));
# Execute pgstattuple() function to get information
about a table.

select * from pgstattuple('userlogging');
```

Referring to [Figure 5.10](#) of Step 1, there are no deleted rows or dead tuples present in the column **n\_dead\_tup** for **userlogging** table. That makes it easier for us to understand from scratch that any further changes in the dataset by the batch clean-up job will generate that deleted row count for the column **n\_dead\_tup** of the **pg\_stat\_user\_tables**. Following the dataset detail for table **userlogging** prior to running the batch clean-up job.

- The size of the table is 16 GB
- Zero deleted rows/dead tuples present in **userlogging** table.

The screenshot shows the pgAdmin interface with two query panes and their corresponding results.

**Query 1:**

```
5 SELECT pg_size.pretty (pg_total_relation_size ('userlogging'));
```

**Data Output:**

	pg_size.pretty
1	16 GB

**Query 2:**

```
5 SELECT n_live_tup, n_dead_tup, relname FROM pg_stat_user_tables WHERE relname = 'userlogging';
```

**Data Output:**

	n_live_tup	n_dead_tup	relname
1	262020566	0	userlogging

**Figure 5.10:** Table size and dead tuple before batch clean-up

2. Now we are good the execute the batch clean-up job.

**Note: Batch clean-up job is optional, for the representation scenario this step not included in this recipe.**

3. Since batch clean-up job was successfully executed in the previous step, this step will identify the size of the **userlogging** table and the dead tuples using the following script:

The screenshot shows three separate query panes in pgAdmin:

- Query 1:** `SELECT pg_size.pretty (pg_total_relation_size ('userlogging'));`. The result is a single row: 16 GB.
- Query 2:** `SELECT n_live_tup, n_dead_tup, relname FROM pg_stat_user_tables WHERE relname = 'userlogging';`. The result is a single row: n\_live\_tup = 223910566, n\_dead\_tup = 38110000, relname = userlogging.
- Query 3:** `select * from pgstattuple('userlogging');`. The result is a single row with various statistics: table\_len = 11602558976, tuple\_count = 223910566, tuple\_len = 8284690942, tuple\_percent = 71.4, dead\_tuple\_count = 336, dead\_tuple\_len = 12432, dead\_tuple\_percent = 0, free\_space = 1558383448, free\_percent = 13.43.

**Figure 5.11:** Table size and dead tuple after batch clean-up

As shown in [Figure 5.11](#), almost 14% of data is deleted from **userlogging** table but as per the [Figure 5.11](#), the size of the **userlogging** still remains same. In the next step, we will be performing the **VACUUM** on **userlogging** table to remove the dead tuples.

4. Refer to [Figure 5.12](#), now, perform the vacuum process on the table by using the **VACUUM** command. This command scans the table and removes the deleted tuples from the data file. It also reclaims the space occupied by the deleted tuples and makes the size of the table smaller. The vacuum process, after being executed, scans the table and looks for

the tuples that are marked as deleted. It then removes those tuples from the data file and makes it available to the system for use. Please refer to the following figure:

```
s=# VACUUM (full, VERBOSE)userlogging;
vacuuming "public.userlogging"
"public.userlogging": found 336 removable, 223910566 nonremovable row versions in 1416328 pages
    0 dead row versions cannot be removed yet.
er: 79.56 s, system: 23.67 s, elapsed: 116.72 s.
```

s=#

**Figure 5.12:** Vacuum userlogging table

- After performing the vacuum process, analyze the **userlogging** table to ensure that the size of the table is reduced and all the deleted tuples are removed from the data file using the following command:

The screenshot shows three separate query results in pgAdmin:

- Query 1:** `SELECT pg_size_pretty (pg_total_relation_size ('userlogging'));`

pg_size_pretty	
text	
1	14 GB

- Query 2:** `select * from pgstattuple('userlogging');`

table_len	tuple_count	tuple_len	tuple_percent	dead_tuple_count	dead_tuple_len	dead_tuple_percent	free_space	free_percent
9915006976	223910566	8284690942	83.56	0	0	0	29052888	0.29

- Query 3:** `SELECT n_live_tup, n_dead_tup, relname FROM pg_stat_user_tables WHERE relname = 'userlogging';`

n_live_tup	n_dead_tup	relname
223910566	0	userlogging

**Figure 5.13:** Table size and dead tuple post vacuum on userlogging table

Referring to [Figure 5.13](#), the conclusion of the PostgreSQL **VACUUM** process is that it is an effective way to maintain database performance, integrity, and consistency. It helps to prevent bloating of the database and reclaims unused space, which can improve query performance. It also helps to detect and fix potential corruption issues. **VACUUM** is an important

part of the regular database maintenance routine, and it should be performed regularly for the best results.

## Recipe 37: Debug PostgreSQL autovacuum problem

Debugging PostgreSQL 15 autovacuum problems can involve several steps, depending on the nature of the issue. Here are a few possible steps to take when debugging autovacuum problems in PostgreSQL 15:

1. **Check the PostgreSQL log files:** The log files may contain useful information about any errors or warnings that occur during the autovacuum process. Look for messages related to autovacuum, such as **autovacuum started** or **autovacuum: found N removable, Y nonremovable row versions**.
2. **Check the PostgreSQL configuration:** Make sure that autovacuum is enabled and that the configuration settings are appropriate for your workload. For example, you may need to adjust the **autovacuum\_vacuum\_scale\_factor** or **autovacuum\_analyze\_scale\_factor** parameters to ensure that autovacuum runs frequently enough to keep up with your database workload.
3. **Use the pgstattuple extension:** The **pgstattuple** extension provides detailed information about the contents of each table, including the number of dead rows and the amount of free space. You can use this information to identify tables that may need to be vacuumed or analyzed more frequently.
4. **Use the pg\_stat\_all\_tables view:** The **pg\_stat\_all\_tables** view provides information about the number of dead rows and the amount of free space for each table in the database. You can use this view to identify tables that may be causing autovacuum problems.
5. **Manually vacuum problematic tables:** If you identify a table that is causing autovacuum problems, you can manually vacuum the table to free up space and reduce the number of dead rows. You can also manually analyze the table to update the statistics used by the query planner.
6. **Consider increasing the autovacuum daemon's verbosity:** By setting the **log\_autovacuum\_min\_duration** parameter to

a lower value, you can make the autovacuum daemon log more information, which can help you diagnose problems.

These are just a few possible steps to take when debugging PostgreSQL 15 autovacuum problems. The specific steps to take will depend on the nature of the issue and the information available.

## Implementing continuous archiving and archive logs

PostgreSQL 15 provides continuous archiving for point-in-time recovery of data. Continuous archiving works by taking regular snapshots of the data and saving them to an external archive. The archive logs contain all the information required to replay the data as it was at the time the snapshot was taken. This allows users to restore the database to any point in time, even if the database has been corrupted or lost. Continuous archiving also helps to protect against data loss due to hardware or software failure.

In PostgreSQL, continuous archiving is enabled by default, and you can configure it using the **wal\_level**, **archive\_mode**, and **archive\_command** configuration parameters. The **wal\_level** parameter determines the level of detail to be included in the WAL, while the **archive\_mode** parameter determines whether continuous archiving is enabled or not. When archiving is enabled, PostgreSQL will write the archived transaction logs to the **archive\_command** location. These logs can then be used to restore the PostgreSQL server to a prior point in time.

These features make it easy for users to maintain a secure and reliable copy of their database and to quickly restore it if needed.

## Recipe 38: PostgreSQL 15 continuous archiving and archive logs examples

These features make it easy for users to maintain a secure and reliable copy of their database and to quickly restore it if needed.

1. The first step to this recipe is to configure **archive\_mode**, **wal\_level** and **archive\_command** in the **postgresql.conf** configuration file. This can also be done by running the following command:

```
# Enable Archiving
```

```
alter system set archive_mode = on;
```

```

# Configure wal_level setting
alter system set wal_level = replica;
# set archive_command configuration to a valid
# directory
alter system set archive_command= 'test ! -f /wal_archive/%f && cp %p
/wal_archive/%f';

```

2. The next step is to restart the PostgreSQL server for the changes to take effect. Once enabled, the WAL files will be stored in a separate directory and can be used for point-in-time recovery if needed:

```

# Restart PostgreSQL service
systemctl restart postgresql-15.service
# Verify the WAL configuration
select name,setting,unit from pg_settings where name in
('archive_mode','archive_command','wal_level');

```

Please refer to the following figure:

	name	setting
	text	text
1	archive_command	test ! -f /wal_archive/%f && cp %p /wal_archive/%f
2	archive_mode	on
3	wal_level	replica

**Figure 5.14:** Verify WAL archival post restart

3. In this step will create a physical backup of the PostgreSQL database cluster using the following command:

```

# Create a physical backup
pg_basebackup -D /Pg_backup/

```

Refer to [Figure 5.15](#). With the execution of the above command, PostgreSQL will create a complete copy of the entire database cluster in the specified backup directory:

```

-bash-4.2$ pg_basebackup -D /Pg_backup/
-bash-4.2$ ls /Pg_backup/
backup_label  current_logfiles  pg_commit_ts  pg_ident.conf  pg_notify  pg_snapshots  pg_subtrans  PG_VERSION  postgresql.auto.conf
backup_manifest  global  pg_dynshmem  pg_logical  pg_replslot  pg_stat  pg_tblspc  pg_wal  postgresql.conf
base          log  pg_hba.conf  pg_multixact  pg_serial  pg_stat_tmp  pg_twophase  pg_xact
-bash-4.2$ 

```

**Figure 5.15:** Execute physical backup

4. At this stage we are in a state to perform the **point-in-time-recovery (PITR)** by using the physical backup and WAL file. But prior to that perform the following perquisite, as follows:

- Verify the PostgreSQL database cluster data directory path using the following command:

```
# List PostgreSQL data directory  
show data_directory;
```

- Stop the PostgreSQL database cluster.
- Remove the data directory of the PostgreSQL cluster or rename it to the backup directory if sufficient space available. (For this recipe we rename the data directory to **backup\_data**).
- Create a new data directory with the same ownership as the previous data directory and make sure that the data directory is empty.

We have now reached a stage where we can proceed with restoring the database that was backed up in step 3 of this recipe.

```
# Restore the database with the unix copy utility  
cp -a /Pg_backup/. /var/lib/pgsql/15/data
```

Refer to [Figure 5.16](#). With the command script WAL was also copied from **/Pg\_backup** to the **pg\_wal** path. If the WAL was generated post physical backup, then ensure to make a backup copy the WAL before removing the data directory to further restore for PITR recovery. Please refer to the following figure:

```
-bash-4.2$ cp -a /Pg_backup/. /var/lib/pgsql/15/data/  
-bash-4.2$ _
```

**Figure 5.16:** Restore backup with Unix copy utility

- This step is all about restoring the database by configuring the **restore\_command** parameter by providing the path to the WAL archive directory that PostgreSQL will use to restore archived WAL segments. This WAL archive directory is an **archive\_command** archive path that we configured in step 1 of this recipe.

```
# Restore archive WAL Segment with configuration in  
postgresql.conf
```

```
restore_command = 'cp /wal_archive/%f %p'
```

Here, the **restore\_command** parameter specifies the **cp** command to copy the archived WAL segment file **%f** to the location specified by **%p**.

6. The next step is to configure the recovery target, the **recovery\_target** parameter is used to specify the target point for database recovery. It allows you to control the point in time or transaction to which you want to recover your database.

Following are the desired target point for database recovery by using **recovery\_target** configuration:

- a. **point in time**: This option allows you to recover the database to a specific point in time by specifying the time as a string in the format YYYY-MM-DD HH:MM:SS.
- b. **transaction ID**: This option allows the recovery of the database up to a specific transaction ID.
- c. **LSN**: This option allows you to recover the database up to a specific Log Sequence Number.
- d. **Immediate**: This option specifies that the recovery process should stop as soon as possible, without applying any additional WAL records.

For this recipe we are not defining any **recovery\_target** parameter option, if no **recovery\_target** option is specified, the server will attempt to perform a full database recovery, using all available WAL segments.

7. Finally, we are good to start the PostgreSQL database cluster service but prior to that manually create a **recovery.signal** file in the data directory. The **recovery.signal** file is a control file that is created during the recovery process to signal the end of the recovery process.

When the PostgreSQL server is started in recovery mode it will look for a **recovery.signal** file in the **data** directory. If the file exists, the server will stop the recovery process and bring the database online.

```

# create a recovery.signal file
touch /var/lib/pgsql/15/data/recovery.signal
# Start PostgreSQL database cluster service
systemctl restart postgresql-15.service

[root@postgresdev 15]# touch /var/lib/pgsql/15/data/recovery.signal
[root@postgresdev 15]# systemctl start postgresql-15.service
[root@postgresdev 15]#

```

**Figure 5.17:** Create signal file for PITR

The PostgreSQL database cluster service is successfully recovered and ready for database operations.

## Enhancing performance and benefits of WAL

PostgreSQL 15 brings several notable improvements in WAL. The most important improvement is the introduction of parallel replay of WAL records. This allows multiple WAL records to be replayed simultaneously, resulting in improved performance for large workloads. Additionally, PostgreSQL 15 includes optimizations for improved WAL logging of standby servers, allowing for faster recovery times.

PostgreSQL 15 also introduces new features for compressing WAL records, which can reduce the amount of disk space required for WAL storage. This can help reduce the overall storage requirements for PostgreSQL databases, making them easier to manage and maintain.

The WAL feature in PostgreSQL 15 provides several performances and other benefits, including:

**Durability:** With WAL, changes made to the database are first written to a log file before they are applied to the actual database files. This ensures that data modifications are durable and recoverable in the event of a crash or other failure.

**Faster recovery:** Since WAL keeps a record of all changes made to the database, it is possible to replay the changes in the event of a crash or other failure, rather than having to perform a time-consuming full recovery from a backup.

**Reduced I/O:** Since WAL records only the changes made to the database, rather than the entire contents of a table, it results in less

I/O overhead and faster write performance.

**Increased concurrency:** Since WAL records changes made by multiple transactions in a serialized order, it allows multiple transactions to concurrently modify the same table without conflicts.

**Point-in-time recovery:** WAL makes it possible to restore the database to any point in time, using the archived WAL files.

**Online backup:** Since WAL records changes as they occur, it is possible to perform online backups of a PostgreSQL database without the need to take it offline.

## Recipe 39: Proactive solution to delete PostgreSQL archive logs

The proactive solution for PostgreSQL archive log deletion involves the use of automatic archiving and log rotation. This process involves setting up an automatic archiving process, which creates an archive log file every time the database is backed up. This archive log file should be configured to be automatically deleted after a certain period. In addition, log rotation should be set up so that older log files are deleted when new logs are created. This will ensure that only the most recent log files are kept, while the older log files are deleted. This will help keep the database size manageable, as well as reduce the risk of data loss due to log file corruption.

**archive\_cleanup\_command** is a utility program for cleaning up archived files created by PostgreSQL. The utility is designed to be used in conjunction with the **archive\_command** configuration parameter, which is used to archive the WAL files for backup and recovery purposes. It is used to keep the size of the WAL Archive directory from growing too large. It removes archived WAL files that are no longer needed for the server to recover to a consistent state in the event of a crash or other failure. **archive\_cleanup\_command** should be used regularly to prevent the WAL Archive directory from becoming too large.

The syntax for the PostgreSQL **archive\_cleanup\_command** is:

```
pg_archivecleanup [option...] archivolocation oldestkeptwalfile
```

Here **archivolocation** is the directory containing the archived WAL files. **oldestkeptwalfile** is the number of archived WAL files to retain. This command is used to keep the archive directory clean, as

well as to save disk space. It is also important to note that this command should be used with caution as any archived WAL files that are deleted cannot be recovered. So, start the recipe for the clean-up of archive logs in the PostgreSQL database using the following steps:

1. Firstly, assume that you have PostgreSQL 15 database cluster instance running as shown in the [Figure 5.18](#):

```
t@postgresdev ~]# systemctl status postgresql-15.service
postgresql-15.service - PostgreSQL 15 database server
   Loaded: loaded (/usr/lib/systemd/system/postgresql-15.service; enabled; vendor preset: disabled)
     Active: active (running) since Tue 2023-01-03 13:26:59 +08; 2 weeks 4 days ago
       Docs: https://www.postgresql.org/docs/15/static/
    Process: 1096 ExecStartPre=/usr/pgsql-15/bin/postgresql-15-check-db-dir ${PGDATA} (code=exited, status=0/SUCCESS)
      Main PID: 1122 (postmaster)
        Tasks: 19 (limit: 30865)
       Memory: 72.2M
          CPU: 805ms
         CGroup: /system.slice/postgresql-15.service
                 └─1122 /usr/pgsql-15/bin/postmaster -D /var/lib/pgsql/15/data/
                     ├─1254 "postgres: logger"
                     ├─1263 "postgres: checkpointer"
                     ├─1264 "postgres: background writer"
                     ├─1281 "postgres: walwriter"
                     ├─1282 "postgres: autovacuum launcher"
                     ├─1285 "postgres: archiver failed on 000000010000000000000006F"
                     ├─1287 "postgres: logical replication launcher"
                     ├─1802 "postgres: postgres postgres 192.168.187.134(33530) idle"
                     ├─1807 "postgres: postgres postgres 192.168.187.134(33532) idle"
                     ├─1813 "postgres: postgres postgres 192.168.187.134(33540) idle"
                     ├─1820 "postgres: postgres postgres 192.168.187.134(33552) idle"
                     ├─3142 "postgres: postgres postgres 192.168.187.134(55754) idle"
                     ├─3144 "postgres: postgres postgres 192.168.187.134(55768) idle"
                     ├─3145 "postgres: postgres postgres 192.168.187.134(55784) idle"
                     ├─3148 "postgres: postgres postgres 192.168.187.134(55814) idle"
                     ├─3150 "postgres: postgres postgres 192.168.187.134(55818) idle"
                     ├─3151 "postgres: postgres postgres 192.168.187.134(55822) idle"
                     └─3152 "postgres: postgres postgres 192.168.187.134(55828) idle"

03 19:53:49 pgsqldev systemd[1]: Starting PostgreSQL 15 database server...
03 19:53:50 pgsqldev postmaster[1122]: 2023-01-03 19:53:50.257 +08 [1122] LOG:  pgaudit extension initialized
03 19:53:50 pgsqldev postmaster[1122]: 2023-01-03 19:53:50.526 +08 [1122] LOG:  redirecting log output to logging collector proc
03 19:53:50 pgsqldev postmaster[1122]: 2023-01-03 19:53:50.526 +08 [1122] HINT:  Future log output will appear in directory "log"
03 19:53:50 pgsqldev systemd[1]: Started PostgreSQL 15 database server.
t@postgresdev ~]#
```

**Figure 5.18: PostgreSQL service status**

2. After establishing the connection to the PostgreSQL database cluster, the first step is to list the archive directory of our PostgreSQL instance using the following command:

### # Command to list the Archive directory

```
select name,setting,unit from pg_settings where name in
('archive_mode','archive_command','archive_timeout');
```

### # Following the output of the above command

name	setting	unit
archive_command	test ! -f /wal_backup/%f && cp %p /wal_backup/%f	
archive_mode	on	
archive_timeout	21600	s
(3 rows)		

From the above output we can understand the **/wal\_backup** is the directory that we are using for storing the WAL archive.

3. In the next step will configure the WAL archive clean-up by creating a shell script that will perform the clean-up using the following script:

```
#!/bin/bash
# Define the archive directory and the retention
period
ARCHIVE_DIR="/wal_backup"
RETENTION_PERIOD=7
# Get the name of the oldest required WAL file
OLDEST_WAL_FILE=$1
# Calculate the cutoff date
CUTOFF_DATE=$(date +%Y-%m-%d --date="${RETENTION_PERIOD}
days ago")
# Find the outdated WAL files
OUTDATED_WAL_FILES=$(find "${ARCHIVE_DIR}" -name "*.backup" -
type f -mtime +${RETENTION_PERIOD})
# Delete the outdated WAL files
for FILE in ${OUTDATED_WAL_FILES}
do
rm -f "${FILE}"
done
```

Here, the script defines the archive directory and the retention period and uses the find command to search for WAL files older than the retention period. The outdated WAL files are then deleted using the **rm** command.

4. Let us make the script executable that we have created in step 3 of this recipe.

```
chmod +X1 arch_cleanup.sh
```

5. Since the WAL archive clean-up script is ready the next step is to configure the **archive\_cleanup\_command** in the PostgreSQL database cluster for clean-up of archive using the script created in step 3 of this recipe and restart the PostgreSQL service for the changes to take effect.

```
# Configure the archive_cleanup_command using
following command
```

```
alter system set archive_cleanup_command =
'/mnt_script/arch_cleanup.sh %r;
```

**# command to list the Archive configuration**

```
select name,setting,unit from pg_settings where name in
('archive_mode','archive_command','archive_timeout',
'archive_cleanup_command');
```

**# Following the output of the above command**

name	setting	unit
archive_cleanup_command		
archive_command	test ! -f /wal_backup/%f && cp %p /wal_backup/%f	
archive_mode	on	
archive_timeout	21600	s

(4 rows)

6. Finally, reload the PostgreSQL for the changes to take effect and verify that the setting is enabled by running the following command:

**# Reload PostgreSQL service**

```
select pg_reload_conf();
```

**# command to list the Archive configuration**

```
select name,setting,unit from pg_settings where name in
('archive_mode','archive_command','archive_timeout',
'archive_cleanup_command');
```

**# Following the output of the above command**

name	setting	unit
archive_cleanup_command	/mnt_script/arch_cleanup.sh	
archive_command	test ! -f /wal_backup/%f && cp %p /wal_backup/%f	
archive_mode	on	
archive_timeout	21600	s

(4 rows)

## Conclusion

In conclusion, the transaction log is a critical component of PostgreSQL that ensures data consistency and durability. It enables

point-in-time recovery and replication, and it provides a mechanism for analyzing and interpreting data changes. In this chapter, we have covered the key concepts related to the transaction log in PostgreSQL, including the WAL protocol, configuration settings, and best practices for usage and maintenance. By understanding how the transaction log works and how it can be used effectively, database administrators and developers can ensure that their PostgreSQL databases remain reliable and robust, even in the face of hardware or software failures.

In the next chapter, we will continue to build upon the foundations of database performance and scalability, focusing specifically on *Partitioning and Sharding* in PostgreSQL. With the objectives of providing a comprehensive understanding of these advanced techniques, developers and database administrators will gain valuable insights into optimizing large datasets for improved efficiency and distributed data management.

## **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

# Partitioning and Sharding

## Introduction

As data volumes continue to grow and the need for high-performance, scalable databases becomes increasingly important. Partitioning and sharding are two techniques that can be used to improve database performance and scalability. Partitioning involves splitting a large table into smaller, more manageable pieces, while sharding distributes data across multiple servers or nodes. In this chapter, we will explore how to implement partitioning and sharding in PostgreSQL, including different techniques, tools, best practices, and challenges. We will also discuss the benefits of partitioning and sharding, including improved query performance, better resource utilization, and simplified data maintenance tasks.

## Structure

In this chapter, we will cover the following topics:

- Partitioning and sharding
- Partitioning classifications
- Partitioning versus sharding
- Shard strategies and rebalancer
- Explain foreign data wrapper

## Objectives

The objective of writing a chapter on *Partitioning and Sharding* in PostgreSQL is to provide developers and database administrators with

an in-depth understanding of the techniques and tools used to optimize database performance and scalability. This chapter will explore how partitioning and sharding can be used to split a large table into smaller, more manageable pieces, and distribute data across multiple servers or nodes. By providing guidance on best practices and tools, this chapter will help readers design and implement robust and efficient partitioned or sharded databases in PostgreSQL.

## Partitioning and sharding

Partitioning and sharding are both techniques used to improve the performance and scalability of databases.

Partitioning is the process of dividing a large database into smaller, more manageable chunks called partitions or shards. Each partition contains a subset of the data and is stored separately. This allows for faster access and retrieval of the data, as well as improved scalability and availability.

Sharding, on the other hand, is a specific form of horizontal partitioning that involves distributing partitions or shards across multiple physical or logical servers in a cluster. Each shard is assigned to a different server, which is responsible for storing and managing the data within that shard. This technique is often used in large-scale distributed databases where a single server may not be able to handle the volume of requests.

Both partitioning and sharding require careful planning and design to ensure that the data is distributed in a way that maximizes performance and avoids issues such as data skew or hotspots. Additionally, these techniques may require changes to the application layer to support the distributed nature of the data. However, when implemented correctly, partitioning and sharding can be powerful tools for managing large and complex databases.

## Partitioning classifications

PostgreSQL 15 includes support for partitioning tables, which allows for better performance and manageability for large tables, and can be done using two methods: range partitioning and list partitioning:

- **Range partitioning:** Range partitioning divides a table into several tables based on a range of values of a specified column. For example, a range partitioned table can be divided into

multiple tables based on the range of values for a certain date column.

- **List partitioning:** List partitioning divides a table into several tables based on a list of values of a specified column. For example, a list partitioned table can be divided into multiple tables based on a list of cities, state or country.

In addition to range and list partitioning, PostgreSQL also supports hash and composite partitioning:

- **Hash partitioning:** This type of partitioning is useful for evenly distributing data across partitions partitioning tables based on the result of a hash function applied to the partition key. The partition key is a column or a set of columns in a table used to split the table into partitions. Additionally, hash partitioning allows for easier sharding of data, which can improve scalability for larger databases.
- **Composite partitioning:** It allows for the partitioning of a single table into multiple partitions based on multiple partition keys or expressions. In other words, instead of using a single column as the partition key, composite partitioning allows you to use a combination of two or more columns as the partition key.

## Recipe 40: Setup and exploit partitioning in PostgreSQL

To begin setting up partitioning in PostgreSQL 15, create the table and set up the columns to be used for partitioning and can be done using the **CREATE TABLE** command with the **PARTITION BY** clause. Next, create a partitioned table that inherits from the parent table.

To review, the steps for setting up partitioning in PostgreSQL 15 are:

1. The first step to this recipe is to choose a partitioning key. The partitioning key is the column that will be used to partition the data. For example, in a scenario, we might choose a column such as **sale\_date**. This will depend on the nature of the data and how it will be accessed.
2. Create a table to act as the parent table that will contain the partitioned data. This is the table that will be partitioned by **RANGE**. For this example, we will create a table called **sales**.

```
# create a range partitioned table
```

```
CREATE TABLE sales (
```

```
    sale_id SERIAL,
```

```
sale_date DATE NOT NULL,  
sale_amount NUMERIC(10,2) NOT NULL,  
PRIMARY KEY (sale_id, sale_date)  
) PARTITION BY RANGE (sale_date);
```

In this example, the **PARTITION BY RANGE** clause specifies that the table will be partitioned based on the **sale\_date** column.

3. Next, we need to create the partition tables. These are the tables that will hold the actual data. We will create partition tables based on ranges of sale dates.

#### # create partition table

```
CREATE TABLE sales_q1_2k22 PARTITION OF sales FOR VALUES FROM  
('2022-01-01') TO ('2022-04-30');  
  
CREATE TABLE sales_q2_2k22 PARTITION OF sales FOR VALUES FROM  
('2022-05-01') TO ('2022-08-31');  
  
CREATE TABLE sales_q3_2k22 PARTITION OF sales FOR VALUES FROM  
('2022-09-01') TO ('2022-12-31');  
  
CREATE TABLE sales_q1_2k23 PARTITION OF sales FOR VALUES FROM  
('2023-01-01') TO ('2023-04-30');  
  
CREATE TABLE sales_q2_2k23 PARTITION OF sales FOR VALUES FROM  
('2023-05-01') TO ('2023-08-31');  
  
CREATE TABLE sales_q3_2k23 PARTITION OF sales FOR VALUES FROM  
('2023-09-01') TO ('2023-12-31');
```

These commands create a tables based on FY 2022 and 2023, that are partitions of the sales table.

The **FOR VALUES** clause specifies the range of values that belong in each partition. In this example, the **sales\_q1\_2k22** table will hold all rows where the **sale\_date** is between January 1, 2022, and April 30, 2022, and the **sales\_q2\_2k22** table will hold all rows where the **sale\_date** is between May 1, 2022, and August 31, 2022. Similarly the table was further partitioned into following financial year.

4. Now we will insert data into the partitioned table. When we insert data, PostgreSQL will automatically route each row to the appropriate partition based on the partition key.

#### # Insert Data into partition table

```
INSERT INTO sales (sale_date, sale_amount) VALUES  
('2022-01-12', 1000.00),  
('2022-05-15', 2000.00),
```

```
('2022-09-19', 1000.00),  
('2023-01-12', 1000.00),  
('2023-05-15', 2000.00),  
('2023-09-19', 1000.00);
```

With the above script:

- The 1<sup>st</sup> row will be inserted into the **sales\_q1\_2k22** partition
- The 2<sup>nd</sup> row will be inserted into the **sales\_q2\_2k22** partition
- The 3<sup>rd</sup> row will be inserted into the **sales\_q3\_2k22** partition
- The 4<sup>th</sup> row will be inserted into the **sales\_q2\_2k23** partition
- The 5<sup>th</sup> row will be inserted (continue to be inserted the appropriate partition).

Further we can add more partition tables based on different ranges of values as needed, and PostgreSQL will automatically route data to the appropriate partition based on the partition key.

5. Partition table created and data inserted to the associated partition. Let us verify the data based on the partition.

First, determine the name of the partition you want to select data from. We use the **pg\_partition\_tree** function to get the list of partition names in the table:

```
# Verify the partitioned table  
SELECT  
    p.relname AS partition_name,  
    pg_get_expr(p.relpartbound, p.oid) AS partition_bound,  
    pg_get_partkeydef(p.oid) AS partition_key  
FROM  
    pg_partition_tree('sales') AS t  
JOIN pg_class AS p ON p.oid = t.relid;
```

Referring to [Figure 6.1](#), The above script execution will give a list of partition names and SQL definition of the partition, which includes the partition range that we can select from:

	partition_name name	partition_bound text	partition_key text
1	sales	[null]	RANGE (sale_date)
2	sales_q1_2k22	FOR VALUES FROM ('2022-01-01') TO ('2022-04-30')	[null]
3	sales_q2_2k22	FOR VALUES FROM ('2022-05-01') TO ('2022-08-31')	[null]
4	sales_q3_2k22	FOR VALUES FROM ('2022-09-01') TO ('2022-12-31')	[null]
5	sales_q1_2k23	FOR VALUES FROM ('2023-01-01') TO ('2023-04-30')	[null]
6	sales_q2_2k23	FOR VALUES FROM ('2023-05-01') TO ('2023-08-31')	[null]
7	sales_q3_2k23	FOR VALUES FROM ('2023-09-01') TO ('2023-12-31')	[null]

**Figure 6.1:** Partition list

6. Now by referring to above [Figure 6.1](#), we can use the partition name in your **SELECT** statement to select data from that specific partition.

# Select data from specific partition

```
SELECT * FROM sales_q2_2k22
WHERE sale_date BETWEEN '2022-04-01' AND '2022-08-31';
```

Please refer to the following figure:

	sale_id [PK] integer	sale_date [PK] date	sale_amount numeric (10,2)
1	2	2022-05-15	2000.00

**Figure 6.2:** Select data with specific partition

Note that the name of the partition table includes the partition name and the partition bounds, separated by underscores. In this example, the partition name is **sales** and the partition bounds are **q2\_2k22**, so the name of the partition table is **sales\_q2\_2k22**:

7. Let us try to insert data into a partition table with a range that has no partition defined in PostgreSQL database.
- Suppose we have a partitioned table named **sales**, partitioned by **sale\_date** column using the range method, and we try to insert a row with a **sale\_date** value outside the existing partition ranges. Please refer to the following figure:

```

3   INSERT INTO sales (sale_date, sale_amount) VALUES (DATE '2021-01-12', 1000.00);

Data Output Messages Notifications

ERROR: no partition of relation "sales" found for row
DETAIL: Partition key of the failing row contains (sale_date) = (2021-01-12).
SQL state: 23514

```

**Figure 6.3:** Insert to partition out of range

- b. This error message indicates that no partition of the sales table exists for the **sale\_date** value **2021-01-12**. To insert data for this range, we need to create a new partition with the appropriate range.
  - c. Since we have done much with the range partition, to create a list partitioned table, we need to follow the same steps as for a range partitioned table, with a few changes.
8. Create a table to act as the parent table that will contain the partitioned data. This is the table that will be partitioned by **LIST**. For this example, we will create a table called sellers.

*# create a List partitioned table*

```

CREATE TABLE seller (
    sale_id SERIAL,
    sale_date DATE NOT NULL,
    product_id TEXT NOT NULL,
    sale_amount NUMERIC(10, 2) NOT NULL,
    PRIMARY KEY (sale_id, product_id)
) PARTITION BY LIST (product_id);

```

In this example, we created a table **sale** with columns **sale\_id**, **sale\_date**, **product\_id**, and **sale\_amount**. We then specify that this table should be partitioned by list based on the **product\_id** column.

Next, we create three partitions of the seller table using the **CREATE TABLE PARTITION OF** syntax. These are the tables that will hold the actual data. We will create partition tables based on list of product id.

*# create a List partitioned table*

```

CREATE TABLE seller_software PARTITION OF seller FOR VALUES IN
('PG_DB', 'IBM_DB2', 'DB_MYSQL', 'RHEL');

```

```

CREATE TABLE seller_hardware PARTITION OF seller FOR VALUES IN

```

```
('PSERIES', 'iSERIES');

CREATE TABLE seller_general PARTITION OF seller DEFAULT;
```

The first partition, **seller\_software**, contains sales for products with **product\_id** values of **PG\_DB**, **IBM\_DB2**, **DB\_MYSQL** and **RHEL**. The second partition, **seller\_hardware**, contains sales for products with **product\_id** values of **PSERIES** and **iSERIES**. Finally, the third partition, **seller\_general**, is the default partition, which will contain all other sales that do not match any of the previous partitions.

- Now we will insert data into the list partitioned table. When we insert data, PostgreSQL will automatically route each row to the appropriate partition based on the partition key.

```
# Insert Data into partition table
```

```
INSERT INTO seller (sale_date, product_id, sale_amount) VALUES ('2023-01-01', 'PG_DB', 100.00);

INSERT INTO seller (sale_date, product_id, sale_amount) VALUES ('2023-01-02', 'iSERIES', 200.00);

INSERT INTO seller (sale_date, product_id, sale_amount) VALUES ('2023-01-03', 'Annual Maintenance', 400.00);
```

With the above script:

- The 1<sup>st</sup> row will be inserted into the **seller\_software** partition.
- The 2<sup>nd</sup> row will be inserted into the **seller\_hardware** partition.
- The 3<sup>rd</sup> row will be inserted into the **seller\_general** partition.

Further we can add more partition tables based on different list of values as needed, and PostgreSQL will automatically route data to the appropriate partition based on the partition key.

- Finally, partition table created and data inserted to the associated partition. Let us verify the data based on the partition.
  - First, determine the name of the partition you want to select data from. We use the **pg\_partition\_tree** function to get the list of partition names in the table.

```
# Verify the partitioned table
```

```
SELECT
```

```

    p.relname AS partition_name,
    pg_get_expr(p.relpertbound, p.oid) AS partition_bound,
    pg_get_partkeydef(p.oid) AS partition_key
  FROM
    pg_partition_tree('seller') AS t
  JOIN pg_class AS p ON p.oid = t.relid;

```

Referring to [Figure 6.3](#), the above script execution will give a list of partition names and SQL definition of the partition, which includes the partition range that we can select from:

	partition_name name	partition_bound text	partition_key text
1	seller	[null]	LIST (product_id)
2	seller_software	FOR VALUES IN ('PG_DB', 'IBM_DB2', 'DB MYSQL', 'RHEL')	[null]
3	seller_hardware	FOR VALUES IN ('PSERIES', 'ISERIES')	[null]
4	seller_general	DEFAULT	[null]

**Figure 6.4:** List partition

- b. Finally by referring to above [Figure 6.3](#), we can use the partition name in your **SELECT** statement to select data from that specific partition.

```

# Select data from seller_software partition
SELECT * FROM seller_software
WHERE sale_date BETWEEN '2023-01-01' AND '2023-03-01';

```

Please refer to the following figure:

	sale_id [PK] integer	sale_date date	product_id [PK] text	sale_amount numeric (10,2)
1	1	2023-01-01	PG_DB	100.00

**Figure 6.5:** Select data with specific partition

```

# Select data from seller_general partition
SELECT * FROM seller_general
WHERE sale_date BETWEEN '2022-01-01' AND '2023-03-01';

```

Please refer to the following figure:

	sale_id [PK] integer	sale_date date	product_id [PK] text	sale_amount numeric (10,2)
1	4	2023-01-03	Annual Maintenance	400.00

**Figure 6.6:** Select data with seller\_general partition

Now that is it for this recipe! We now have a range and list partitioned table in PostgreSQL 15 database. We can continue to add more partition tables based on different ranges/list of values as needed, and PostgreSQL will automatically route data to the appropriate partition based on the partition key. Let us jump on to the next recipe to further get and insight to partitioning in PostgreSQL 15.

## Recipe 41: Partition management with PostgreSQL 15

Partition management in PostgreSQL 15 is the process of attaching and detaching partitions in PostgreSQL 15, it is a way to manage and manipulate partitions for a partitioned table. Attaching a partition is the process of adding a new partition to an already partitioned table, while detaching a partition is the process of removing a partition from a partitioned table.

To attach a partition to a partitioned table, first, a new partition must be created with the same schema as the other partitions in the table. Then, the partition can be attached to the table using the **ALTER TABLE** command with the **ATTACH PARTITION** subcommand, specifying the partition name or partition constraint.

Here are the steps to attach a partition in PostgreSQL partitioned table:

1. First, create the new regular table with the desired schema and partition constraints, and then use the **ALTER TABLE** command with the **ATTACH PARTITION** option to attach it to the partitioned table.

Suppose we have a partitioned table named **sales**, partitioned by **sale\_date** column using the range method that we created in our previous recipe *Setup and exploit partitioning in PostgreSQL*. We use the sales table to attach FY 2021 data that exist in the regular table. Please refer to the following figure:

	partition_name	partition_bound
1	sales	[null]
2	sales_q1_2k21	FOR VALUES FROM ('2022-01-01') TO ('2022-04-30')
3	sales_q2_2k21	FOR VALUES FROM ('2022-05-01') TO ('2022-08-31')

**Figure 6.7:** List available partition

2. Next, we create three new regular table. We use the LIKE clause to create a regular table that has the same structure (columns, constraints, indexes, and so on) as an existing table (sales table).

```
# Create regular table for FY 2021
```

```
create table sales_q1_2k21 ( like sales including all );
create table sales_q2_2k21 ( like sales including all );
create table sales_q3_2k21 ( like sales including all );
```

3. Since the regular table is ready to be attached to the partition sales. To attach a partition to a partitioned table, use the **ALTER TABLE** command with the **ATTACH PARTITION** option, followed by the name of the table to attach.

```
# Attach regular table to a partitioned table
```

```
ALTER TABLE SALES ATTACH PARTITION sales_q1_2k21 FOR VALUES
FROM ('2021-01-01') TO ('2021-04-30');
ALTER TABLE SALES ATTACH PARTITION sales_q2_2k21 FOR VALUES
FROM ('2021-05-01') TO ('2021-08-31');
ALTER TABLE SALES ATTACH PARTITION sales_q3_2k21 FOR VALUES
FROM ('2021-09-01') TO ('2021-12-31');
```

Let us verify the newly attached partition using the following script.

```
# Verify the partitioned table
```

```
SELECT
    p.relname AS partition_name,
    pg_get_expr(p.relpartbound, p.oid) AS partition_bound,
    pg_get_partkeydef(p.oid) AS partition_key
FROM
    pg_partition_tree('sales') AS t
JOIN pg_class AS p ON p.oid = t.relid;
```

Referring to [Figure 6.7](#), The above script execution will give a list of partition names and SQL definition of the partition, which includes the newly attached that was attached in the previous step:

	partition_name	partition_bound
1	sales	[null]
2	sales_q1_2k22	FOR VALUES FROM ('2022-01-01') TO ('2022-04-30')
3	sales_q2_2k22	FOR VALUES FROM ('2022-05-01') TO ('2022-08-31')
4	sales_q3_2k22	FOR VALUES FROM ('2022-09-01') TO ('2022-12-31')
5	sales_q1_2k23	FOR VALUES FROM ('2023-01-01') TO ('2023-04-30')
6	sales_q2_2k23	FOR VALUES FROM ('2023-05-01') TO ('2023-08-31')

**Figure 6.8:** Partition post attached

To detach a partition from a partitioned table, first, any data in the partition must be moved or removed, and any foreign key constraints referencing the partition must be dropped. Then, the partition can be detached from the table using the **ALTER TABLE** command with the **DETACH PARTITION** subcommand, specifying the partition name or partition constraint.

Here are the steps to detach a partition in PostgreSQL partitioned table:

1. To detach a partition from a partitioned table, use the **ALTER TABLE** command with the **DETACH PARTITION** option, followed by the name of the partition you want to detach.
  - Suppose from partitioned table named **sales**, FY 2022 data partition no longer required to the business and the decision is to archive the data to the regular table.
  - To detach a partition from a partitioned table, use the **ALTER TABLE** command with the **DETACH PARTITION** option, followed by the name of the partition to detach.

```
# Detach partition from partitioned table
```

```
ALTER TABLE sales DETACH PARTITION sales_q1_2k22;
```

```
ALTER TABLE sales DETACH PARTITION sales_q2_2k22;
```

```
ALTER TABLE sales DETACH PARTITION sales_q3_2k22;
```

2. Let us verify the partition post detach of the partition using the following script.

```
# Verify the partitioned table
```

```
SELECT
```

```
p.relname AS partition_name,
```

```
pg_get_expr(p.relpartbound, p.oid) AS partition_bound,
```

```
pg_get_partkeydef(p.oid) AS partition_key
```

```
FROM
```

```

pg_partition_tree('sales') AS t
JOIN pg_class AS p ON p.oid = t.relid;

```

Referring to [Figure 6.8](#), The above script execution will give a list of partition names and SQL definition of the partition:

	partition_name name	partition_bound text	partition_key text
1	sales	[null]	RANGE (sale_date)
2	sales_q1_2k23	FOR VALUES FROM ('2023-01-01') TO ('2023-04-30')	[null]
3	sales_q2_2k23	FOR VALUES FROM ('2023-05-01') TO ('2023-08-31')	[null]
4	sales_q3_2k23	FOR VALUES FROM ('2023-09-01') TO ('2023-12-31')	[null]
5	sales_q1_2k21	FOR VALUES FROM ('2021-01-01') TO ('2021-04-30')	[null]
6	sales_q2_2k21	FOR VALUES FROM ('2021-05-01') TO ('2021-08-31')	[null]
7	sales_q3_2k21	FOR VALUES FROM ('2021-09-01') TO ('2021-12-31')	[null]

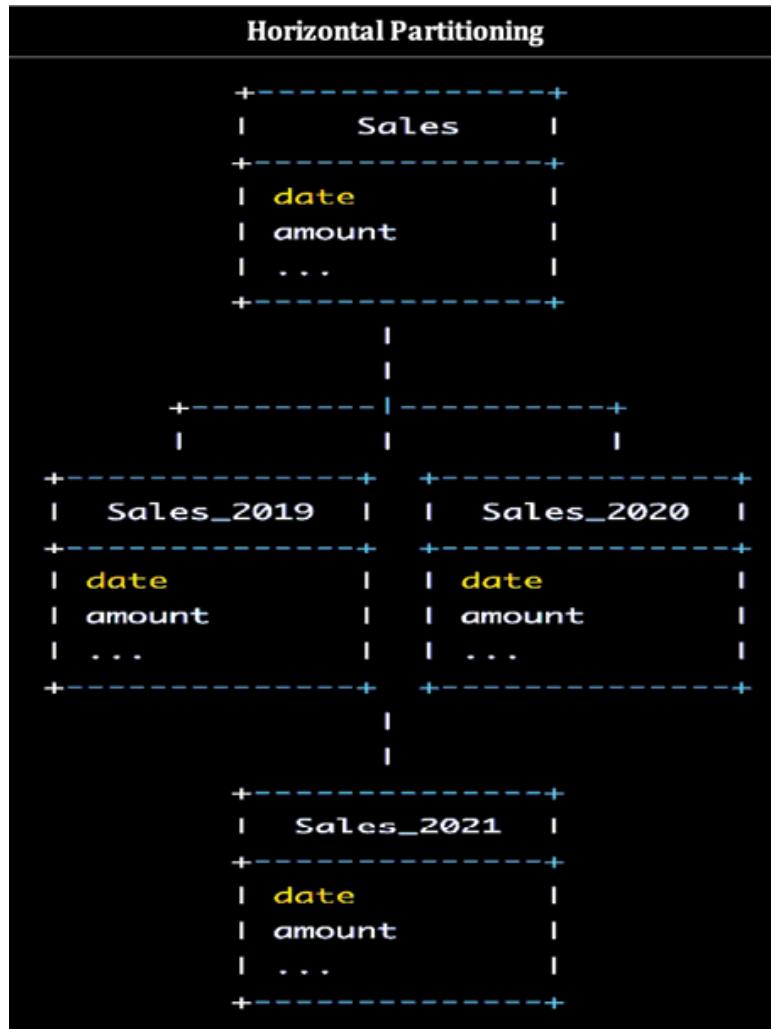
**Figure 6.9: Partition post detach**

## Recipe 42: Getting insight to vertical and horizontal partitioning

In PostgreSQL 15, understanding the concepts of vertical and horizontal partitioning is crucial for optimizing database performance and managing large datasets effectively. Vertical partitioning involves splitting a table into multiple tables, each containing a subset of columns. This strategy can reduce I/O operations, especially when some columns are rarely accessed. On the other hand, horizontal partitioning divides a table's data into smaller, more manageable chunks, called partitions, based on a specific column or condition. By doing so, horizontal partitioning improves query performance and simplifies data maintenance.

### Horizontal partitioning

Determine the partitioning key: In horizontal partitioning, the partitioning key is used to divide the data horizontally into separate tables. The partitioning key can be any column or combination of columns that represent a logical division of data. For example, we have a table of sales data, we partition the data by **sale\_date**, with each partition containing all the sales data for a particular year. Please refer to the following figure:



*Figure 6.10: Horizontal partitioning*

1. **Create a parent table by defining the partitioning key:**  
Create a table to act as the parent table that will contain the partitioned data. This is the table that will be partitioned by **RANGE**. For this example, we will create a table called **sales**.

```
# create a range partitioned table
```

```
CREATE TABLE sales (
    sale_id SERIAL,
    sale_date DATE NOT NULL,
    sale_amount NUMERIC(10,2) NOT NULL,
    PRIMARY KEY (sale_id, sale_date)
) PARTITION BY RANGE (sale_date);
```

In this example, the **PARTITION BY RANGE** clause specifies that the table will be partitioned based on the **sale\_date** column.

2. **Create the partition:** Create the partition using the partitioning key values. In PostgreSQL, we use the **CREATE TABLE** statement to create a table, and the **PARTITION OF** clause to specify the partition of the defined table.

```
# create partition for sales table
```

```
CREATE TABLE sales_2k19 PARTITION OF sales FOR VALUES FROM ('2019-01-01') TO ('2019-12-31');
```

This creates a new table called **sales\_2k19** that is a partition of the existing **sales** table. The partition includes all rows where the partitioning key is between **2019-01-01** and **2019-12-**.

3. **Create additional partitions:** Repeat the previous step to create additional partitions for each year of data.

```
# create partition for sales table
```

```
CREATE TABLE sales_2k20 PARTITION OF sales FOR VALUES FROM ('2020-01-01') TO ('2020-12-31');
```

```
CREATE TABLE sales_2k21 PARTITION OF sales FOR VALUES FROM ('2021-01-01') TO ('2021-12-31');
```

```
CREATE TABLE sales_2k22 PARTITION OF sales FOR VALUES FROM ('2022-01-01') TO ('2022-12-31');
```

This creates three more partitions, one for 2020 data, one for 2021 data and one for 2022 data.

4. **Insert data into partitions:** Insert data into the appropriate partition using the partitioning key.

```
# Insert Data into partition table
```

```
INSERT INTO sales (sale_date, sale_amount) VALUES  
('2019-01-12', 1000.00),  
('2019-05-15', 2000.00),  
('2020-01-12', 1000.00),  
('2020-05-15', 1000.00),  
('2021-01-12', 2000.00),  
('2021-05-15', 1000.00);
```

This inserts a new row into the **sales\_2k19**, **sales\_2k20** and **sales\_2k21** table with the date and amount values specified.

5. **Query partitions:** Query the partitions as if they were regular tables. PostgreSQL will automatically route queries to the appropriate partition based on the partitioning key.

```
# Verify the sales table
```

```
SELECT * FROM sales WHERE sale_date BETWEEN '2019-01-01' AND '2020-
```

12-31';

This selects all sales data from 2019, 2020, and 2021 because the query is executed against all three partitions.

## **Vertical partitioning**

Determine the partitioning columns: In vertical partitioning, the partitioning columns are used to divide the data vertically into separate tables. The partitioning columns can be any columns that are frequently accessed together or that are large or infrequently accessed. For example, if you have a table of customer data, you might partition the data into a basic table with frequently accessed columns like name and email, and an extended table with less frequently accessed columns like address and phone number. Please refer to the following figure:

Vertical Partitioning		
<pre>+-----+        Customer        +-----+   id       name     email     address     phone     ...    +-----+</pre>		
<pre>                        +-----+-----+                 +-----+-----+   Customer_Basic      Customer_Extended   +-----+-----+   id            id           name          address     email         phone     +-----+-----+</pre>		

*Figure 6.11: Vertical partitioning*

### Recipe 43: Working with automatic partition with PostgreSQL

Automatic partitioning in PostgreSQL 15 refers to the ability to automatically create and manage partitions in a partitioned table without the need for manual intervention.

Automatic partitioning can be achieved using various methods like **pg\_partman** extension, custom scripts with a scheduler, or triggers. The goal of automatic partitioning is to reduce manual effort by efficiently managing the partitioned data. With automatic partitioning, new partitions can be created and managed automatically based on specific criteria, such as time intervals or data size, without the need for manual intervention.

## Manage partition using pg\_partman extension

**pg\_partman** is a PostgreSQL extension that provides an easy and efficient way to manage partitioned tables in PostgreSQL. It simplifies the creation and management of partitions by automating many of the tasks involved, such as creating partitions, removing partitions, and rotating partitions. Here are the recipe steps to use **pg\_partman** in PostgreSQL.

Let us work with an example of food delivery service to start with recipe with the following steps:

1. The first step to this recipe is to install the **pg\_partman** extension in our PostgreSQL database.

```
# Install pg_partman extension
```

```
CREATE EXTENSION pg_partman SCHEMA partman;
```

Here, **CREATE EXTENSION** is a PostgreSQL command that allows users to install extensions to add functionality to the database system. In this case, the **pg\_partman** extension is being installed.

**SCHEMA partman** specifies the schema where the extension should be installed. In this case, the extension is being installed in the partman schema.

2. Create a table to act as the parent table that will contain the partitioned data and the table that to be partitioned. For this recipe example, we will create a table called **orders**.

```
# create a range partitioned table
```

```
CREATE TABLE orders (
    id SERIAL,
    restaurant_id INT NOT NULL,
    customer_id INT NOT NULL,
    order_date DATE NOT NULL,
    order_total NUMERIC(10, 2) NOT NULL,
```

```

PRIMARY KEY (id, order_date)
) PARTITION BY RANGE (order_date);

```

3. Now we are ready to create a partition using **create\_parent** function on the order table of our food delivery service.

```

# create partition for order table
select partman.create_parent('public.orders','order_date','native','monthly');
# Alternate method to create partition for order table
select partman.create_parent('public.orders','order_date','native',
p_interval=> 'monthly');

```

**partman.create\_parent** is a function provided by the **pg\_partman** extension in PostgreSQL that creates a partition for order table with a monthly partition based on the **order\_date** column using following argument values:

- **public.orders**: Parent partition table
- **order\_date**: Parent table column name that belongs to the partitioning key
- **native**: Allows to use native partitioning methods
- **monthly**: Defines the monthly interval for each partition

Apart from the above arguments that we used on our recipe example, there a lot more arguments available that provides more granular level functions for table partitioning as shown in the [Table 6.1](#):

Argument	Description
<b>p_parent_table</b>	Name of the parent table
<b>p_control</b>	Name of the column used as the partition key
<b>p_type</b>	Type of partitioning to be used ('native', 'partman')
<b>p_interval</b>	The interval at which partitions should be created ('yearly', 'quarterly', 'monthly', and so on)
<b>p_constraint_cols</b>	Whether to create a constraint on the partitioned table
<b>p_inherit_fk</b>	Whether to automatically inherit foreign keys from the parent table to the partitioned tables

**Table 6.1:** *create\_parent argument in pg\_partman*

4. Finally, a partition is created for order table. Let us verify the partition that we have created in the previous step of this recipe:

```
# Verify the partitioned table
```

```
SELECT
    p.relname AS partition_name,
    pg_get_expr(p.relpartbound, p.oid) AS partition_bound,
    pg_get_partkeydef(p.oid) AS partition_key
FROM
    pg_partition_tree('orders') AS t
JOIN pg_class AS p ON p.oid = t.relid;
```

Referring to [Figure 6.11](#), The above script execution will give a list of partition names and SQL definition of the partition, which includes the partition range for each partition:

	partition_name name	partition_bound text	partition_key text
1	orders	[null]	RANGE (order_date)
2	orders_p2022_11	FOR VALUES FROM ('2022-11-01') TO ('2022-12-01')	[null]
3	orders_p2022_12	FOR VALUES FROM ('2022-12-01') TO ('2023-01-01')	[null]
4	orders_p2023_01	FOR VALUES FROM ('2023-01-01') TO ('2023-02-01')	[null]
5	orders_p2023_02	FOR VALUES FROM ('2023-02-01') TO ('2023-03-01')	[null]
6	orders_p2023_03	FOR VALUES FROM ('2023-03-01') TO ('2023-04-01')	[null]
7	orders_p2023_04	FOR VALUES FROM ('2023-04-01') TO ('2023-05-01')	[null]
8	orders_p2023_05	FOR VALUES FROM ('2023-05-01') TO ('2023-06-01')	[null]
9	orders_p2023_06	FOR VALUES FROM ('2023-06-01') TO ('2023-07-01')	[null]
10	orders_p2023_07	FOR VALUES FROM ('2023-07-01') TO ('2023-08-01')	[null]
11	orders_default	DEFAULT	[null]

**Figure 6.12:** Automatic partition with pg\_partman

By default, 10 partition created by the **partman.create\_parent** function without passing any argument like **premake**. So let identify the partition configuration in the next step.

5. Execute the following command to determine the **pg\_partman** partition configuration:

```
# verify partition for order table
```

```
select
```

```
parent_table, control, partition_type,
```

```
partition_interval, premake, retention,
```

```
retention_keep_table
```

```
from partman.part_config
```

```
where parent_table='public.orders';
```

6. The above SQL command selects specific columns from the **partman.part\_config** table, which is used to store the partition configuration information for tables managed by **pg\_partman**. Please refer to the following figure:

	parent_table [PK] text	control text	partition_type text	partition_interval text	premake integer	retention text	retention_keep_table boolean
1	public.orders	order_date	native	1 mon	4	[null]	true

**Figure 6.13:** part\_config detail

Following a breakdown of the selected columns output as shown in [Figure 6.12](#):

Column	Description	Existing configuration
<b>parent_table</b>	The name of the parent table that is being managed by <b>pg_partman</b> .	<b>Public.orders</b>
<b>control</b>	The partition control method used by <b>pg_partman</b> for the parent table.	<b>order_date</b>
<b>partition_type</b>	The type of partitioning used for the parent table.	<b>Native</b>
<b>partition_interval</b>	The interval at which new partitions are created.	<b>1 mon</b>
<b>premake</b>	The number of partitions that are created ahead of time. This can help reduce the overhead of creating new partitions on-the-fly.	<b>4</b>
<b>retention</b>	The length of time (in seconds) that partitions should be kept before they are dropped. If set to none, partitions will never be dropped automatically.	-
<b>retention_keep_table</b>	The Boolean value that define where partitions can be moved to instead of being dropped. This can be useful for archiving data.	<b>T (true)</b>

**Table 6.2:** part\_config description

7. Since the configuration of **premake** is **4**, that means only 4 partitions are created ahead of time. But the business decided to have **15** partitions to be created ahead. So, in that case we need to update the configuration of the orders partition table using the following SQL command.

```
# Amend partition configuration  
update partman.part_config SET premake = 15 WHERE  
parent_table='public.orders';
```

Where:

- **premake** defines the new number of future partitions to create for parent table.
- **parent\_table** defines the name of the parent table for which you want to update the number of future partitions created in advance.

8. At this stage, we have update the partition configuration for table **public.orders**, the new partition only get created after executing the maintenance on partition table. The main idea behind this recipe is to create a partition automatically based on the partition table configuration. To achieve this, **pg\_cron** scheduler is needed that invoke the following maintenance functions.

```
# Execute partition maintenance  
CALL partman.run_maintenance_proc();
```

9. Let us schedule the maintenance query with the **pg\_cron** using the following SQL script.

```
# Schedule partition maintenance  
SELECT cron.schedule('0 9 * * *', $$ CALL  
partman.run_maintenance_proc()$$);
```

10. The above query schedules a cron job to run at 9 AM every day. The job will execute the **partman.run\_maintenance** function for the **public.orders** table. This means that the maintenance process will be executed daily at 9 AM for the **public.orders** table, and the progress of the job will be monitored by executing the following SQL command.

```
# verify the partition maintenance job  
select * from cron.job_run_details where jobid = 5 order by start_time desc  
limit 10;
```

11. Referring to [Figure 6.13](#), the above SQL query will display the details of the last **10** runs of the **pg\_cron** job with ID **5**. The result will be ordered by the start time of the job, with the most recent job at the top of the list. Please refer to the following figure:

JobId	RnId	Job_Pid	Database	Username	Command	Status	Return_Message	Start_Time	End_Time
1	5	21	6934	postgres	CALL partman.run_maintenance_proc()	succeeded	CALL	2023-03-06 09:00:00.03674+08	2023-03-06 09:01:00.02854+08
2	5	20	2831	postgres	CALL partman.run_maintenance_proc()	succeeded	CALL	2023-03-05 09:00:00.0102+08	2023-03-05 09:01:00.003894+08

**Figure 6.14:** partman job log

For each job run, the output will include details such as the start time and end time of the job, the exit code, and any error messages. This information can be useful for troubleshooting and monitoring the performance of scheduled jobs.

12. To clarify, **partman.run\_maintenance** is a function provided by the **pg\_partman** extension that performs maintenance operations on partitioned tables. Scheduling this function to run on a regular basis with **pg\_cron** ensures that partition maintenance tasks are performed automatically and in a timely manner.

Post successful execution of the maintenance script via **pg\_cron**, the new partition added to the orders tables. So, let us verify the newly added partition by executing following SQL script.

# Verify the partitioned table

```

SELECT
    p.relname AS partition_name,
    pg_get_expr(p.relpartbound, p.oid) AS partition_bound,
    pg_get_partkeydef(p.oid) AS partition_key
FROM
    pg_partition_tree('orders') AS t
JOIN pg_class AS p ON p.oid = t.relid;
```

Referring to [Figure 6.14](#), the above script execution will give a list of partition names and SQL definition of the partition with 15 newly added partition. The update to the premake to value 15, added 15 new partition post successful execution of the maintenance script via **pg\_cron**. Please refer to the following figure:

	partition_name name	partition_bound text	partition_key text
1	orders	[null]	RANGE (order_date)
2	orders_p2022_11	FOR VALUES FROM ('2022-11-01') TO ('2022-12-01')	[null]
3	orders_p2022_12	FOR VALUES FROM ('2022-12-01') TO ('2023-01-01')	[null]
4	orders_p2023_01	FOR VALUES FROM ('2023-01-01') TO ('2023-02-01')	[null]
5	orders_p2023_02	FOR VALUES FROM ('2023-02-01') TO ('2023-03-01')	[null]
6	orders_p2023_03	FOR VALUES FROM ('2023-03-01') TO ('2023-04-01')	[null]
7	orders_p2023_04	FOR VALUES FROM ('2023-04-01') TO ('2023-05-01')	[null]
8	orders_p2023_05	FOR VALUES FROM ('2023-05-01') TO ('2023-06-01')	[null]
9	orders_p2023_06	FOR VALUES FROM ('2023-06-01') TO ('2023-07-01')	[null]
10	orders_p2023_07	FOR VALUES FROM ('2023-07-01') TO ('2023-08-01')	[null]
11	orders_p2023_08	FOR VALUES FROM ('2023-08-01') TO ('2023-09-01')	[null]
12	orders_p2023_09	FOR VALUES FROM ('2023-09-01') TO ('2023-10-01')	[null]
13	orders_p2023_10	FOR VALUES FROM ('2023-10-01') TO ('2023-11-01')	[null]
14	orders_p2023_11	FOR VALUES FROM ('2023-11-01') TO ('2023-12-01')	[null]
15	orders_p2023_12	FOR VALUES FROM ('2023-12-01') TO ('2024-01-01')	[null]
16	orders_p2024_01	FOR VALUES FROM ('2024-01-01') TO ('2024-02-01')	[null]
17	orders_p2024_02	FOR VALUES FROM ('2024-02-01') TO ('2024-03-01')	[null]
18	orders_p2024_03	FOR VALUES FROM ('2024-03-01') TO ('2024-04-01')	[null]
19	orders_p2024_04	FOR VALUES FROM ('2024-04-01') TO ('2024-05-01')	[null]
20	orders_p2024_05	FOR VALUES FROM ('2024-05-01') TO ('2024-06-01')	[null]
21	orders_p2024_06	FOR VALUES FROM ('2024-06-01') TO ('2024-07-01')	[null]

**Figure 6.15:** Partition created with partman

## Recipe 44: Manage partition using custom scripts with a scheduler

In this recipe, we will learn how to manage partition using custom scripts with a scheduler for PostgreSQL 15. This recipe assumes that PostgreSQL 15 already installed and have administrative privileges:

1. The first step to this recipe is to create a partition management shell script. We have written the custom scripts that perform the partition management tasks. For example, create new partitions, drop old partitions, or move data between partitions.

```
#!/bin/bash
# Set environment variables for PostgreSQL connection
export PGHOST=localhost
export PGPORT=5432
export PGUSER=postgres
export PGPASSWORD=postgres
export PGDATABASE=postgres
# Define partitioning parameters
parent_table_name="orders"
```

```

partition_field_name="order_date"
partition_interval="monthly"
start_date=$(date +%Y-%m-%d)
end_date="$(date +%Y-%m-%d -d 'today + 20 months')"
# Create a partition for table orders
current_date="${start_date}"
while [[ "${current_date}" < "${end_date}" ]]; do
    partition_name="${parent_table_name}_p$(date +%Y_%m -d
    "${current_date}")"
    partition_bound="${current_date}::date"
    next_month="$(date +%Y-%m -d "${current_date} + 1 month")"
    psql -c "CREATE TABLE ${partition_name} PARTITION OF
${parent_table_name} FOR VALUES FROM (${partition_bound}) TO
('${next_month}-01'::date - interval '1 day');" -q
    current_date="${next_month}-01"
done

```

The above shell script creates a series of PostgreSQL partitions for a parent table called **orders**. The script uses environment variables to define the connection parameters for the PostgreSQL database, such as the host, port, username, password, and database name.

The script also defines partitioning parameters for the parent table, including the name of the partition key field, the partition interval (in this case, monthly), the start date for the partitioning scheme, and the end date (which is calculated as the current date plus 20 months).

The main logic of the script uses a while loop to create a partition table for each month between the start date and the end date. The loop starts by setting the current date to the start date and continues while the current date is less than the end date.

Inside the loop, the script calculates the name of the partition table based on the current date, using the format `orders_pYYYY_MM`. It also calculates the partition bound for the current month, which is the first day of the month in the format **YYYY-MM-DD**. It then calculates the next month using the same format.

2. Navigate to the directory where you saved the **partition\_management.sh** script and grant execute permissions to the script.

```
# Grant Execute permission to the script
```

```
chmod +x partition_management.sh
```

### 3. Schedule the partition management script in the **crontab**.

```
# Grant Execute permission to the script
```

```
0 0 1,7,13,19,25 * * /script/partition_management.sh >>
```

```
/script/log/partition.log 2>&1
```

In the above example, we are running the **partition\_management.sh** script executed at midnight (0 0 \* \* \*) on the 1<sup>st</sup>, 7<sup>th</sup>, 13<sup>th</sup>, 19<sup>th</sup>, and 25<sup>th</sup> day of every month, and redirecting the output to a log file named **partition.log**.

### 4. Monitor the PostgreSQL database to make sure the partition management commands are executing correctly.

Referring to the execution output of following script from the [Figure 6.15](#), creates a partition of the parent table for the specified partition bound (from the partition bound to the first day of the next month minus one day).

```
# Verify the partitioned table
```

```
SELECT
```

```
p.relname AS partition_name,
```

```
pg_get_expr(p.relpartbound, p.oid) AS partition_bound,
```

```
pg_get_partkeydef(p.oid) AS partition_key
```

```
FROM
```

```
pg_partition_tree('orders') AS t
```

```
JOIN pg_class AS p ON p.oid = t.relid;
```

Referring to [Figure 6.15](#), The above script execution will give a list of partition names and SQL definition of the partition with +/-20 newly added partition which is calculated from the current date. The shell script creates partition based on the specified partition bound which is calculated on the basis of **FROM (\${partition\_bound}) TO ('\${next\_month}-01'::date - interval '1 day')** of the shell script. Please refer to the following figure:

	partition_name name	partition_bound text	partition_key text
1	orders	[null]	RANGE (order_date)
2	orders_p2022_01	FOR VALUES FROM ('2022-01-01') TO ('2022-01-31')	[null]
3	orders_p2023_03	FOR VALUES FROM ('2023-03-06') TO ('2023-03-31')	[null]
4	orders_p2023_04	FOR VALUES FROM ('2023-04-01') TO ('2023-04-30')	[null]
5	orders_p2023_05	FOR VALUES FROM ('2023-05-01') TO ('2023-05-31')	[null]
6	orders_p2023_06	FOR VALUES FROM ('2023-06-01') TO ('2023-06-30')	[null]
7	orders_p2023_07	FOR VALUES FROM ('2023-07-01') TO ('2023-07-31')	[null]
8	orders_p2023_08	FOR VALUES FROM ('2023-08-01') TO ('2023-08-31')	[null]
9	orders_p2023_09	FOR VALUES FROM ('2023-09-01') TO ('2023-09-30')	[null]
10	orders_p2023_10	FOR VALUES FROM ('2023-10-01') TO ('2023-10-31')	[null]
11	orders_p2023_11	FOR VALUES FROM ('2023-11-01') TO ('2023-11-30')	[null]
12	orders_p2023_12	FOR VALUES FROM ('2023-12-01') TO ('2023-12-31')	[null]
13	orders_p2024_05	FOR VALUES FROM ('2024-05-01') TO ('2024-05-31')	[null]
14	orders_p2024_06	FOR VALUES FROM ('2024-06-01') TO ('2024-06-30')	[null]
15	orders_p2024_07	FOR VALUES FROM ('2024-07-01') TO ('2024-07-31')	[null]
16	orders_p2024_08	FOR VALUES FROM ('2024-08-01') TO ('2024-08-31')	[null]
17	orders_p2024_09	FOR VALUES FROM ('2024-09-01') TO ('2024-09-30')	[null]
18	orders_p2024_10	FOR VALUES FROM ('2024-10-01') TO ('2024-10-31')	[null]
19	orders_p2024_11	FOR VALUES FROM ('2024-11-01') TO ('2024-11-30')	[null]
20	orders_p2024_01	FOR VALUES FROM ('2024-01-01') TO ('2024-01-31')	[null]
21	orders_p2024_02	FOR VALUES FROM ('2024-02-01') TO ('2024-02-29')	[null]
22	orders_p2024_03	FOR VALUES FROM ('2024-03-01') TO ('2024-03-31')	[null]
23	orders_p2024_04	FOR VALUES FROM ('2024-04-01') TO ('2024-04-30')	[null]

**Figure 6.16:** partition created with pg\_cron

In summary, the script creates partitions for a PostgreSQL table named **orders** based on a specified partitioning parameter, which is the **order\_date** column, using the Bash shell and the PostgreSQL command-line utility **psql**. It creates a loop that iterates over the partitioning interval, creates a partition for each month until the end date is reached.

## Recipe 45: Getting insight to partition pruning

Partition pruning is a feature in PostgreSQL that allows the query optimizer to eliminate partitions that are not needed for a particular query, which can significantly improve query performance on partitioned tables.

To better understand partition pruning in PostgreSQL 15, let us first review some basic concepts. A partitioned table in PostgreSQL is a table that is divided into smaller sub-tables called partitions, each with its own distinct set of data. Partitioning can be done by range, list, or hash. When a query is executed against a partitioned table, the query planner first determines which partitions need to be scanned to produce the results. Partition pruning is the process of eliminating unnecessary partitions from this list, reducing the amount of data that needs to be scanned.

PostgreSQL 15's improved partition pruning allows for more accurate and efficient query execution. This feature allows for better optimization of queries by reducing the number of partitions that need to be accessed.

Partition pruning in PostgreSQL works by analyzing the **WHERE** clause of a query to determine which partitions need to be scanned. It then uses the partition bounds of each partition to determine which partitions can be eliminated from the list of partitions that need to be scanned.

Here is a recipe on how partition pruning can work on a table by **UBER** data in as an example set:

1. First, we create a table named trips that stores UBER ride data, partitioned by date using the **trip\_date** column.

```
# Create a partition table
```

```
CREATE TABLE trips (
    trip_id int,
    rider_id int,
    driver_id int,
    trip_date date,
    pickup_location text,
    dropoff_location text,
    fare numeric
) PARTITION BY RANGE (trip_date);
```

2. We then create partitions based on date ranges, for this recipe example set.

```
# Create a partition table
```

```
CREATE TABLE trips_2020_01_01 PARTITION OF trips FOR VALUES FROM
('2020-01-01') TO ('2020-02-01');
CREATE TABLE trips_2020_02_01 PARTITION OF trips FOR VALUES FROM
('2020-02-01') TO ('2020-03-01');
```

```
CREATE TABLE trips_2020_03_01 PARTITION OF trips FOR VALUES FROM ('2020-03-01') TO ('2020-04-01');
```

# Create an index on the trip\_date column

```
CREATE INDEX trips_2020_trip_date_idx ON trips (trip_date);
```

3. Now, let us assume we want to run a query to get the total fare for all trips that occurred on January 1, 2020. With partition pruning on, the **EXPLAIN** statement shows that only the relevant **trips\_2020** partition needs to be scanned:

Execute the Select statement to list the fare of the trip

```
SELECT SUM(fare) FROM trips WHERE trip_date = '2020-01-01';
```

Execute EXPLAIN

```
EXPLAIN SELECT SUM(fare) FROM trips WHERE trip_date = '2020-01-01';
```

QUERY PLAN

```
-----  
Aggregate (cost=0.00..0.01 rows=1 width=8)
```

```
-> Index Scan using trips_2020_trip_date_idx on trips_2020 (cost=0.00..0.00  
rows=1 width=12)
```

```
Index Cond: (trip_date = '2020-01-01'::date)
```

Execute EXPLAIN ANALYZE

```
EXPLAIN ANALYZE SELECT SUM(fare) FROM trips WHERE trip_date =  
'2020-01-01';
```

QUERY PLAN

```
-----  
Aggregate (cost=8.35..8.36 rows=1 width=8) (actual time=0.012..0.012  
rows=1 loops=1)
```

```
-> Index Scan using trips_2020_trip_date_idx on trips_2020 (cost=0.00..8.33  
rows=1 width=12) (actual time=0.008..0.008 rows=1 loops=1)
```

```
Index Cond: (trip_date = '2020-01-01'::date)
```

Planning Time: 0.127 ms

Execution Time: 0.034 ms

(The query plan shows that an index scan is used on the **trips\_2020** partition, and only one row needs to be fetched to get the total fare. The actual execution time is only **0.034 ms**, which is extremely fast).

The query plan shows that an index scan is used on the **trips\_2020** partition, and only one row needs to be fetched to get

the total fare. This is a very efficient query plan and should execute quickly.

When this query is executed, PostgreSQL query planner will use partition pruning to eliminate all other partitions that do not contain data for January 1, 2020. It can do this by examining the partition bounds specified when the partitions were created and identifying which partitions fall within the specified date range.

As a result, only the **trips\_2020\_01\_01** partition needs to be scanned, and all other partitions are skipped. This significantly reduces the amount of data that needs to be processed and can improve query performance.

- Now let us turn partition pruning off and run the same query with **EXPLAIN**.

#### *Execute EXPLAIN with Partition Pruning Disable*

```
SET enable_partition_pruning = off;  
EXPLAIN SELECT SUM(fare) FROM trips WHERE trip_date = '2020-01-01';  
          QUERY PLAN
```

```
-----  
Finalize Aggregate (cost=124090.27..124090.28 rows=1 width=8)  
 -> Gather (cost=124089.91..124090.26 rows=4 width=8)  
       Workers Planned: 4  
         -> Partial Aggregate (cost=123089.91..123089.92 rows=1 width=8)  
           -> Parallel Seq Scan on trips (cost=0.00..123073.00 rows=6751  
width=8)  
                 Filter: (trip_date = '2020-01-01'::date)
```

#### *Execute EXPLAIN ANALYZE with Partition Pruning DISABLE*

```
SET enable_partition_pruning = off;  
EXPLAIN ANALYZE SELECT SUM(fare) FROM trips WHERE trip_date =  
'2020-01-01';  
          QUERY PLAN
```

```
-----  
Finalize Aggregate (cost=985931.02..985931.03 rows=1 width=8) (actual  
time=3501.618..3501.619 rows=1 loops=1)  
 -> Gather (cost=985930.66..985931.01 rows=4 width=8) (actual  
time=3501.517..3501.518 rows=4 loops=1)  
       Workers Planned: 4  
       Workers Launched: 4
```

```

-> Partial Aggregate (cost=984930.66..984930.67 rows=1 width=8) (actual
   time=3501.321..3501.321 rows=1 loops=5)
   -> Parallel Seq Scan on trips (cost=0.00..983348.88  rows=645583
      width=8) (actual time=0.019..3428.119 rows=515013 loops=5)
         Filter: (trip_date = '2020-01-01'::date)
         Rows Removed by Filter: 899941
      Planning Time: 0.076 ms
      Execution Time: 3501.690 ms

```

(The query plan shows that a parallel sequential scan of the entire trips table is performed, and each worker fetches over 500,000 rows to get the total fare. The actual execution time is **3501.690 ms**, which is much slower than the previous query with partition pruning on)

The query plan shows that a parallel sequential scan of the entire trips table is performed, and each worker fetches over 6000 rows to get the total fare. This is a much less efficient query plan and will likely execute much more slowly than the previous query with partition pruning on.

## Partitioning versus sharding

The following *Table 6.3* will provide a comprehensive overview of partitioning and sharding in PostgreSQL 15:

Category	Partitioning	Sharding
Definition	Physical separation of a table's data into multiple pieces, each partition is treated as a separate table.	Process of horizontally partitioning data across multiple databases or servers.
Objective	Improve the performance of queries that access a small subset of rows from a large table.	Spread the load of a large database across multiple servers.
Ease of implementation	Partitioning is implemented in the database itself, either through the use of a partitioning key or by manually splitting the data into multiple tables.	Sharding is implemented at the application layer, with the application itself responsible for routing requests to the appropriate server.
Scalability	Scaling up a single database.	Scaling out a database across multiple servers.

Category	Partitioning	Sharding
Maintenance	Requires less maintenance, as the database handles the splitting of data.	Requires more maintenance, as the application must be kept up to date with changes in the data.

**Table 6.3: Partitioning versus sharding**

## Shard strategies and rebalancer

In PostgreSQL 15, there are several strategies for sharding data across multiple servers to improve scalability and performance. Some of the most common sharding strategies include:

- **Horizontal sharding:** Partitioning data based on rows, such as by assigning ranges of primary key values to different shards. This can help distribute the data evenly across multiple servers, which can improve query performance and scalability.
- **Vertical sharding:** partitioning data based on columns, such as by splitting a large table into smaller tables that contain a subset of the columns. This can help reduce the size of each table, which can improve query performance and reduce storage requirements.

**Note:** Both horizontal and vertical sharding can be combined with other sharding strategies, such as range, hash, or list-based sharding, to create more complex partitioning schemes that can improve query performance and scalability for larger databases.

- **Range-based sharding:** This strategy involves sharding data based on a range of values, such as date range, numerical range, and alphabetical range. This strategy is suitable for workloads that have predictable data access patterns and when data is evenly distributed across the range values.
- **List-based sharding:** This strategy involves sharding data based on a predefined list of values. This means that data is split across multiple shards based on specific values in a column, such as geographic regions, product categories, or customer segments.
- **Hash-based sharding:** This strategy involves sharding data based on a hash function applied to a specific column, such as a

user ID or hash of a URL. Hash sharding can work well when data is not evenly distributed across range values.

## Recipe 46: Configure sharding with Citus Data

Citus is an extension to PostgreSQL that enables distributed SQL queries across multiple nodes. Creating a PostgreSQL shard infrastructure with Citus involves the following steps:

**Install PostgreSQL on each node:** To get started, we need to install PostgreSQL. Refer to the recipe, Working with Installation from Binaries, from [Chapter 1, Up and running with PostgreSQL 15](#), for instructions on how to install PostgreSQL.

**Install Citus on each node:** Install Citus on each node by following the Citus installation. This involves adding the Citus repository, installing the Citus extension, and modifying the PostgreSQL configuration file(**postgresql.conf**).

### 1. Install Citus extension

- We use the Citus **rpm** installation method from yum using following command by downloading the Citus RPM from PostgreSQL download repository.

```
# Grant Execute permission to the script  
yum localinstall -y  
    citus_15-11.2.0-1.rhel9.x86_64.rpm  
    citus_15-devel-11.2.0-1.rhel9.x86_64.rpm  
    citus_15-llvmjit-11.2.0-1.rhel9.x86_64.rpm
```

- Add **citus** to the **shared\_preload\_libraries** configuration variable in the **postgresql.conf** file on each of the nodes, as shown in [Figure 6.11](#) for reference:

```
[root@pgdev ~]# cat /var/lib/pgsql/15/data/postgresql.conf |grep -i shared_preload_libraries  
shared_preload_libraries = 'pg_cron, citus'      # (change requires restart)  
[root@pgdev ~]#
```

**Figure 6.17:** citus shared library configuration

- Restart the PostgreSQL service for the changes to take effect.

```
systemctl restart postgresql-15.service
```

- Initialize the Citus extension on the database.

```
# Install Citus extension
```

```
create extension citus;
```

## 2. Set up a coordinator node

- a. Choose one of your PostgreSQL nodes to act as the coordinator node. This node will handle queries and direct them to the appropriate shards.
- b. Connect to the coordinator node as a superuser and set the configuration parameter **citus.shard\_replication\_factor** to the desired replication factor. The replication factor determines the number of replicas for each shard and provides fault tolerance. For example, to set a replication factor of **2**, use the following SQL command:

```
ALTER SYSTEM SET citus.shard_replication_factor = 2;
```

- c. Reload the PostgreSQL configuration for the changes to take effect.

```
SELECT pg_reload_conf();
```

## 3. Add worker nodes to coordinator

- a. Connect to the coordinator node as a superuser and execute the following SQL command to add the worker nodes to the coordinator:

```
SELECT * FROM citus_add_node('<worker_node_ip>', <worker_node_port>);
```

Replace **<worker\_node\_ip>** and **<worker\_node\_port>** with the actual values of each worker node.

## 4. Create distributed tables

- a. On the coordinator node, create distributed tables that will be sharded across the worker nodes. Distributed tables are logical tables that map to physical shards on worker nodes.
- b. Create a table and distribute the table based on the column, which serves as the primary key.

```
# Create a regular table (Optional)
```

```
CREATE TABLE customer (
    customer_id bigint,
    first_name text,
    last_name text,
    email text,
    address jsonb,
```

```
PRIMARY KEY (customer_id)
);
# Distribute the table
SELECT create_distributed_table('customer', 'customer_id');
```

## 5. Shard rebalancing

- a. After creating or modifying the distribution of a table, you should perform shard rebalancing to ensure an even distribution of data across the worker nodes. Execute the **rebalance\_table\_shards** function to rebalance the shards for the customer table. This will redistribute the data across the worker nodes.

```
SELECT rebalance_table_shards('customer');
```

By following this recipe, you have successfully configured sharding with Citus Data for your PostgreSQL 15 database. The Citus extension enables you to distribute data across multiple shards, allowing you to scale your database horizontally and handle large volumes of data efficiently.

## Rebalancer

There are several shard strategies for distributing data across multiple servers, as we discussed earlier. However, simply partitioning data is not enough; you also need to balance the data across the shards to ensure optimal performance and scalability. To do this, you can use a shard rebalancer.

Rebalancer is a tool or process that redistributes data across shards to optimize performance and scalability. A rebalancer is necessary because, over time, the distribution of data across shards can become overwhelmed, resulting in some shards being overloaded while others are underutilized. This can cause performance issues, reduce scalability, and create data consistency problems. The rebalancer works by monitoring the usage of the nodes and adjusting the data distribution accordingly. It can also be used to migrate data from one node to another in order to balance the load.

## Foreign data wrapper

**Foreign data wrapper (FDW)**, allows users to access data stored in multiple external sources as if it were a local PostgreSQL table. The FDW provides a bridge between external data sources and PostgreSQL,

allowing users to query, join, and manipulate data stored in these external sources as if it were part of the PostgreSQL database.

Foreign data wrappers in PostgreSQL are implemented as extensions, and they can be written in any language that is supported by PostgreSQL's extension infrastructure.

In fact, PostgreSQL's extension infrastructure provides a great deal of flexibility when it comes to implementing FDWs. Extension writers can use any language that is supported by PostgreSQL's PL/Proxy, PL/Perl, PL/Python, or PL/Java extensions to implement FDWs.

The FDW supports a wide range of data sources, including Oracle, Microsoft SQL Server, IBM DB2, MongoDB, HBase, Apache Solr, and Apache Cassandra. It also supports file formats such as CSV, JSON, XML, and Avro. With the PostgreSQL 15 FDW, users can easily join data from multiple external sources, as well as aggregate and filter data from these sources. The FDW supports all PostgreSQL data types, including arrays, ranges, and timestamps.

Some of the key improvements in PostgreSQL 15's FDW functionality include:

- **Improved pushdown support:** Allowing more efficient execution of queries against remote data sources. This can result in improved performance and scalability when interacting with remote databases, reducing the amount of data that needs to be transferred over the network.
- **Parallel query execution:** Parallel query execution with FDW queries help to reduce network traffic, as more of the query processing can be performed on the remote data source, multiple worker processes to execute an FDW query simultaneously.

## Conclusion

Partitioning and sharding are powerful techniques that can significantly improve database performance and scalability in PostgreSQL. However, they also come with their own set of challenges, such as increased complexity and the need for careful planning and design. In this chapter, we have covered the different techniques and tools available for partitioning and sharding in PostgreSQL, along with best practices and guidelines for implementation. By following these best practices and using the right tools, developers and database administrators can design and implement a partitioned or sharded database that is optimized for their specific use case. With the knowledge gained from

this chapter, readers can take advantage of partitioning and sharding to improve the performance and scalability of their PostgreSQL databases.

In the next chapter, we will continue to build upon these principles, diving deeper into the intricacies of replication and high availability. We will explore advanced replication scenarios, automatic failover mechanisms, and best practices for maintaining data consistency and reliability in mission-critical environments.

# CHAPTER 7

# Replication and High Availability

## Introduction

In today's digital era, the availability and reliability of databases are crucial for business continuity. In case of a database failure, businesses can face huge losses in terms of revenue, customer trust, and reputation. Therefore, it is essential to have a robust replication and high availability strategy in place to ensure maximum uptime and data protection. Replication and high availability are two critical aspects of database management that can help in achieving these objectives.

## Structure

In this chapter, we will cover the following topics:

- Replication and high availability
- Understanding the CAP theorem
- Replication classification
- Discovering replication slots
- Understanding proxy and load balancing
- Monitoring replication

## Objectives

The paramount objective of this chapter is to provide an extensive guide on replication and high availability in PostgreSQL, leveraging the versatile capabilities of Repmgr, Bucardo, and Pg\_auto\_failover. This

chapter is designed to offer comprehensive insights into database replication, exploring various replication architectures such as primary-standby and multi-master setups. Additionally, we delve into advanced strategies, including setting up delay standby, performing **point-in-time recovery (PITR)**, and promoting a standby database to the primary role in PostgreSQL.

In pursuit of high availability solutions, this chapter also guides you through the deployment of PostgreSQL Automatic Failover using the Pg\_auto\_failover tool. Additionally, we explore the usage of HAProxy for effective high availability, providing you with the knowledge to create resilient PostgreSQL environments. The inclusion of multi-master replication using Bucardo adds depth to our exploration, offering you a diverse toolkit for managing PostgreSQL databases in various scenarios.

## Replication and high availability

Replication is the process of creating and maintaining duplicate copies of data in multiple databases. This is done to ensure that the data is always available even in case of a failure of the primary database.

PostgreSQL supports several types of replications, including streaming replication, logical replication, and synchronous replication. Replication is the process of creating and maintaining duplicate copies of data in multiple databases. This is done to ensure that the data is always available even in case of a failure of the primary database. PostgreSQL supports several types of replications, including streaming replication, logical replication, and synchronous replication.

Streaming replication is a simple and easy-to-use method of replication that involves creating a standby server that replicates data from the primary server by continuously streaming the **Write-Ahead Log (WAL)** data. The standby server can be promoted to the primary server in case of a failure, ensuring minimal downtime. PostgreSQL 15 introduces several improvements to streaming replication, including support for replication slots that allow a standby server to retain WAL data even when the primary server is down.

Logical replication is a more advanced form of replication that allows for selective replication of specific tables or data subsets. This is particularly useful for scenarios where only a subset of the data needs to be replicated, such as for reporting or backup purposes. PostgreSQL 15 includes several enhancements to logical replication, including support for parallel apply and improved conflict resolution.

Synchronous replication is a form of replication that ensures that all data changes are committed to both the primary and standby servers before the transaction is considered complete. This provides a high level of data consistency but can impact performance. PostgreSQL 15 includes improvements to synchronous replication, including support for multiple standby servers and the ability to configure synchronous replication on a per-database basis.

High availability refers to the ability of a system to continue operating in the event of a failure. PostgreSQL 15 includes several features that help ensure high availability, including:

- **Automatic failover:** This feature allows for automatic promotion of a standby server to the primary server in case of a failure. PostgreSQL 15 includes enhancements to the automatic failover mechanism, including support for cascading replication and the ability to specify a custom failover target.
- **Consistent backup and recovery:** PostgreSQL 15 includes improvements to the backup and recovery mechanisms, including support for incremental backups and point-in-time recovery.
- **Improved monitoring and management:** PostgreSQL 15 includes enhancements to the monitoring and management tools, including support for query profiling and improved visibility into replication status.

Overall, PostgreSQL 15 offers a robust set of replication and high availability features that enable organizations to ensure that their critical data is always available and accessible.

## **Understanding the CAP theorem**

The CAP theorem is a fundamental concept in distributed computing that states that a distributed system cannot guarantee all three of the following properties at the same time: consistency, availability, and partition tolerance. PostgreSQL 15, like any other distributed system, is subject to the CAP theorem and must make trade-offs among these properties.

Consistency refers to the idea that all nodes in a distributed system see the same data at the same time. This means that if a write operation is performed, all subsequent read operations should return the updated data. PostgreSQL ensures consistency by using the traditional two-phase commit protocol, which ensures that all replicas commit the transaction before it is marked as complete.

Availability refers to the ability of a distributed system to respond to client requests even in the presence of node failures. PostgreSQL ensures availability by using synchronous replication, where multiple replicas of the data are maintained, and if one node fails, another node can be used to service requests.

Partition tolerance refers to the ability of a distributed system to continue operating even if network partitions occur, that is, some nodes are unable to communicate with each other. PostgreSQL ensures partition tolerance by using the quorum-based synchronous replication technique, where a quorum of nodes is required to commit a transaction.

PostgreSQL 15 achieves consistency and availability by using synchronous replication, and partition tolerance by using quorum-based synchronous replication. However, in the event of a network partition, the system may sacrifice consistency to ensure availability and partition tolerance. Therefore, when using PostgreSQL 15 in a distributed system, it is important to carefully consider the trade-offs between consistency, availability, and partition tolerance based on the specific requirements of the application.

## Replication classification

Replication is the process of copying and maintaining database objects and data from one database to another database, either on the same server or on different servers. There are several ways to classify database replication, based on the architecture and data synchronization methods.

- **Architecture-based replication:** Architecture-based replication is classified based on the location of the replication server and how the data is being transferred. Following are the types of replication architecture:
  - **Master-slave replication:** In this architecture, the master (Primary) database is responsible for the read-write operations, and the slave (Standby) database(s) is/are used for read-only operations. The primary database sends updates to the standby(s) on a regular basis.
  - **Master-master replication:** In this architecture, there are multiple master (Primary) databases, and each database can perform read and write operations. The databases communicate with each other to keep the data in sync.

- **Data synchronization-based replication:** Data synchronization-based replication is classified based on how the data is being transferred from one database to another. There are two types of replication methods:
  - **Synchronous replication:** In this method, the data is transferred from one database to another database in real-time. Before the transaction is considered complete, the data is committed to both the source and destination databases.
  - **Asynchronous replication:** In this method, the data is transferred from one database to another database in batches. The data is not committed to the destination database until the batch transfer is complete.

Replication is a critical component of high availability and disaster recovery solutions for databases. By replicating the database to multiple servers, we can ensure that data is available even if one of the servers fails.

## **Recipe 47: Scaling Postgres with primary-standby replication**

To scale PostgreSQL, one of the most common techniques is to use primary-standby replication. Primary-standby replication is a technique that involves replicating data from a primary database to one or more standby databases. The primary database is the primary database where all the write operations occur. The standby databases are read-only databases that receive a copy of the data from the primary database. Standby databases can be used to handle read queries, thereby reducing the load on the primary database.

There is no limit on the number of standby databases that can be created in primary-standby replication. However, it is important to note that the more standby databases you have, the more resources will be required to maintain the replication process.

Here is the recipe for setting up primary -standby replication with four standby databases:

1. First, install PostgreSQL on both the primary and standby servers. Ensure to install the same version of PostgreSQL on primary and all the slave servers. Refer to the recipe *Working with Installation from Binaries* from [Chapter 1](#), *Up and running with PostgreSQL 15* for instructions on how to setup PostgreSQL instance.

- Configure the primary database by modifying the **postgresql.conf** file, which is in the PostgreSQL data directory and update the following configuration parameters.

Here is an illustration:

```
# Primary configuration
wal_level = logical
max_wal_senders = 10
listen_addresses = 'Primary_IP'
```

Let us go through each of these configuration arguments in detail as given in *Table 7.1*:

Argument	Description
<b>wal_level</b>	Set it to logical to enable logical replication.
<b>max_wal_senders</b>	Set it to the maximum no of concurrent running WAL transfer.
<b>listen_addresses</b>	Set it to the IP address of the primary server to allow standby server to connect to it.

**Table 7.1:** WAL configuration on primary node

Modify these parameters using a text editor or by running the **ALTER SYSTEM** command. After modifying the **postgresql.conf** file, restart the primary database for the changes to take effect:

- Create a replication user that will be used by the standby database to connect to the primary database. The replication user must have the **REPLICATION** privilege. Create the replication user using the following SQL command:

```
# Create replication user
```

```
CREATE USER repuser WITH REPLICATION PASSWORD 'Replication@123';
```

Replace **repuser** and password with the username and password of your choice.

- Before taking the base backup, make sure to empty the data directory on all the standby servers to avoid any conflicts or errors during the replication setup process.
- Create a backup of the primary database that will be used to initialize the standby database. For this recipe, we are using the **pg\_basebackup** command to create a base backup of the primary database and which can be used to initialize a standby server or a replica.

```
# Create replication user
pg_basebackup -h primary_ip -U replication_user -D /path/to/backup -Fp -Xs -
P -v -R
```

Replace **primary\_ip** with the IP address of the primary server and **/path/to/backup** with the directory where you want to store the backup.

Below is a detailed description of the various options used in the command:

Options	Description
<b>-h</b>	Specifies the IP address or hostname of the primary server.
<b>-U</b>	Specifies the username to use when connecting to the primary server. This should be a user with replication permissions.
<b>-D</b>	Specifies the directory in which the backup will be created.
<b>-F</b>	Specifies the format of the backup. <b>p</b> indicates plain-text format, which is the default. Other options include <b>t</b> (tar format) and <b>c</b> (custom format).
<b>-X</b>	Specifies the method used for streaming the WAL records from the primary to the backup. <b>s</b> indicates synchronous streaming replication. Other options include <b>none (n)</b> and fetching ( <b>f</b> ) WAL files from archive).
<b>-P</b>	Shows the progress of the backup process.
<b>-v</b>	Increases the verbosity of the output, providing more detailed information during the backup process.
<b>-R</b>	Enables the creation of a replication slot. A replication slot is a point of continuity between the primary and the replica

**Table 7.2: pg\_basebackup command options**

Configure the standby database by creating a **standby.signal** file on all the standby database and update the following parameters:

```
# Create replication user
```

```
Restore_command = 'cp /pg_archive/%f %p'
```

```
primary_conninfo = 'host=primary_ip port=pg_port user=replication_user
password=password'
```

The **primary\_conninfo** configuration parameter is used in PostgreSQL to specify the connection information for a streaming replication standby server to connect to the primary server. Below is a detailed description of the options used in the **primary\_conninfo** string:

Options	Description
<b>host</b>	Specifies the hostname or IP address of the primary server to connect to.
<b>port</b>	Specifies the port number to use when connecting to the primary server. By default, PostgreSQL uses port 5432.
<b>user</b>	Specifies the username to use when connecting to the primary server. This should be a user with replication permissions.
<b>password</b>	Specifies the password to use when connecting to the primary server. This should be the password for the replication user specified in the user option.

**Table 7.3:** *primary\_conninfo command options*

- We can start the PostgreSQL cluster service on the standby servers. The standby servers will connect to the primary server and start streaming the WAL logs.
- Monitor the replication process to ensure that it is working properly. You can use the **pg\_stat\_replication** view to check the status of the replication. Run the following command on the primary server.
- If the primary database fails, perform a failover by promoting the standby database to be the new primary database. This will require updating the application connection strings to point to the new primary database.

We have now set up primary-standby replication in PostgreSQL 15 using streaming replication. The standby servers will receive a continuous stream of WAL logs from the primary server, ensuring that the data on all servers stays in sync.

## Using replication slots

In PostgreSQL replication, a replication slot is a persistent communication channel between a primary and a standby server. It is used to keep track of the WAL segments that the standby server has received and applied, and to ensure that the primary server does not remove any WAL segments that are still needed by the standby server.

Replication slots have several benefits, including:

- Ensuring that WAL segments needed by standby servers are not removed by the primary server before they have been received and applied.

- Allowing standby servers to recover quickly and efficiently in the event of a failover, because they do not need to catch up on a large backlog of WAL segment.
- Providing a simple way to monitor the replication status of standby servers.

## **Recipe 48: Scaling Postgres with multi-master replication**

Multi-master replication is a technique in PostgreSQL that allows multiple nodes to act as primary nodes, allowing for writes to be performed on any of the primary nodes and then automatically replicated to all the other primary nodes.

In multi-master replication, all nodes are considered equal, and changes made to any node are replicated to all other nodes. This means that all nodes can accept writes and reads at any time, providing high availability and scalability of the database by distributing the load across multiple nodes.

Tools for multi-master replication in PostgreSQL 15 include Bucardo, Pgpool-II, repmgr, OmniDB, Postgres-BDR, Londiste, and Postgres-XL. These tools provide various features like connection pooling, load balancing, conflict resolution, and logical and asynchronous replication.

In this recipe, we will set up a 2-node PostgreSQL multi-master replication using Bucardo.

Following are the prerequisite that make up a whole for this recipe:

- PostgreSQL 15 installed on both nodes
- Bucardo extension installed on both nodes
- Identical PostgreSQL configurations on both nodes
- Network connectivity between both nodes

1. Begin by installing PostgreSQL on both servers. It is crucial to ensure that the same version of PostgreSQL is installed on both servers. For detailed guidance on setting up a PostgreSQL instance, Refer to the recipe *Working with Installation from Binaries* in [Chapter 1, Up and Running with PostgreSQL 15](#). This step is a prerequisite for successfully executing the procedures outlined in this recipe.
2. Assuming that, we have downloaded the Bucardo packages

from the download site. If not, then download the latest package of Bucardo from their official website at <https://bucardo.org/Bucardo/> and continue with the installation of Bucardo package by using the following steps.

At first need to have **perl-CPAN** module installed on all the nodes.

```
# Install Package Manager for Perl module
```

```
yum install perl-CPAN
```

3. The Perl module **DBIx::Safe** is required for Bucardo installation, as it is a dependency for the Bucardo codebase. Download the **DBIx::Safe** package from the official website at <https://bucardo.org/Bucardo/> and continue with the installation on all the nodes. Following command will install **DBIx::Safe** package:

```
# Extract the DBIx::Safe downloaded archive
```

```
tar -xzf DBIx-Safe-1.2.5.tar.gz
```

```
# Change to the DBIx-Safe-1.2.5 directory once extracted
```

```
cd DBIx-Safe-1.2.5
```

```
# Build and install the module using make
```

```
perl Makefile.PL
```

```
make
```

```
make install
```

4. Since **DBIx::Safe** package successfully installed on both nodes, in this step will install **Bucardo** package:

```
# Extract the Bucardo downloaded archive
```

```
tar -xzf Bucardo-5.6.0.tar.gz
```

```
# Change to the Bucardo-5.6.0 directory once extracted
```

```
cd Bucardo-5.6.0
```

```
# Build and install the module using make
```

```
perl Makefile.PL
```

```
make
```

```
make install
```

5. Install the Perl procedural language for the PostgreSQL database server on both the nodes. Download the Perl procedural language package from the official website at <https://yum.postgresql.org/rpmchart/> and continue with the installation. Following command will install Perl procedural language:

```
# Install Perl procedural language
```

```
yum localinstall -y postgresql15-plperl-15.1-1PGDG.rhel9.x86_64.rpm
```

6. At this stage we are in a state to initialize the Bucardo database schema using the following command on the both primary nodes:

```
# Initialize the Bucardo database schema
```

```
bucardo install
```

Please refer to the following figure:

```

[root@pgmaster soft]#
[root@pgmaster soft]# bucardo install
This will install the bucardo database into an existing Postgres cluster.
Postgres must have been compiled with Perl support,
and you must connect as a superuser

Current connection settings:
1. Host: <none>
2. Port: 5432
3. User: bucardo
4. Database: bucardo
5. PID directory: /var/run/bucardo
Enter a number to change it, P to proceed, or Q to quit: 1

Change the host to: localhost

Changed host to: localhost
Current connection settings:
1. Host: localhost
2. Port: 5432
3. User: bucardo
4. Database: bucardo
5. PID directory: /var/run/bucardo
Enter a number to change it, P to proceed, or Q to quit: 3

Change the user to: postgres

Changed user to: postgres
Current connection settings:
1. Host: localhost
2. Port: 5432
3. User: postgres
4. Database: bucardo
5. PID directory: /var/run/bucardo
Enter a number to change it, P to proceed, or Q to quit: 4

Change the database name to: postgres

Changed database name to: postgres
Current connection settings:
1. Host: localhost
2. Port: 5432
3. User: postgres
4. Database: postgres
5. PID directory: /var/run/bucardo
Enter a number to change it, P to proceed, or Q to quit: P

Password for user postgres:
Password for user postgres:
Creating superuser 'bucardo'
Password for user postgres:
Attempting to create and populate the bucardo database and schema
Password for user postgres:
Database creation is complete

Updated configuration setting "piddir"
Installation is now complete.
If you see errors or need help, please email bucardo-general@bucardo.org

You may want to check over the configuration variables next, by running:
bucardo show all
Change any setting by using: bucardo set foo=bar

[root@pgmaster soft]#

```

```

graph TD
    A["[root@pgmaster soft]# bucardo install"] --> B["Current connection settings:  
1. Host: <none>  
2. Port: 5432  
3. User: bucardo  
4. Database: bucardo  
5. PID directory: /var/run/bucardo  
Enter a number to change it, P to proceed, or Q to quit: 1"]
    B --> C["Change the hostname"]
    C --> D["Change the host to: localhost"]
    D --> E["Changed host to: localhost  
Current connection settings:  
1. Host: localhost  
2. Port: 5432  
3. User: bucardo  
4. Database: bucardo  
5. PID directory: /var/run/bucardo  
Enter a number to change it, P to proceed, or Q to quit: 3"]
    E --> F["Change the user"]
    F --> G["Change the user to: postgres"]
    G --> H["Changed user to: postgres  
Current connection settings:  
1. Host: localhost  
2. Port: 5432  
3. User: postgres  
4. Database: bucardo  
5. PID directory: /var/run/bucardo  
Enter a number to change it, P to proceed, or Q to quit: 4"]
    H --> I["Change the database"]
    I --> J["Change the database name to: postgres"]
    J --> K["Changed database name to: postgres  
Current connection settings:  
1. Host: localhost  
2. Port: 5432  
3. User: postgres  
4. Database: postgres  
5. PID directory: /var/run/bucardo  
Enter a number to change it, P to proceed, or Q to quit: P"]
    K --> L["Proceed with the configured setting"]
    L --> M["Password for user postgres:  
Password for user postgres:  
Creating superuser 'bucardo'  
Password for user postgres:  
Attempting to create and populate the bucardo database and schema  
Password for user postgres:  
Database creation is complete  
  
Updated configuration setting \"piddir\"\nInstallation is now complete.\nIf you see errors or need help, please email bucardo-general@bucardo.org  
  
You may want to check over the configuration variables next, by running:  
bucardo show all  
Change any setting by using: bucardo set foo=bar"]

```

**Figure 7.1:** Initialize bucardo schema

Referring to [Figure 7.1](#), The bucardo install command installs the Bucardo database and schema into an existing PostgreSQL cluster. The user needs to be logged in as a PostgreSQL superuser to run this command.

The command prompts the user to enter the connection settings for the PostgreSQL cluster. In this example, the default values are displayed, and the user is prompted to change them if necessary. The user is asked to enter the host, user, and database name to connect to. The default values are none for the host, bucardo for the user, and bucardo for the database. The user is prompted to enter a number to change each setting, or to press *P* to proceed with the default settings.

After entering the correct connection settings, the user is prompted to enter the password for the PostgreSQL superuser `postgres`. The command then creates the Bucardo database and schema and populates it with the necessary tables and data. It also creates a Bucardo superuser with the username **bucardo**.

Finally, the command updates the Bucardo configuration setting `paddir` and prints a message indicating that the installation is complete. The user is prompted to run the **bucardo show all** command to check the configuration variables.

Start the bucardo service and verify the status (see [Figure 7.2](#)):

```
# Start bucardo service
bucardo start
# Check bucardo service status
bucardo status
```

Please refer to the following figure:

```
[root@pgmaster postgresql]# bucardo start
Checking for existing processes
Removing file "/var/run/bucardo/fullstopbucardo"
Starting Bucardo
```

**Figure 7.2:** bucardo start-up

The **bucardo start** command starts the Bucardo daemon, which is a background service that performs replication between databases.

Referring to [Figure 7.2](#), the command initially scans for any pre-existing Bucardo processes. If the `/var/run/bucardo/fullstopbucardo` file is present, it is removed. Subsequently, the command triggers the Bucardo daemon's initiation and generates a confirmation message indicating its successful start.

Execute the command **bucardo status** to verify the status of the Bucardo daemon:

```
[root@pgmaster postgresql]# bucardo status  
PID of Bucardo MCP: 6238  
No syncs have been created yet.  
[root@pgmaster postgresql]#
```

**Figure 7.3:** bucardo status

Referring to [Figure 7.3](#), the **bucardo status** command displays the current status of the Bucardo replication system. The command returns the **process ID (PID)** of the Bucardo **Master Control Program (MCP)**, which is the main process that manages replication. In this case, the PID is 6238.

The command also reports that no synchronization (sync) configurations have been created yet, which means there is no active data replication taking place:

1. This step is all about configuring replication between both the primary node.
  - a. On the first primary node, add both the node as a source database server using **bucardo add database** command as described in the following command:

*# Add first primary node*

```
bucardo add database pg1 dbname=postgres host=192.168.187.133  
port=5432 user=bucardo
```

*# Add second primary node*

```
bucardo add database pg2 dbname=postgres host=192.168.187.134  
port=5432 user=bucardo
```

- b. On the first primary node, create a database group using **bucardo add dbgroup** command as described in the following command:

*# Add database group on first primary node*

```
bucardo add dbgroup pg_dbgroup pg1:source pg2:source
```

**Note:** In Bucardo, a **dbgroup** is a way to group together one or more Postgres databases or database clusters into a single entity. This can be useful for setting up multi-master replication or for other complex replication scenarios where multiple databases need to be synchronized.

- c. On the first primary node, add the database object like **tables** and **sequences** that synchronized with second primary database:

```
# Add tables for synchronization on first primary node
```

```
bucardo add all tables db=pg1 -herd=relgrp
```

```
# Add sequences for synchronization on first primary node
```

```
bucardo add all sequences db=pg1 -herd=relgrp
```

- d. On the first primary node, create a new sync configuration, which specifies the tables to replicate and how they should be replicated:

```
# Add sync configuration on first primary node
```

```
bucardo add sync sync1 relgroup=relgrp dbs=pg1:source,pg2:source
```

- e. On the second primary node, add both the node as a source database server using **bucardo add database** command as described in the following command:

```
# Add first primary node
```

```
bucardo add database pg1 dbname=postgres host=192.168.187.133
```

```
port=5432 user=bucardo
```

```
# Add second primary node
```

```
bucardo add database pg2 dbname=postgres host=192.168.187.134
```

```
port=5432 user=bucardo
```

- f. On the second primary node, create a database group using **bucardo add dbgroup** command as described in the following command:

```
# Add database group on second primary node
```

```
bucardo add dbgroup pg_dbgroup pg1:source pg2:source
```

- g. On the second primary node, add the database object like tables and sequences that synchronized with first primary database:

```
# Add tables for synchronization on first primary node
```

```
bucardo add all tables db=pg1 -herd=relgrp
```

```
# Add sequences for synchronization on first primary node
```

```
bucardo add all sequences db=pg1 -herd=relgrp
```

- h. On the second primary node, create a new sync configuration, which specifies the tables to replicate and how they should be replicated:

```
# Add sync configuration on first primary node
```

```
bucardo add sync sync1 relgroup=relgrp dbs=pg1:source,pg2:source
```

- i. Start the Bucardo daemon on both nodes:

```
# Restart bucardo demon on both the primary  
node
```

```
bucardo restart
```

2. Now, any changes made to the any table on either node will be replicated to the other node. Note that both nodes will be active primary, meaning that updates can be made to either node and will be replicated to the other node. This allows for a true multi-primary replication setup. First, verify that Bucardo sync status from both the node using the following command:

- a. Verify sync status on first primary node using bucardo status command, If Bucardo is running, we see a list of syncs and other information in the

*Figure 7.4:*

```
[root@pgmaster ~]# bucardo status  
PID of Bucardo MCP: 8307  
Name      State    Last good     Time     Last I/D     Last bad     Time  
=====+=====+=====+=====+=====+=====+=====+=====+  
sync1 | Good   | 15:49:32 | 6s      | 0/0       | none       |
```

*Figure 7.4: bucardo - Verify sync status on primary*

- b. Verify sync status on second primary node using **bucardo status** command, if Bucardo is running, we see a list of syncs and other information in the *Figure 7.5*:

```
[root@pgmaster2 ~]# bucardo status  
PID of Bucardo MCP: 7831  
Name      State    Last good     Time     Last I/D     Last bad     Time  
=====+=====+=====+=====+=====+=====+=====+=====+  
sync1 | Good   | 15:49:24 | 21s     | 2/2       | none       |
```

*Figure 7.5: bucardo: Verify sync status on second primary*

3. Here are the example steps to perform some transactions on the v node and sync on the other primary node.

- a. Connect to the first primary node, and create a test table and insert some data into it:

```
# Connect to the 1st primary node
```

```
psql -h <1st_primary_node_ip> -p <1st_primary_node_port> -U  
<bucardo_user> -d <db_name>
```

```

# Create a table "test_table" on the first primary node
create table test_table (id serial primary key, name text);
# Insert the data on table "test_table" on the first primary node
insert into test_table (name) values ('John'), ('Jane'), ('Bob');

```

- b. Connect to the second primary node, create a test table without any dataset and verify dataset does not exist:

```

# Connect to the Second primary node
psql -h <2nd_primary_node_ip> -p <2nd_primary_node_port> -U
<bucardo_user> -d <db_name>
# Create a table "test_table" on the first primary node
create table test_table (id serial primary key, name text);

```

- c. Add a new table that synchronize with second primary database.

```

# Add tables for synchronization on first primary node
bucardo add all tables db=pg1 --herd=relgrp

```

- d. Following is the output of the execution of above command for table synchronization:

```

[root@pgmaster ~]# bucardo add all tables db=pg1 --herd=relgrp
Added table public.test_table to relgroup relgrp
New tables added: 1
Already added: 1

```

**Figure 7.6:** bucardo: Add table for synchronization

- e. Create a new sync in bucardo on both the baster node, that will replicate the **test\_table** dataset from the first primary node to the second primary node:

```

# Add sync configuration on both primary node

```

```

bucardo add sync sync2 relgroup=relgrp dbs=pg1:source,pg2:source

```

- f. Start the Bucardo daemon on both nodes.

```

# Restart bucardo demon on both the primary node

```

```

bucardo restart

```

- g. Verify the current status of the Bucardo replication system on both the primary node using **bucardo status** command.

- h. Following is the output of the execution of **bucardo status** command (see *Figure 7.7*):

1 <sup>st</sup> Master Node							
[root@pgmaster ~]# bucardo status							
PID of Bucardo MCP: 9047							
Name	State	Last good	Time	Last I/D	Last bad	Time	
sync1	Good	17:14:17	0s	0/0	none		
sync2	Good	17:14:17	0s	0/0	none		
2 <sup>nd</sup> Master Node							
[root@pgmaster2 ~]# bucardo status							
PID of Bucardo MCP: 8661							
Name	State	Last good	Time	Last I/D	Last bad	Time	
sync1	Good	17:14:29	1s	0/0	none		
sync2	Good	17:14:29	1s	0/0	none		

*Figure 7.7: bucardo: Verify replication status on both the primary node*

Referring to *Figure 7.7*, the output shows that there are two syncs named **sync1** and **sync2**, and both are in the **Good** state, meaning they are operating normally. The syncs have not encountered any bad completion since the last successful completion, as indicated by the **none** value under the **Last bad** column.

- i. Initiate a connection with the secondary primary node and ensure the successful replication of new data. This verification should reveal the presence of the recently inserted data that originated from the primary node:

*# Connect to the Second primary node*

```
psql -h <2nd_primary_node_ip> -p <2nd_primary_node_port> -U
<bucardo_user> -d <db_name>
```

*# Verify the populated from second primary node*

```
Select * from test_table;
```

Please refer to the following figure:

```
postgres=# select * from public.test_table;
 id | name
----+-----
  4 | John
  5 | Jane
  6 | Bob
(3 rows)
```

**Figure 7.8:** bucardo: Verify replicated data on second primary

As shown in [Figure 7.8](#), the output indicates that the second primary node has successfully populated the data into the **test\_table** table. This indicates that the synchronization between the two primary nodes has been successful and the data on both nodes is now consistent.

## Recipe 49: Setup delay standby in PostgreSQL

Delay standby is a feature of PostgreSQL's replication system that allows you to introduce a delay in the application of changes from the primary server to the standby server. This delay can be useful in scenarios where you want to protect against accidental data changes or errors on the primary server, as it provides a buffer time for you to catch and correct those errors before they are applied to the standby server.

In a delay standby configuration, the standby server receives WAL records from the primary server but applies those records after a specified time delay. This delay can be set to any value that you choose, and it can be adjusted as needed based on your application's requirements.

In this recipe, we will discuss in detail how to set up PostgreSQL 15 replication with delay standby. A delay standby is a standby server that intentionally lags behind the primary server by a certain amount of time. This is useful in situations where you need to maintain a backup copy of your primary database, but also want to have a delay in replication to protect against accidental data loss or corruption.

Let us begin the recipe on how to set up delay standby in PostgreSQL 15:

1. First, install and configure PostgreSQL on each servers and ensure to install the same version of PostgreSQL on both the servers. Refer to the recipe *Working with Installation from Binaries* from

*Chapter 1, Up and running with PostgreSQL 15* for instructions on how to setup PostgreSQL instance.

2. On the primary server, open the **postgresql.conf** file and add the following lines to enable WAL-based replication:

```
# WAL configuration on primary server
```

```
wal_level = replica
```

```
max_wal_senders = 10
```

```
archive_mode = on
```

```
archive_command = 'cp %p /path/to/archive/%f'
```

3. The **wal\_level** parameter must be set to replica in order to enable WAL-based replication. The **max\_wal\_senders** parameter controls the maximum number of standby servers that can connect to the primary server.

4. Start the PostgreSQL service on the standby server. The server will start in standby mode and will begin streaming WAL segments from the primary server:

```
# Start postgresql cluster service
```

```
systemctl start postgresql-15.service
```

5. Create a backup of the primary server, run the following command on the standby server to take a full backup:

```
# Create a backup of the standby server
```

```
pg_basebackup -h <primary_serv_ip> -U replicapg -p 5432 -D
```

```
/PG_DATAPATH/ -Fp -Xs -P -R
```

This command connects to the primary server using the replicapg user, creates a backup in the specified directory, and streams the WAL files to the backup directory.

6. Add the delay parameter, Add the following parameter to the **postgresql.conf** file on the standby server:

```
# Configuration on postgresql.conf file
```

```
recovery_min_apply_delay = '5min'
```

7. The **recovery\_min\_apply\_delay** configuration parameter can be set in the **postgresql.conf** file of the standby server. If we set it in the **postgresql.conf** file, it will apply to all future recovery processes on that server.

8. To set **recovery\_min\_apply\_delay** in the **postgresql.conf** file, add the following line:

```
# Configuration on postgresql.conf file
```

```
recovery_min_apply_delay = '5min'
```

9. Start the PostgreSQL service on the standby server. The server will start in standby mode and will begin streaming WAL segments from the primary server.

```
# Start postgresql cluster service
```

```
systemctl start postgresql-15.service
```

We have now set up a delayed standby in PostgreSQL 15. The standby server will lag behind the primary server by the specified amount of time.

10. Let us perform a lag test to ensure that the replication delay is working as expected.

- a. Connect to the primary server and determine the current **log sequence number (LSN)** using the following SQL command:

```
# Verify the current LSN from Primary server
```

```
SELECT pg_current_wal_lsn();
```

- b. Connect to the delayed standby server and determine the latest received LSN and latest applied LSN using the following SQL commands.

```
# Verify the LAST Receive and Replay LSN from Delay Standby server
```

```
select now(), pg_is_in_recovery(), pg_is_wal_replay_paused(),
       pg_last_wal_receive_lsn(), pg_last_wal_replay_lsn();
```

11. Referring to *Figure 7.9*, in the output, the value of **pg\_last\_wal\_receive\_lsn()** is **0/3B000000** and the **value of pg\_last\_wal\_replay\_lsn()** is **0/3909E3A0**. In the following steps we will use the **pg\_wal\_lsn\_diff** function to calculate the lag size. Please refer to the following figure:

	now	pg_is_in_recovery()	pg_is_wal_replay_paused()	pg_last_wal_receive_lsn()	pg_last_wal_replay_lsn()
1	2023-04-02 15:59:03.832234+08	true	false	0/3B000000	0/3909E3A0

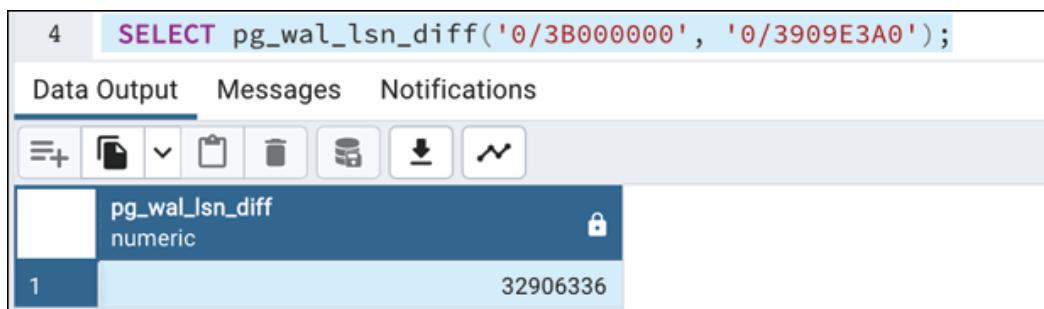
*Figure 7.9: Delay Standby: Verify WAL LSN*

12. Calculate the lag as the difference between the current LSN on the primary server and the latest applied LSN on the delayed standby server.

```
# Verify LSN difference
```

```
SELECT pg_wal_lsn_diff('0/3B000000', '0/3909E3A0');
```

13. To calculate the replication lag, we need to subtract the value of **`pg_last_wal_replay_lsn()`** from the value of **`pg_last_wal_receive_lsn()`**. The difference between these two values is the amount of WAL data that has been received by the standby but not yet replayed, and represents the replication lag in terms of the number of bytes. Please refer to the following figure:



**Figure 7.10:** Delay Standby: WAL LSN difference between primary and standby

14. Referring to [Figure 7.10](#), The output of the **`pg_wal_lsn_diff()`** function indicates the replication lag in bytes between the primary server and the standby server. In this case, the function takes two arguments representing the LSN values of the last WAL record received by the standby server (**0/3B000000**) and the last WAL record replayed by the standby server (**0/3909E3A0**). The function then calculates the difference between these two values and returns the result.

## Recipe 50: Performing PITR recovery using delay standby

Performing a point-in-time recovery using a delayed standby involves recovering a PostgreSQL database to a specific point in time by replaying transaction logs on a standby server that has a delay. The delay allows you to recover to a point in time before the point of failure, and potentially recover from data loss or corruption.

In this recipe, we will be setting-up PostgreSQL cluster with a detailed instructions to perform a PITR recovery using a delayed standby on PostgreSQL 15:

1. By referring to the previous *Setup delay standby in PostgreSQL* recipe, we are reusing the same recipe steps until step 5 to configure this PostgreSQL cluster. At this point, our PostgreSQL database cluster is a running and streaming replication, enabled between the primary and the delayed standby server.

- Determine the point in time to which we want to recover. We can do this by examining the transaction logs or by using a backup taken at the desired point in time.

Let us assume that we have a PostgreSQL database that is experiencing some issues (critical data deleted from one table) and we want to perform a PITR recovery using a specific timestamp:

- Identify the LSN range for the time when the data was deleted. Referring to the *Figure 7-11*, In this case, the LSN value before the deletion was **0/B2426758**, and the LSN value after the deletion was **0/B5000110**. So, the LSN range for the deletion is **0/B2426720** to **0/B5000110**. In this case, the WAL file containing the LSN range is **00000001000000000000B2** and **00000001000000000000B5**:

Before Deletion		
pg_walfile_name	pg_current_wal_lsn	Pg_size.pretty
text	0/B2426758	691 MB

After Deletion		
pg_walfile_name	pg_current_wal_lsn	Pg_size.pretty
text	0/B5000110	691 MB

*Figure 7.11: PITR Recovery: LSN for before and after data deletion*

- Check the WAL files in the PostgreSQL data directory for the relevant time frame. We can use the **pg\_wal** subdirectory to find the WAL files by using following command:

```
# Analyze the WAL
```

```
pg_waldump -s 0/B2426758 -e 0/B5000110
```

**Note: WAL files are named in a format that includes a timestamp, such as “00000001000000000000B2” and “00000001000000000000B5” or use pg\_waldump to analyze the WAL file and identify the transaction to rollback.**

- Open the PostgreSQL configuration file **postgresql.conf** on the standby server and set the following parameter:

```
# Update recovery timestamp on postgresql.conf file
```

```
recovery_target_time = '2023-04-02 20:36:00'
```

4. If we want to recover to a specific LSN value instead of a timestamp, we can use the following configuration instead:

```
# Update recovery timestamp on postgresql.conf file
```

```
recovery_target_lsn = '0/B2426758 '
```

5. Restart the PostgreSQL service on standby server for the changes to take effect:

```
# Start postgresql cluster service
```

```
sudo systemctl start postgresql-15.service
```

When the standby server restarts, it will start the recovery process and apply WAL records until it reaches the target time or LSN value specified in the configuration file. Note that the recovery process may take some time depending on the size of the WAL archive and the amount of data that needs to be replayed.

Once the recovery process is complete, the database will be in a consistent state up to the specified point in time. You can now connect to the database and verify that the recovered data is correct.

Note that during the recovery process, any data that was modified after the specified point in time will be lost. Therefore, it is important to carefully choose the recovery point and to ensure that all necessary data is backed up before starting the recovery process.

## Recipe 51: PostgreSQL 15 promote standby database to primary

In PostgreSQL, a standby server is a read-only replica of the primary server, which continuously receives WAL records from the primary server and applies them to its own database. Promoting a standby server to the primary server means making it the new primary server and stopping the replication from the old primary server.

In this recipe will work on the detailed steps to promote a standby server as the primary server in PostgreSQL, the following steps can be taken:

1. Stop all connections to the application that is using the primary server. We can do this by gracefully shutting down the application or redirecting its connections to another server. This is an important step to prevent any data loss or inconsistency during the promotion process.
2. On primary server, issue the command **SELECT pg\_switch\_wal()**; to ensure that the standby server has the latest

WAL records:

```
# Check the replication status
```

```
SELECT pg_switch_wal();
```

3. Connect to the standby server using **psql** and issue the following command to check the replication status:

```
# Check the replication status
```

```
SELECT pg_is_in_recovery();
```

This should return a value of **t** which means the server is in recovery mode.

4. Wait until the server has caught up with the primary server and is no longer receiving WAL records. You can check this by periodically issuing the command **SELECT pg\_last\_wal\_receive\_lsn()**. When this value remains constant for a few minutes, it means that the server is caught up.
5. Issue the command **SELECT pg\_promote()**; to promote the standby server to the primary server.

```
# Promote the standby server to the primary server.
```

```
SELECT pg_promote();
```

6. On the newly promoted primary, Check the status of the server by issuing the command **SELECT pg\_is\_in\_recovery()**. By referring the [Figure 7.12](#), The result should be **f**, indicating that the server is now the primary server:

The screenshot shows a PostgreSQL terminal window. The command entered is `3 SELECT pg_is_in_recovery();`. Below the command, there are tabs for Data Output, Messages, and Notifications. Under the Data Output tab, there is a table with one row. The table has two columns: the first column contains the value '1' and the second column contains the value 'false'. The table header includes the column name 'pg\_is\_in\_recovery' and its type 'boolean'. There is also a lock icon next to the column name.

1	false
---	-------

**Figure 7.12:** Recovery status post promoting new primary

7. Update the client connection strings to point to the new primary server. These steps will help to promote a standby server to the primary server in PostgreSQL.

## Recipe 52: Installing and configuring repmgr for PostgreSQL cluster

`repmgr` is a tool for managing PostgreSQL replication clusters. It provides features for automatic failover, promotion, and monitoring of replication nodes. `repmgr` is designed to simplify the management of complex replication clusters, making it easier to maintain high availability and reduce downtime.

The need for `repmgr` arises from the complexity of managing large PostgreSQL replication clusters. Without proper management tools, replicating data across multiple nodes can be a daunting task, especially when it comes to managing failover and promotion. This can lead to extended downtime in the event of a node failure or network partition.

Before installing and configuring `repmgr` for PostgreSQL 15, there are several prerequisites that need to be met. These include (see [Table 7.4](#)):

Pre-req	Description
PostgreSQL version	<code>repmgr</code> requires PostgreSQL 15 or later to be installed on all nodes in the replication cluster (specific to this recipe).
PostgreSQL contrib	<code>repmgr</code> depends on the <b>pg_rewind</b> tool which is part of the PostgreSQL contrib package. Ensure that PostgreSQL contrib is installed on all nodes.
SSH	uses SSH to connect to remote nodes in the replication cluster. Ensure that SSH is installed and properly configured on all nodes.
rsync	uses rsync to synchronize data between nodes. Ensure that rsync is installed and properly configured on all nodes.
Repmgr user	Requires a dedicated PostgreSQL user for replication management.
Superuser privileges	Requires superuser privileges on the PostgreSQL cluster. Ensure that the user running the installation and configuration has superuser privileges.

**Table 7.4:** `repmgr` prerequisite

The following recipe example steps to configuring `repmgr` for PostgreSQL 15 is based on two-nodes cluster. However, the steps for configuring `repmgr` remain largely the same regardless of the number of nodes in the cluster. However, as the number of nodes increases, the complexity of the configuration may also increase.

Now, let us go through the recipe steps to install and configure `repmgr` for a PostgreSQL 15 cluster:

1. First, install and configure PostgreSQL on each server and ensure to install the same version of PostgreSQL on both the servers. Refer to the recipe *Working with Installation from Binaries* from [Chapter 1, Up and running with PostgreSQL 15](#) for instructions on how to setup PostgreSQL instance.
2. On the primary server, open the **postgresql.conf** file and add the following lines to enable WAL-based replication.

**# WAL configuration on primary server**

```
wal_level = replica
max_wal_senders = 10
archive_mode = on
archive_command = 'cp %p /path/to/archive/%f'
hot_standby = on
# repmgr library configuration on primary server
shared_preload_libraries = 'repmgr'
```

3. The first set of configuration options pertains to WAL, which is a key component of PostgreSQL's replication system.

The second set of configuration options pertains to the **repmgr** library, which is a PostgreSQL extension used for managing replication clusters:

1. On the primary server, open the **pg\_hba.conf** file and add the following lines to enable authentication configuration for replication.

**# Configure pg\_hba.conf for repmgr**

```
host repmgr repmgr 192.168.187.134/32 trust
host repmgr repmgr 192.168.187.133/32 trust
```

2. Restart the PostgreSQL service on the primary node for the change to take effect on primary node.

**# Start postgresql cluster service**

```
systemctl start postgresql-15.service
```

3. If we have downloaded the **repmgr** packages from the download site. If not, then download the latest package of repmgr from PostgreSQL official website at <https://download.postgresql.org/pub/repos/yum/15> and continue with the installation of repmgr package by using the following command on both the primary and standby node:

**# Install repmgr on primary and standby node**

```
yum localinstall -y
repmgr_15-5.3.3-1.rhel9.x86_64.rpm
```

```
repmgr_15-devel-5.3.3-1.rhel9.x86_64.rpm  
repmgr_15-llvmjit-5.3.3-1.rhel9.x86_64.rpm
```

4. Create a **repmgr** database and user in PostgreSQL on primary node. We can do this by running the following SQL commands as the Postgres superuser:

```
# Create user for repmgr with replication privileges
```

```
CREATE USER repmgr WITH REPLICATION SUPERUSER PASSWORD  
'Repmgr123';  
# Create database for repmgr  
CREATE DATABASE repmgr owner repmgr;  
# Set the search path to repmgr  
ALTER USER repmgr SET search_path TO repmgr, "$user", public;
```

repmgr uses its own database to store metadata about the replication cluster. Once the repmgr metadata database is set up, repmgr uses it to store information about the replication cluster and to perform various administrative tasks, such as monitoring nodes and managing failover and promotion. The metadata stored in the **repmgr** database is critical to the proper functioning of **repmgr** and should be backed up regularly along with the other databases in the PostgreSQL cluster:

1. On the primary server configure **repmgr**, open the **/etc/repmgr/15/repmgr.conf** file and add the following lines:

```
# Repmgr configuration on primary node
```

```
node_id=1  
node_name='node_master'  
conninfo='host=192.168.17.134 dbname=repmgr port=5432 user=repmgr  
password=Repmgr123 connect_timeout=2'  
data_directory='/var/lib/pgsql/15/data'  
log_file='/var/lib/pgsql/15/data/log/repmgr.log'
```

Let us go through each of these configuration options in detail as given in [Table 7.5](#):

Configuration	Description
<b>node_id</b>	Unique identifier for this node in the replication cluster.
<b>node_name</b>	Human-readable name for this node in the replication cluster.
<b>conninfo</b>	Connection information for this node's PostgreSQL instance.
	Path to this node's PostgreSQL data directory.

<b>data_directory</b>	
<b>log_file</b>	Specifies the location of the repmgr log file.

**Table 7.5:** repmgr configuration options

2. On the primary node, register the repmgr primary node by running the following command as the postgres superuser.

```
# Register primary node
```

```
/usr/pgsql-15/bin/repmgr -f /etc/repmgr/15/repmgr.conf primary register
```

As shown in *Figure 7.13*, This command registers the current server as the primary node in the replication cluster:

```
[postgres@pgmaster2 ~]$ /usr/pgsql-15/bin/repmgr -f /etc/repmgr/15/repmgr.conf primary register
INFO: connecting to primary database...
NOTICE: attempting to install extension "repmgr"
NOTICE: "repmgr" extension successfully installed
NOTICE: primary node record (ID: 1) registered
```

**Figure 7.13:** repmgr: Register primary node

3. Verify the registered nodes in a PostgreSQL 15 cluster using **repmgr** by executing the following SQL script on primary node:

```
# Verify the registered node
```

```
SELECT node_id, node_name, type, conninfo, repluser, config_file FROM
repmgr.nodes;
```

As shown in *Figure 7.14*, this will return a list of nodes that are currently registered with **repmgr**, including their ID, name, type, status, and location:

4 SELECT node_id, node_name, type, active, conninfo, repluser, config_file FROM repmgr.nodes;							
Data Output		Messages		Notifications			
node_id	[PK] integer	node.name	text	type	text	active	boolean
1		node.master	primary	true	host=192.168.187.134 dbname=repmgr port=5432 user=repmgr password=password connect_timeout=2	repmgr	config_file

**Figure 7.14:** repmgr: Verify registered node

4. Configure the **repmgr.conf** file on the standby node to specify the connection information for the standby node, like how it was done for the primary node. We will also need to set the **node\_id** and **node\_name** parameters to unique values for the standby node.

```
# Repmgr configuration on standby node
```

```
node_id=2
```

```
node_name='node_slave'
```

```
conninfo='host=192.168.17.133 dbname=repmgr port=5432 user=repmgr
```

```
password=password connect_timeout=2'
```

```
data_directory='/var/lib/pgsql/15/data'
```

```
log_file='/var/lib/pgsql/15/data/log/repmgr.log'
```

5. Use the **repmgr** clone command on the standby node to clone the primary node to the standby node.

```
# Execute dry-run on the standby node
```

```
/usr/pgsql-15/bin/repmgr -h 192.168.187.134 -U repmgr -d repmgr -f  
/etc/repmgr/15/repmgr.conf -D /var/lib/pgsql/15/data standby clone --dry-run
```

As shown in *Figure 7.15*, this command will connect to the standby node and set up a replication slot, as well as add the standby node to the repmgr metadata in a dry-run mode. The command should be run with the **--dry-run** flag first to check the configuration. Please refer to the following figure:

```
# Output of dry-run on the standby node  
NOTICE: using provided configuration file "/etc/repmgr/15/repmgr.conf"  
WARNING: following problems with command line parameters detected:  
-D--pgdata will be ignored if a repmgr configuration file is provided  
NOTICE: destination directory "/var/lib/pgsql/15/data" provided  
INFO: connecting to source node  
DETAIL: connection string is: host=192.168.187.134 user=repmgr dbname=repmgr  
INFO: "replication" extension is installed in database "replica"  
INFO: replication slot usage not requested; no replication slot will be set up for this standby  
INFO: parameter "max_wal_senders" set to 10  
NOTICE: checking for available wal senders on the source node (2 required)  
INFO: sufficient wal senders available on the source node  
DETAIL: 2 required, 10 available  
NOTICE: checking replication connections can be made to the source server (2 required)  
INFO: required number of replication connections could be made to the source server  
DETAIL: 2 replication connections required  
WARNING: data checksums are not enabled and "wal_log_hints" is "off"  
DETAIL: pg_rewind requires "wal_log_hints" to be enabled  
NOTICE: standby will attach to upstream node 1  
HINT: consider using the -c/--fail-over-checkpoint option  
INFO: would execute:  
/usr/pgsql-15/bin/pg_basebackup -i "repmgr base backup" -D /var/lib/pgsql/15/data -h 192.168.187.134 -p 5432 -U repmgr -X stream  
INFO: all prerequisites for "standby clone" are met
```

*Figure 7.15: repmgr: dry-run on secondary node*

6. Referring to *Figure 7.15*, the **--dry-run** output looks correct. Run the **repmgr** standby clone command on standby node again without the **--dry-run** flag to clone the primary node to the standby node.

```
# Execute without dry-run on the standby node
```

```
/usr/pgsql-15/bin/repmgr -h 192.168.187.134 -U repmgr -d repmgr -f  
/etc/repmgr/15/repmgr.conf -D /var/lib/pgsql/15/data standby clone
```

As shown in *Figure 7.16*, primary node successfully cloned to standby node:

```
# Output of Master node cloning on the standby node
WARNING: following problems with command line parameters detected:
-D/-pgdata will be ignored if a repmgr configuration file is provided
NOTICE: destination directory "/var/lib/pgsql/15/data" provided
INFO: connecting to source node
DETAIL: connection string is: host=192.168.187.134 user=repmgr dbname=repmgr
DETAIL: current installation size is 39 MB
INFO: replication slot usage not requested; no replication slot will be set up for this standby
NOTICE: checking for available walsenders on the source node (2 required)
NOTICE: checking replication connections can be made to the source server (2 required)
WARNING: data checksums are not enabled and "wal_log_hints" is "off"
DETAIL: pg_rewind requires "wal_log_hints" to be enabled
INFO: checking and correcting permissions on existing directory "/var/lib/pgsql/15/data"
NOTICE: starting backup (using pg_basebackup)...
HINT: this may take some time; consider using the -c/--fast-checkpoint option
INFO: executing:
/usr/pgsql-15/bin/pg_basebackup -l "repmgr base backup" -D /var/lib/pgsql/15/data -h 192.168.187.134 -p 5432 -U repmgr -X stream
NOTICE: standby clone (using pg_basebackup) complete
NOTICE: you can now start your PostgreSQL server
HINT: for example: pg_ctl -D /var/lib/pgsql/15/data start
HINT: after starting the server, you need to register this standby with "repmgr standby register"
```

**Figure 7.16:** repmgr: Cloning on standby node

7. Run the **repmgr** standby register command on the standby node as the postgres superuser to register it with **repmgr**.

*# Register Standby node*

```
/usr/pgsql-15/bin/repmgr -f /etc/repmgr/15/repmgr.conf standby register
```

8. Verify the registered nodes in a PostgreSQL 15 cluster using **repmgr** by executing the following SQL script on primary node.

*# Verify the registered node*

```
SELECT node_id, node_name, type, conninfo, repluser, config_file FROM  
repmgr.nodes;
```

As shown in [Figure 7.17](#), this will return a list of nodes that are currently registered with **repmgr**, including their ID, name, type, status, and location and the status of the cluster. Please refer to the following figure:

Registered node						
4	SELECT node_id, node_name, type, conninfo, repluser, config_file FROM repmgr.nodes;					
Data Output		Messages		Notifications		
	node_id [PK] integer	node_name text	type text	conninfo text	repluser character varying (63)	config_file text
1	1	node_master	primary	host=192.168.187.134 dbname=repmgr port=5432 user=repmgr password=password connect_timeout=2	repmgr	/etc/repmgr/15/repmgr.conf
2	2	node_slave	standby	host=192.168.187.133 dbname=repmgr port=5432 user=repmgr password=password connect_timeout=2	repmgr	/etc/repmgr/15/repmgr.conf

## Repmgr cluster status

```
# repmgr cluster status output
ID | Name | Role | Status | Upstream | Location | Priority | Timeline | Connection string
1  | node_master | primary | * running | default | 100 | 2 | host=192.168.187.134 dbname=repmgr port=5432 user=repmgr password=password
2  | node_slave | standby | running | node_master | default | 100 | 2 | host=192.168.187.133 dbname=repmgr port=5432 user=repmgr password=password
```

**Figure 7.17:** `repmgr`: Verify registered node and cluster status

9. Start the **repmgrd** daemon, execute the following command on the primary and standby node:

# start the repmgrd daemon on primary and Standby  
#

```
/usr/pgsql-15/bin/repmgrd -f /etc/repmgr/15/repmgr.conf
```

10. Once the standby node is registered with **repmgr** and cloned from the primary node, it should automatically begin replicating data from the primary node. Let us test replication by creating sample data on the primary node and verifying that it appears on the standby node:

- Connect to the primary node using **psql** and create a sample database and table.

```
# Create sample database and table on primary node  
for replication test  
  
psql -h <primary_node_ip> -U postgres  
CREATE DATABASE replica_db;  
\c replica_db  
CREATE TABLE replicat_table (id serial primary key, name varchar(50));  
INSERT INTO replicat_table (name) VALUES ('test');
```

- Connect to the standby node using **psql** and verify that the **replica\_db** database and **replica\_table** table were replicated from the primary node.

```
# verify replica_db database and replica_table  
table on the standby node  
  
psql -h <standby_node_ip> -U postgres  
\c replica_db  
SELECT * FROM replica_table;
```

This should return the row that was inserted into the **replica\_table** table on the primary node. By creating sample data on the primary node and verifying that it appears on the standby node, you can confirm that replication is functioning correctly and that the replication cluster with repmgr is set up properly.

## Switchover versus failover

Switchover and failover are two important techniques in PostgreSQL database administration that are used to maintain high availability and prevent downtime.

Switchover is a planned operation in which the standby node is promoted to primary while the current primary node is demoted to become the new standby node. Failover, on the other hand, is an unplanned operation in which the primary node fails, and the standby node automatically becomes the new primary node.

In our recipe, we have used **repmgr** for PostgreSQL to perform both switchover and failover.

The following *Table 7.6* summarizes the main differences between the two techniques in the context of our recipe:

<b>Switchover</b>	<b>Failover</b>
Intended to perform a planned switch of roles.	Intended to perform an unplanned failover.
Primary is healthy and the switch is initiated.	Primary has failed and the failover is initiated.
Can be performed with minimal to no downtime.	Can result in significant downtime and data loss.
Requires both the old primary and new primary.	Only requires the new primary.
Can involve re-joining the old primary as standby.	Does not involve re-joining the old primary.

**Table 7.6:** Switchover versus failover

## Recipe 53: Switchover with repmgr for PostgreSQL

Switchover is a process of gracefully promoting a standby node to a primary node in a PostgreSQL replication cluster, typically performed during planned maintenance or upgrades. repmgr is a tool that simplifies the switchover process by automating the steps required to promote a standby node to a primary node.

In this recipe, we will be getting through the detailed steps to switchover PostgreSQL database with repmgr. The recipe steps to perform switchover PostgreSQL database cluster is based on two-nodes cluster.

1. Check the status of the replication cluster by running the following command on the primary node:

```
# Verify the status of replication cluster on primary node
```

```
repmgr -f /etc/repmgr/15/repmgr.conf cluster show
```

Refer *Figure 7.18*, this command will display the status of the replication cluster, including the status of each node, their roles, and their priority.

```
# Verify the status of replication cluster on primary node
ID | Name      | Role   | Status    | Upstream | Location | Priority | Timeline | Connection string
--+---+-----+-----+-----+-----+-----+-----+-----+
1 | node_master | primary | * running |          | default | 100     | 2        | host=192.168.187.134 dbname=repmgr port=5432 user=repmgr password=password
2 | node_slave  | standby  | running   | node_master| default | 100     | 2        | host=192.168.187.133 dbname=repmgr port=5432 user=repmgr password=password
```

**Figure 7.18:** Switchover: Replication cluster status

2. Check that the standby node is ready to be promoted and ensure no pending archive ready file using the following command on primary node:

```
# Verify the status of replication cluster on primary node
```

```
repmgr -f /etc/repmgr/15/repmgr.conf node check --archive-ready
```

Before promoting the standby node, this command is executed on the primary node to check WAL which have not been archived.

3. Execute the dry run on the standby node before switchover the standby node (node2) to the primary node.

```
# Execute dry-run switchover on standby node
```

```
repmgr -f /etc/repmgr/15/repmgr.conf standby switchover --dry-run
```

4. Verify the output to ensure that the promotion in dry run would succeed.

**Note: Ensure that passwordless SSH is working between both the node before attempting to perform a switchover with repmgr.**

5. If the dry run is successful, switchover the standby node to the primary role by running the following command on the standby node:

```
# Execute switchover on standby node
```

```
repmgr -f /etc/repmgr/15/repmgr.conf standby switchover
```

Referring to [Figure 7.19](#), verify the output to ensure that the promotion would succeed:

```
postgres@pgmaster ~ $ /usr/pgsql-15/bin/repmgr -f /etc/repmgr/15/repmgr.conf standby switchover
NOTICE: executing switchover on node "node_slave" (ID: 2)
NOTICE: attempting to pause repmgrd on 2 nodes
NOTICE: local node "node_slave" (ID: 2) will be promoted to primary; current primary "node_master" (ID: 1) will be demoted to standby
NOTICE: stopping current primary node "node_master" (ID: 1)
NOTICE: issuing CHECKPOINT on node "node_master" (ID: 1)
DETAIL: executing server command "/usr/pgsql-15/bin/pg_ctl -D '/var/lib/pgsql/15/data' -W -m fast stop"
INFO: checking for primary shutdown; 1 of 60 attempts ("shutdown_check_timeout")
INFO: checking for primary shutdown; 2 of 60 attempts ("shutdown_check_timeout")
NOTICE: current primary has been cleanly shut down at location 1/1F000028
NOTICE: promoting standby to primary
DETAIL: promoting server "node_slave" (ID: 2) using pg_promote()
NOTICE: waiting up to 60 seconds (parameter "promote_check_timeout") for promotion to complete
NOTICE: STANDBY PROMOTE successful
DETAIL: server "node_slave" (ID: 2) was successfully promoted to primary
NOTICE: node "node_slave" (ID: 2) promoted to primary, node "node_master" (ID: 1) demoted to standby
NOTICE: switchover was successful
DETAIL: node "node_slave" is now primary and node "node_master" is attached as standby
NOTICE: STANDBY SWITCHOVER has completed successfully
```

**Figure 7.19:** Switchover – promoting primary

6. At this stage switchover was successful, check the new primary node's status by running the following command:

```
# New Primary node status
```

```
/usr/pgsql-15/bin/repmgr -f /etc/repmgr/15/repmgr.conf node status
```

7. Verify the output to ensure that the new primary have be assigned the role as primary by referring to the [Figure 7.20](#):

```
[postgres@pgmaster ~]$ /usr/pgsql-15/bin/repmgr -f /etc/repmgr/15/repmgr.conf node status
Node "node_slave":
  PostgreSQL version: 15.1
  Total data size: 47 MB
  Conninfo: host=192.168.187.133 dbname=repmgr port=5432 user=repmgr password=password connect_timeout=2
  Role: primary
  WAL archiving: enabled
  Archive command: test ! -f /arch_wal/%f && cp %p /arch_wal/%f
  WALS pending archiving: 0 pending files
  Replication connections: 1 (of maximal 10)
  Replication slots: 0 physical (of maximal 10; 0 missing)
  Replication lag: n/a
```

**Figure 7.20:** Switchover - node status

Performing a switchover with **repmgr** for PostgreSQL can be a straightforward process if the necessary steps are followed correctly. By following the steps outlined in the recipe above, we can ensure that our switchover is successful and that our standby node is promoted to the primary role.

## Recipe 54: Failover with repmgr for PostgreSQL

Failover is the process of promoting a standby node to a primary node in a PostgreSQL replication cluster when the current primary node fails or becomes unavailable. **repmgr** provides automated failover, which makes the process of promoting a standby to a primary node much faster and more reliable than manual failover.

In this recipe, we will be getting through the detailed steps to failover PostgreSQL database with **repmgr**. The recipe steps to perform failover PostgreSQL database cluster is based on two-nodes cluster.

Check the health of the primary node, before initiating the failover process, it is important to check the health of the primary node. This can be done by running the following command on the primary node:

```
# Check the Primary node health by executing on
primary/standby node
```

```
/usr/pgsql-15/bin/repmgr -f /etc/repmgr/15/repmgr.conf node check
```

Referring to [Figure 7.21](#), from above command execution, we can see that there is an error indicating that the connection to the database has failed. The error message suggests that the PostgreSQL server at IP address **192.168.187.133** (Primary node) and port **5432** may not be running or may not be accepting TCP/IP connections. Please refer to the following figure:

```
[postgres@pgmaster2 ~]$ /usr/pgsql-15/bin/repmgr -f /etc/repmgr/15/repmgr.conf node check
ERROR: connection to database failed
DETAIL:
connection to server at "192.168.187.133", port 5432 failed: Connection refused
Is the server running on that host and accepting TCP/IP connections?

DETAIL: attempted to connect using:
user=repmgr password=password connect_timeout=2 dbname=repmgr host=192.168.187.133 port=5432 fallback_application_name=repmgr options=-csearch_path=
```

**Figure 7.21:** Failover – node status on primary

- As the primary node is down or unavailable, the next step is to promote the standby node to become the new primary node. This can be done by running the following command on the standby node.

*# Execute switchover on standby node*

```
repmgr -f /etc/repmgr/15/repmgr.conf standby promote
```

- Referring to [Figure 7.22](#), the output of the command shows that the standby node has been successfully promoted to a primary node. The **NOTICE: STANDBY PROMOTE** successful message indicates that the promotion process has completed successfully. Please refer to the following figure:

```
[postgres@pgmaster2 ~]$ /usr/pgsql-15/bin/repmgr -f /etc/repmgr/15/repmgr.conf standby promote
NOTICE: promoting standby to primary
DETAIL: promoting server "node_master" (ID: 1) using pg_promote()
NOTICE: waiting up to 60 seconds (parameter "promote_check_timeout") for promotion to complete
NOTICE: STANDBY PROMOTE successful
DETAIL: server "node_master" (ID: 1) was successfully promoted to primary
```

**Figure 7.22:** Failover – Promote standby to primary

- After completing the failover process, it is important to verify that the new primary node is functioning correctly. This can be done by running some basic SQL queries on the new primary node to ensure that data is being replicated correctly.

In conclusion, we have successfully performed a failover with **repmgr** for our PostgreSQL database. It is important to note that a failover should be executed carefully and only, when necessary, as it involves downtime and potential data loss.

## Understanding proxy and load balancing

Proxy and load balancing are two essential components of modern application architectures that can help improve the performance, scalability, and availability of web applications.

A proxy server is an intermediary server that sits between clients and servers, acting as a gateway or a filter for client requests. It can provide many benefits, such as load balancing, caching, security, and content

filtering. A proxy server can distribute client requests to multiple backend servers, thus improving performance and availability by reducing the load on any one server.

Load balancing, on the other hand, is the process of distributing network traffic across multiple servers to ensure that no single server is overwhelmed with traffic. A load balancer acts as a reverse proxy and distributes client requests to multiple backend servers based on various algorithms such as round-robin, least connections, IP hash, and so on. Load balancing can also provide high availability and fault tolerance, ensuring that if one server fails, traffic is automatically routed to other available servers.

## **Recipe 55: Deploying PostgreSQL Automatic Failover for HA solution**

PostgreSQL Automatic Failover is a process that ensures high availability of PostgreSQL databases by automatically promoting a standby server to become the new primary server when the primary server fails or becomes unavailable. This ensures that there is minimal downtime, and the database remains available to clients.

**pg\_auto\_failover** is a tool that simplifies the setup and management of PostgreSQL Automatic Failover. It monitors the PostgreSQL instances and automates the failover process when necessary.

Following is some of the key options provided by **pg\_auto\_failover**:

Options	Description
Monitor PostgreSQL instances	Continuously monitors the PostgreSQL instances to ensure they are available and synchronized.
Automatic failover	When the primary server fails, <b>pg_auto_failover</b> promotes the standby server to become the new primary server automatically.
Multiple standby servers	Manage multiple standby servers to ensure high availability and fault tolerance.
Replication quorum	Allows to set a replication quorum to determine how many standby servers must be available before automatic failover can occur.
Manual failover	Can manually trigger a failover process using the <b>pg_autoctl</b> command-line interface.
Customizable configuration	Allows to customize the configuration to meet your specific needs, including setting up custom scripts for monitoring and managing the PostgreSQL instances.

**Table 7.7:** pg\_auto\_failover configuration options

In this recipe we will guide through the detailed steps to configure the automatic failover with **pg\_auto\_failover**.

1. First, install and configure PostgreSQL on each servers and ensure to install the same version of PostgreSQL on both the servers. Refer to the recipe *Working with Installation from Binaries* from [Chapter 1, Up and running with PostgreSQL 15](#) for instructions on how to setup PostgreSQL instance.
2. The next step is to configure streaming replication between the primary and standby servers. To set up streaming replication, edit the **postgresql.conf** file on the primary server and set the following parameters:

```
# WAL configuration on primary server
```

```
wal_level = replica
```

```
max_wal_senders = 10
```

```
archive_mode = on
```

```
archive_command = 'cp %p /path/to/archive/%f'
```

Also, edit the **pg\_hba.conf** file on the primary server and add the following line to allow replication connections from the standby server:

```
# Update the pg_hba.conf on primary server with  
standby server ip
```

```
host replication all standby_ip_address/32 md5
```

3. Then, create a replication user on the primary server using the following command:

```
# Create replication user on primary server
```

```
CREATE USER replicapg REPLICATION LOGIN PASSWORD 'password';
```

4. Create a backup of the primary server, run the following command on the standby server to take a full backup:

```
# Create a backup of the standby server
```

```
pg_basebackup -h <primary_serv_ip> -U replicapg -p 5432 -D
```

```
/PG_DATAPATH/ -Fp -Xs -P -R
```

5. This command connects to the primary server using the replicapg user, creates a backup in the specified directory, and streams the WAL files to the backup directory.

If we have downloaded the **pg\_auto\_failover packages** from the download site. If not, then download the latest package of **pg\_auto\_failover** from website at <https://download.postgresql.org/pub/repos/yum/15/> and continue

with the installation of **pg\_auto\_failover** package by using the following steps:

1. At first need to have below package downloaded on all three server:

```
# pg_auto_failover downloaded package  
pg_auto_failover_15-2.0-1.rhel9.x86_64.rpm  
pg_auto_failover_15-llvmjit-2.0-1.rhel9.x86_64.rpm
```

2. Next, install the package from **yum** utility on all three server using below command:

```
# Install pg_auto_failover package  
yum localinstall pg_auto_failover_15-llvmjit-2.0-1.rhel9.x86_64.rpm  
pg_auto_failover_15-2.0-1.rhel9.x86_64.rpm -y
```

3. Verify the **pg\_auto\_failover** installed version on all three server.

```
# Verify the installed pg_auto_failover version  
/usr/pgsql-15/bin/pg_autoctl --version
```

4. Create a directory for **pg\_auto\_failover** monitoring data with the ownership of the directory to **postgres** user and group on the monitoring server.

```
# Create a directory for  
mkdir -p /var/lib/pgsql/15/data/auto_monitor
```

5. Initialize **pg\_auto\_failover** on the monitor node on the monitoring server.

```
# Initialize pg_auto_failover  
/usr/pgsql-15/bin/pg_autoctl create monitor --hostname localhost --pgdata  
/var/lib/pgsql/15/data/auto_monitor --pgport 5888 --auth trust --no-ssl
```

6. This command creates a new **pg\_auto\_failover** monitor node running on the local machine with a new PostgreSQL instance on port 5888, no authentication needed, and without SSL encryption. It is worth noting that these options are not suitable for a production environment and should be changed accordingly for security reasons.

7. Start the **pg\_auto\_failover** monitor with the **pg\_autoctl** run command.

```
# Start the pg_auto_failover service  
/usr/pgsql-15/bin/pg_autoctl run &
```

8. Use the **pg\_autoctl** show **uri** command to generate a connection string for the **pg\_auto\_failover** monitor.

```
# generate a connection string for the pg_auto_failover
```

*monitor*

```
/usr/pgsql-15/bin/pg_autoctl show uri
```

Referring to *Figure 7.23*, this command should output a connection string that we can use to connect to the **pg\_auto\_failover** monitor:

```
[postgres@localhost ~]$ /usr/pgsql-15/bin/pg_autoctl show uri --pgdata /var/lib/pgsql/15/data/
  Type |    Name | Connection string
-----+-----+
  monitor | monitor | postgres://autoctl_node@localhost:5888/pg_auto_failover?sslmode=prefer
  formation | default |
```

*Figure 7.23: pg\_auto\_failover - Verify connection string*

- Once the **pg\_auto\_failover** monitor node is created on the monitoring server, we can check its status by running.

*# Verify the node status*

```
/usr/pgsql-15/bin/pg_autoctl show state --pgdata
/var/lib/pgsql/15/data/
```

Referring to *Figure 7.24*, the output of the above command execution returns that none of the node present in the cluster.

```
[postgres@localhost ~]$ /usr/pgsql-15/bin/pg_autoctl show state --pgdata /var/lib/pgsql/15/data/
  Name | Node | Host:Port |     LSN |   Reachable |      Current State | Assigned State
-----+-----+-----+-----+-----+-----+

```

*Figure 7.24: pg\_auto\_failover - Monitoring node status*

- The output is empty because there are no PostgreSQL instances being monitored by the **pg\_auto\_failover** monitor node. The output of the command will list all the PostgreSQL instances that are being monitored by the **pg\_auto\_failover** monitor node, along with their respective states as reported and assigned by the monitor.
- Configure the primary database with the **pg\_auto\_failover** configuration using the following command:

*# Configure primary database*

```
/usr/pgsql-15/bin/pg_autoctl create postgres --hostname 192.168.187.134 --
pgdata /var/lib/pgsql/15/data/auto_monitor --pgport 5433 --auth trust --no-ssl --
username postgres --dbname dbmon --monitor
'postgres://autoctl_node@192.168.187.134:5888/pg_auto_failover?
sslmode=prefer'
```

- Configure the standby database with the **pg\_auto\_failover** configuration using the following command:

```
# Configure primary database
/usr/pgsql-15/bin/pg_autoctl create postgres --hostname 192.168.187.134 --
pgport 5434 --pgdata /var/lib/pgsql/15/data/auto_monitor --auth trust --no-ssl --
monitor 'postgres://autoctl_node@192.168.187.134:5888/pg_auto_failover?
sslmode=prefer' --run
```

13. Confirm synchronization of the previously added secondary node with the primary node by executing the following command:

```
# Verify the added node status
/usr/pgsql-15/bin/pg_autoctl show state --pgdata
/var/lib/pgsql/15/data/
```

Referring to [Figure 7.25](#), the output of the above command execution returns that secondary node successfully added and is in sync with the primary node:

[postgres@localhost ~]\$ /usr/pgsql-15/bin/pg_autoctl show state --pgdata /var/lib/pgsql/15/data/						
Name	Node	Host:Port	LSN	Reachable	Current State	Assigned State
node_1	1	192.168.187.134:5433	0/F5000060	yes	primary	primary
node_2	2	192.168.187.133:5434	0/F5000060	yes	Secondary	Secondary

**Figure 7.25:** pg\_auto\_failover - node list

14. Let us initiate the graceful switchover by running the following command on standby node:

```
# Graceful switchover
/usr/pgsql-15/bin/pg_autoctl perform switchover
```

This will initiate the switchover process, promoting the standby node to become the new primary node.

15. Verify the status of the switchover by running the following command on the new primary:

```
# Verify the switchover status on the new primary node
/usr/pgsql-15/bin/pg_autoctl show state --pgdata
/var/lib/pgsql/15/data/
```

Referring to [Figure 7.26](#), the output of the above command execution returns that the standby node successfully switchover as the new primary:

[postgres@localhost ~]\$ /usr/pgsql-15/bin/pg_autoctl show state --pgdata /var/lib/pgsql/15/data/						
Name	Node	Host:Port	LSN	Reachable	Current State	Assigned State
node_1	1	192.168.187.134:5433	0/F701BF00	yes	Secondary	Secondary
node_2	2	192.168.187.133:5434	0/F701BF00	yes	primary	primary

**Figure 7.26:** pg\_auto\_failover - node switchover status

Overall, **pg\_auto\_failover** simplifies the process of setting up and managing PostgreSQL Automatic Failover, providing a reliable and robust solution for high availability and fault tolerance.

## Recipe 56: Using HAProxy for HA solution

**High availability (HA)** solutions are important for any production-grade application that requires constant uptime. HAProxy and PgBouncer are two popular tools used for achieving HA in PostgreSQL.

HAProxy is a free and open-source load balancing software that can be used to distribute the incoming traffic across multiple PostgreSQL database servers. This ensures that the load is evenly distributed and no single database server is overburdened. HAProxy also provides failover support, which means that if one server fails, it can redirect the traffic to another available server.

This recipe will guide through the process of setting up HA with HAProxy in PostgreSQL 15. Before starting the recipe, make sure to have a basic understanding of PostgreSQL, HAProxy, and PgBouncer, and that we have already installed PostgreSQL 15 and HAProxy on our system. Additionally, it is recommended to have a backup of your PostgreSQL database before making any changes to your setup.

Our recipe setup for HA with HAProxy and PgBouncer in PostgreSQL 15 consists of the following components:

- Two PostgreSQL 15 servers (primary and standby server)
- One dedicated HAProxy server (act as the load balancer)

To establish a robust PostgreSQL setup with high availability, follow these steps:

1. First, install and configure PostgreSQL on each server and ensure to install the same version of PostgreSQL on both the servers. Refer to the recipe *Working with Installation from Binaries* from [Chapter 1](#), *Up and running with PostgreSQL 15* for instructions on how to setup PostgreSQL instance.
2. Configure a primary-standby replication between both the PostgreSQL node. Refer to the recipe *Scaling Postgres with Primary-Standby Replication* from [Chapter 7](#), *Replication and High Availability* for instructions on how to setup PostgreSQL primary-standby replication. At this stage replication is configured, Verify the replication status between the primary and standby server in PostgreSQL 15 before configuring HAProxy using following steps:

- a. Connect to the PostgreSQL instance on the primary server using the **psql** command-line tool and run the following SQL command to check the replication status:

```
# Verify the status of replication
```

```
select pid,username,application_name,state,sync_state,sent_lsn,reply_time
FROM pg_stat_replication;
```

Referring to *Figure 7.27*, this command will return a list of replication connections, including the status of each connection. If the replication is working correctly, we should see one row for each replication connection, with the state column set to streaming. Please refer to the following figure:

3 SELECT pid,username,application_name,state,sync_state,sent_lsn,reply_time FROM pg_stat_replication;						
	pid	username	application_name	state	sync_state	sent_lsn
1	4344	replicapg	walreceiver	streaming	async	0/1001DE90 2023-03-31 18:52:08.469263+08

**Figure 7.27:** HAProxy - Replication connection list

- b. Run the following SQL command to check the last transaction received by the standby server:

```
# Verify the status of replication
```

```
select pg_last_wal_receive_lsn;
```

Referring to *Figure 7.28*, this command will return the location in the WAL where the last transaction was received by the standby server. The output shows that the standby server has received the WAL up to the same location as reported by the primary servers **sent\_lsn** value (**0/1001DE90**). This indicates that the replication stream is working correctly, and the standby server is up to date with the primary server. Please refer to the following figure:

3 SELECT pg_last_wal_receive_lsn();	
	pg_last_wal_receive_lsn
1	0/1001DE90

**Figure 7.28:** HAProxy - last received WAL on standby

As we have verified that the replication is working, now we can proceed with configuring HAProxy for high availability:

## 1. Install and configure HAProxy on a separate dedicated servers:

- Install the HAProxy package by running the following command on a dedicated proxy node:

```
# Install HAProxy package on a dedicated proxy node
```

```
yum install haproxy -y
```

- Once installed, edit the HAProxy configuration file at **/etc/haproxy/haproxy.cfg** to define the backend PostgreSQL servers and the load balancing algorithm. Here is an example configuration based on our recipe setup:

```
# Update the configuration in /etc/haproxy/haproxy.cfg
```

```
file
```

```
global
```

```
log 127.0.0.1 local2
```

```
chroot /var/lib/haproxy
```

```
pidfile /var/run/haproxy.pid
```

```
maxconn 1000
```

```
user haproxy
```

```
group haproxy
```

```
daemon
```

```
defaults
```

```
mode tcp
```

```
log global
```

```
retries 3
```

```
timeout connect 10s
```

```
timeout client 1m
```

```
timeout server 1m
```

```
timeout check 10s
```

```
listen stats
```

```
mode tcp
```

```
bind *:5000
```

```
stats enable
```

```
stats uri /
```

```
listen postgresql_primary
```

```
bind *:5432
```

```

mode tcp
option tcp-check
default_backend postgresql_secondary
listen postgresql_secondary
bind *:5432
mode tcp
balance roundrobin
default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
server pg_pri 192.168.187.133:5432 check
server pg_sec 192.168.187.134:5432 check

```

In this configuration, we first define some global settings for HAProxy, including logging and user/group settings. We also define some default settings for our TCP mode.

Next, we define a frontend called **postgresql\_primary** that listens on port 5432 and specifies that the mode is TCP. We also define a backend called **postgresql\_secondary** that uses the round-robin algorithm to distribute traffic across two PostgreSQL servers: **pg\_pri** and **pg\_sec**. The check directive is used to verify the health of each server. Note that we are using the option `tcp-check` directive to perform a TCP check on the servers, and we are expecting the string **Postgres** in the server response:

- Once the configuration file has been updated, we can restart the HAProxy service using the following command:

*# Restart haproxy service*

```
systemctl restart haproxy
```

Now the HAProxy node will balance traffic across the two PostgreSQL nodes based on the round-robin algorithm.

**Note: Overall, this configuration provides a simple and effective way to provide high availability for a PostgreSQL database cluster using HAProxy. This is a basic configuration and may not be suitable for all production environments. Additional configuration may be necessary to handle failover and backups, for example. It is important to thoroughly test any HAProxy configuration before deploying it in a production environment.**

Status of HAProxy can also be viewed from web browser, enter the IP address or hostname of the machine running HAProxy followed by port

5000 (we have defined our stats port as 5000) in the address bar. For example, if the IP address of your HAProxy server is 192.168.1.100, you would enter **<http://192.168.187.135:5000>**

On the stats page, you can view information such as the number of requests per second, the number of active connections, the response time, and the status of each server in the backend pool. You can also view more detailed information by clicking on the tabs at the top of the page.

## **Monitoring replication**

Monitoring replication in PostgreSQL 15 involves tracking and assessing the synchronization of data across multiple database instances, ensuring data consistency and availability. This process is crucial for maintaining high availability, disaster recovery, and load distribution. PostgreSQL 15 offers several tools and features to facilitate effective replication monitoring. Administrators can utilize built-in monitoring extensions such as **pg\_stat\_replication** and **pg\_stat\_wal\_receiver** to gather real-time information about the status of replication processes. Additionally, PostgreSQL enhances the logical replication mechanism, allowing for better monitoring and management of replication streams. This involves tracking replication lag, identifying potential issues, and implementing corrective measures promptly. By actively monitoring replication in PostgreSQL 15, administrators can ensure that data is replicated accurately, downtime is minimized, and the overall performance and reliability of the database system are optimized.

## **Conclusion**

Throughout the journey, we traversed diverse replication architectures, from the fundamental primary-standby setups to the intricacies of multi-master replication using Bucardo. The recipes provided a hands-on experience, showcasing the scaling of Postgres with primary-standby replication, setup of delay standby, and intricate operations like promoting a standby database to the primary role in PostgreSQL 15.

As we navigated the diverse toolkit, deploying PostgreSQL Automatic Failover with Pg\_auto\_failover and leveraging HAProxy for high availability added robust layers to our exploration. Each tool, repmgr, Bucardo, and Pg\_auto\_failover, demonstrated its unique strengths in enhancing PostgreSQL's replication and high availability capabilities.

In reflection, the wealth of insights gained from this chapter positions us at the forefront of crafting resilient PostgreSQL environments. While each tool exhibited noteworthy capabilities, repmgr stands out as a robust and user-friendly solution for achieving high availability and seamless replication in PostgreSQL. The chapter not only equips you with the knowledge to implement these strategies effectively but also instills the confidence to adapt and tailor them to the unique needs of your organization.

Turning our attention to the next chapter, we shift gears to explore the world of SQL in PostgreSQL 15. Our objective is to delve into advanced topics such as joins, subqueries, and indexing – all crucial for efficient data management in PostgreSQL. So, let us embark on the exciting journey of exploring the world of SQL on PostgreSQL 15!

# CHAPTER 8

# Leveraging SQL

## Introduction

In this chapter we will focus on exploring the capabilities of SQL for accessing and manipulating the data stored in PostgreSQL databases. This chapter assumes that the reader has some basic knowledge of PostgreSQL and aims to enhance that knowledge by introducing various advanced SQL concepts. By leveraging SQL, readers will be able to work more efficiently with relational data in PostgreSQL, which is a powerful and widely used open-source relational database management system.

In today's world, data is considered the new currency, and understanding how to access, manage, and manipulate data in a PostgreSQL database is becoming increasingly important for businesses and organizations worldwide.

Overall, this chapter will equip the readers with necessary skills and knowledge to confidently work with PostgreSQL databases using SQL. By leveraging these skills, readers will be able to develop sophisticated database applications and handle complex data with ease, ultimately leading to more successful and efficient data management practices.

## Structure

In this chapter, we will cover the following topics:

- Explore data access
- Working with tables and data
- Explore DDL, DML, TCL and DCL

## Objectives

The objective of this chapter is to provide a solid understanding of SQL and its various features and look at how it can be leveraged to work with PostgreSQL databases. We will begin by reviewing the fundamentals of SQL including syntax, data types, and functions. From there, we will explore more advanced topics such as joins, subqueries, and indexing, all of which are essential for efficient data management in PostgreSQL. Finally, we will conclude with practical examples of how to use SQL to work with PostgreSQL databases, giving you a clear understanding of how to apply these concepts in real-world scenarios.

By the end of this chapter, you will be equipped with the necessary skills to confidently work with relational data in PostgreSQL databases using SQL. Whether you're a database administrator, developer, or data analyst, this chapter will provide you with the tools and knowledge you need to make the most of PostgreSQL's powerful features. So, let us get started and begin exploring the world of SQL on PostgreSQL 15!

## Exploring data access

Data access refers to the process of retrieving, manipulating, and storing data in a **database management system (DBMS)**. In a DBMS, data is stored in tables, which are organized into schemas. A schema is a collection of related database objects, such as tables, views, and stored procedures.

PostgreSQL is a powerful open-source relational database management system that offers a wide range of data access methods for users. In PostgreSQL, there are several methods of data access available in modern DBMSs, including SQL, client libraries, APIs, command-line tools, **graphical user interfaces (GUIs)**, and extensions.

Here is the comparison of data access methods in PostgreSQL:

Data access method	Description
SQL	Standard method for querying, inserting, updating, and deleting data in a PostgreSQL database using a common language.
Client libraries	Software components that provide programmatic access to a PostgreSQL database using a programming language such as C++, Java, Python, or Ruby.
APIs	Standardized way of interacting with a PostgreSQL database across different programming languages and platforms.
Command-line tool	Programs that allow users to interact with a PostgreSQL database using a command-line interface.
Graphical user interfaces (GUIs)	Visual interface that allows users to interact with a PostgreSQL database.
Extensions	Built-in extensions that provide additional functionality to a PostgreSQL database.

**Table 8.1:** Data access method

## Working with tables and data

PostgreSQL is a powerful and feature-rich open-source **relational database management system (RDBMS)** that offers robust support for working with tables and managing data. In this topic, we will explore various aspects of working with tables and data in PostgreSQL, including table creation, data insertion, retrieval, modification, and deletion, as well as common table operations and best practices.

### Recipe 57: Tables and Data Operations

This recipe covers key aspects, including the creation of tables, efficient data insertion, retrieval, modification, and deletion techniques. Readers will explore common table operations, best practices, and gain valuable insights into maximizing the potential of PostgreSQL for seamless and efficient data handling.

## Table creation

Creating tables in PostgreSQL involves defining the table definition, which includes specifying the table name, column names, data types, and constraints. Let us consider an example scenario where we want to create a table to store information about customers in an e-commerce application. The table definition could look like this:

```
CREATE TABLE customers (
    customer_id INT GENERATED BY DEFAULT AS IDENTITY PRIMARY
    KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(15),
    address TEXT
);
```

Referring to *Figure 8.1*, we created a table called **customers** with columns for **customer\_id** (auto-incrementing using **GENERATED BY DEFAULT AS IDENTITY**), **first\_name**, **last\_name**, **email** (unique constraint), **phone**, and **address**. This utilizes the **GENERATED BY DEFAULT AS IDENTITY** syntax to achieve auto-incrementing primary key functionality. Please refer to the following figure:

10 select * from pg_catalog.pg_tables where schemaname='public' and tablename='customers';								
	schemaname name	tablename name	tableowner name	tablespace name	hasindexes boolean	hasrules boolean	hastriggers boolean	rowsecurity boolean
1	public	customers	postgres	[null]	true	false	false	false

**Figure 8.1:** Explore data access – List customer table

## Data insertion

Once the table is created, we can insert data into it. Let us consider an example where we want to insert a new customer

into the **customers** table.

```
INSERT INTO
    CUSTOMERS (first_name, last_name, email, phone, address)
VALUES
(
    'Bugs',
    'Bunny',
    'bugsbunny@pgexample.com',
    '000-111-1234',
    'XYZ Street, IN'
);
```

In this example, we use the **INSERT INTO** statement to insert a new row into the **customers** table with values for the different columns. The **VALUES** clause specifies the values to be inserted.

## Data retrieval

Retrieving data from tables is a common operation in database applications. Let us consider an example scenario where we want to retrieve all the customers from the **customers** table who have a specific last name.

```
SELECT
    customer_id, first_name, last_name, email
FROM customers
WHERE last_name = 'Bunny';
```

Referring to [\*Figure 8.2\*](#), the execution of above select statement retrieve data from the **customers** table. The **WHERE** clause is used to specify the condition for filtering the data, in this case, retrieving customers with the last name **Bunny**. Please refer to the following figure:

```

10  SELECT customer_id, first_name, last_name, email
11  FROM customers
12  WHERE last_name = 'Bunny';

```

Data Output Messages Notifications

	customer_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (100)
1	1	Bugs	Bunny	bugsbunny@pgexample.com

**Figure 8.2:** Data retrieval – Retrieve data from customer table

## Data modification

Modifying data in tables is another common operation in database applications. Let us consider an example scenario where we want to update the email address of a specific customer in the **customers** table.

# Update the email

```

UPDATE customers
SET email = 'bbunny@pgexample.com'
WHERE customer_id = 1;

```

In the above script, we use the **UPDATE** statement to modify the data in the **customers** table. By referring to the [Figure 8.3](#), with the execution of above **update** statement, the **SET** clause specifies the new value for the **email** column, and the **WHERE** clause is used to specify the condition for updating the data. In this case, updating the customer with **customer\_id 1**. Please refer to the following figure:

```

10  SELECT
11      customer_id, first_name, last_name, email
12  FROM customers
13  WHERE last_name = 'Bunny';

```

Data Output Messages Notifications

	customer_id [PK] integer	first_name character varying (50)	last_name character varying (50)	email character varying (100)
1	1	Bugs	Bunny	bbunny@pgexample.com

**Figure 8.3:** Data modification – Retrieve modified data from customer table

## Table operations

PostgreSQL database offers a rich set of table operations for managing tables and their data. Let us consider an example where we want to add a new column to the **customers** table to store the customer's country.

```
# Update the email
```

```
ALTER TABLE
```

```
customers
```

```
ADD COLUMN country VARCHAR(50);
```

In the above command, we use the **ALTER TABLE** statement to modify the structure of the **customers** table by adding a new column called **country** with a data type of **VARCHAR** and a maximum length of **50** characters. This is just one of the many table operations that can be performed in PostgreSQL, including renaming columns, changing data types, adding constraints, and more.

## Data integrity and transactions

PostgreSQL provides several mechanisms for ensuring data integrity and consistency, including primary keys, foreign keys, unique constraints, and transactions. Let us consider an example of adding a unique constraint to the **email** column.

```
# Update the email
```

```
ALTER TABLE
```

```
customers
```

```
ADD CONSTRAINT unique_email UNIQUE (email);
```

In the above example, we add a unique constraint to the **email** column of the **customers** table, ensuring that each email address is unique.

## Recipe 58: Querying PostgreSQL data

Querying PostgreSQL data is one of the most essential operations in any database management system. In this recipe, we will

explore steps for querying PostgreSQL data, along with an example.

Before we begin, let us first understand some key concepts related to querying PostgreSQL data. PostgreSQL supports a wide range of data types, including numeric, string, date/time, Boolean, and more. It also provides a rich set of SQL commands and functions that allow users to manipulate data in various ways. The **SELECT** statement is the primary command used for querying data from a PostgreSQL database.

Let us consider an example to understand how to query PostgreSQL data. Suppose we have a database that contains a table named **employees** with the following columns: **id**, **name**, **department**, **salary**, and **hire\_date**. We want to query data from this table to retrieve the names and salaries of all employees who were hired after a specific date and are part of a specific department by retrieving information from the **department** table by using following steps:

1. To query PostgreSQL data, the initial step is to connect to the PostgreSQL database using the terminal-based **psql prompt**. To connect to the PostgreSQL database using **psql**, we use the following command syntax:

```
# Update the email
```

```
psql -U <username> -d <database_name>
```

For example, to connect to a database named **mydb** as a user named **postgres**, we can use the following command:

```
# Update the email
```

```
psql -U postgres -d mydb
```

2. Write a **SELECT** statement to query data from a table. To retrieve data from the **employees** table, we can use the following **SELECT** statement:

```
# Update the email
```

```
SELECT name, salary FROM employees;
```

Referring to [Figure 8.4](#), this statement will retrieve the names and salaries of all employees in the **employees** table:

The screenshot shows the pgAdmin interface with a query window containing the following code:

```
10 select * from employees;
```

Below the code, there are tabs for "Data Output", "Messages", and "Notifications". Under "Data Output", there is a toolbar with icons for copy, paste, refresh, delete, save, download, and search. The main area displays the results of the query as a table:

	<b>id</b> [PK] integer	<b>name</b> character varying (50)	<b>department</b> character varying (50)	<b>salary</b> character varying (15)	<b>hire_date</b> date
1	1	Popeye	Finance	5500	2021-09-01
2	2	Pluto	Finance	6000	2021-12-01
3	3	Homer Simpson	Sales	7000	2022-03-01
4	4	Charlie Brown	Sales	5600	2022-09-01
5	5	Johnny Bravo	IT	8000	2023-01-02

**Figure 8.4:** Querying PG data - Employees table

- To filter the data based on specific criteria, we can use the **WHERE** clause. For example, to retrieve the names and salaries of all employees who were hired after January 1st, 2022, and are part of the **Sales** department, we can use the following **SELECT** statement:

# Update the email

```
SELECT name, salary FROM employees WHERE hire_date > '2022-01-01' AND department = 'Sales';
```

Referring to **Figure 8.5**, above statement will retrieve the names and salaries of all employees who meet the specified criteria:

The screenshot shows the pgAdmin interface with a query window containing the following code:

```
10 SELECT name, salary FROM employees WHERE hire_date > '2022-01-01' AND department = 'Sales';
```

Below the code, there are tabs for "Data Output", "Messages", and "Notifications". Under "Data Output", there is a toolbar with icons for copy, paste, refresh, delete, save, download, and search. The main area displays the results of the query as a table:

	<b>name</b> character varying (50)	<b>salary</b> character varying (15)
1	Homer Simpson	7000
2	Charlie Brown	5600

**Figure 8.5:** Querying PG Data - Data filter from employees table

- Use the **ORDER BY** clause to sort data in ascending or descending order. To sort the data in ascending or descending order based on a specific column, we can use the **ORDER BY** clause. For example, to retrieve the names and salaries of all employees who meet the specified criteria and sort the data in descending order based on the

salary column, we can use the following **SELECT** statement:

```
# Execute select query on employees table
```

```
SELECT name, salary FROM employees WHERE hire_date > '2022-01-01' AND department = 'Sales' ORDER BY salary DESC;
```

Referring to *Figure 8.6*, above statement will retrieve the names and salaries of all employees who meet the specified criteria and sort the data in descending order based on the salary column:

	<code>id</code>	<code>name</code>	<code>salary</code>
1	3	Homer Simpson	7000
2	4	Charlie Brown	5600

**Figure 8.6:** Querying PG data - Criteria based retrieval from *employees* table

5. Use the **LIMIT** clause to limit the number of rows returned. To limit the number of rows returned, we can use the **LIMIT** clause. For example, to retrieve the top five names and salaries of all employees who meet the specified criteria and sort the data in descending order based on the salary column, we can use the following **SELECT** statement:

```
# Select query with LIMIT clause
```

```
SELECT name, salary FROM employees WHERE hire_date > '2022-01-01' AND department = 'Sales' ORDER BY salary DESC LIMIT 5;
```

Referring to *Figure 8.7*, above statement will retrieve the top five names and salaries of all employees who meet the specified criteria and sort the data in descending order based on the salary column:

	<code>id</code>	<code>name</code>	<code>salary</code>
1	5	Johnny Bravo	8000

**Figure 8.7:** Querying PG data - Order based retrieval from *employees* table

In conclusion, querying PostgreSQL data is a fundamental task for anyone working with a PostgreSQL database. In this example, we have covered the essential steps for querying PostgreSQL data, including connecting to the database using **psql**, writing a **SELECT** statement to retrieve data, using the **WHERE** clause to filter data based on specific criteria, using the **ORDER BY** clause to sort data, and using the **LIMIT** clause to limit the number of rows returned. With these steps, you can query PostgreSQL data efficiently and effectively to get the information you need.

## **Recipe 59: Querying using shell script in PostgreSQL**

PostgreSQL is a powerful database management system that allows users to store, manage, and retrieve large amounts of data efficiently. In this recipe, we will explore how to run a query using a shell script in PostgreSQL.

Before we begin, let us first understand some key concepts related to running a query using a shell script in PostgreSQL. A shell script is a program written in a scripting language that is interpreted by a shell program. In PostgreSQL, the shell script can be used to automate tasks such as running a query on a database. The shell script can execute **psql** commands that connect to the database, run the query, and return the result.

Let us consider an example to understand how to run a query using a shell script in PostgreSQL. Suppose we have a database that contains a table named **sales** with the following columns: **id**, **product\_name**, **sale\_date**, and **amount**. We want to run a query that retrieves the total amount of sales for a specific product and saves it to a file:

1. Create a shell script for querying PosgreSQL database. To create a shell script, open a text editor and save the file with a **.sh** extension. For example, create a file named **sales\_total.sh**.
2. Connect to the PostgreSQL database using **psql**. In the shell script, we need to connect to the PostgreSQL database

using **psql**. To do this, we can use the following command syntax:

```
psql -U <username> -d <database_name> -c "<query>"
```

Replace **<username>** with your PostgreSQL username, **<database\_name>** with the name of the database you want to connect to, and **<query>** with the SQL query you want to execute. For example:

```
psql -U postgres -d mydb -c "SELECT SUM(amount) FROM sales  
WHERE product_name = 'Server_Hardware';"
```

Referring to *Figure 8.8*, With above command will connect to the **mydb** database as the **postgres** user and execute the SQL query that retrieves the total amount of sales for **Server\_hardware**:

```
[postgres@pgmaster2 ~]$ psql -U postgres -d mydb -c "SELECT SUM(amount) FROM sales WHERE product_name = 'Server_hardware';"  
sum  
-----  
1450000  
(1 row)
```

**Figure 8.8:** Shell script – Connection test

3. Run a query on the database. In the shell script, we can run the SQL query on the database using the **psql** command. To run the query, add the **psql** command to the shell script. For example:

```
psql -U postgres -d mydb -c "SELECT SUM(amount) FROM sales  
WHERE product_name = 'Server_hardware';" > sales_total.txt
```

This command will run the SQL query and save the result to a file named **sales\_total.txt**:

4. Parse and process the query results. In the shell script, we can parse and process the query results using standard Unix commands such as **awk** or **sed**. For example, to extract the sales total value from the result file, we can use the following command:

```
cat sales_total.txt | awk '{print $1}'
```

Referring to *Figure 8.9*, this command will extract the sales total value from the file and print it to the console:

```
[postgres@pgmaster2 ~]$ cat sales_total.txt | awk '{print $1}'  
sum  
-----  
1450000  
(1
```

**Figure 8.9:** shell script- Export data print

5. Exit **psql**. In the shell script, we need to exit **psql** after running the query. To do this, we can use the following command:

```
psql -U <username> -d <database_name> -c "\q"
```

Replace **<username>** with your PostgreSQL username and **<database\_name>** with the name of the database you want to exit from. For example:

```
psql -U postgres -d mydb -c "\q"
```

This command will exit from the **mydb** database as the **postgres** user.

Here is the complete shell script that retrieves the total amount of sales for **Server\_hardware** and saves it to a file:

```
#!/bin/bash  
psql -U postgres -d mydb -c "SELECT SUM(amount) FROM sales  
WHERE product_name = 'Product A';" > sales_total.txt  
cat sales_total.txt | awk '{print $1}'  
psql -U postgres -d mydb -c "\q"
```

Save this script as **sales\_total.sh** and run it from the terminal using the following command:

```
# Execute the Script
```

```
./sales_total.sh
```

Referring to [Figure 8.10](#), this will execute the script and print the total sales amount for **Server\_hardware** to the console. The result will also be saved to a file named **sales\_total.txt**:

```
[postgres@pgmaster2 ~]$ ./sales_total.sh  
sum  
-----  
1450000  
(1
```

**Figure 8.10:** shell script- execution

In this recipe, we learned how to run a query using a shell script in PostgreSQL. This can be useful for automating repetitive tasks, such as generating reports or processing data. By following the steps outlined in this recipe, you can create a shell script that connects to a PostgreSQL database, runs a SQL query, processes the results, and exits from the database.

## Exploring DML, DDL, TCL and DCL

Explore DML, DDL, TCL, and DCL is a comprehensive topic that covers the different types of SQL statements used in PostgreSQL for manipulating data, defining database objects, managing transactions, and controlling access to the database. It delves into the details of **data manipulation language (DML)** statements for retrieving, adding, modifying, and deleting data, **data definition language (DDL)** statements for defining, modifying, and deleting database objects, **transaction control language (TCL)** statements for managing transactions, and **data control language (DCL)** statements for managing permissions and access levels.

## Recipe 60: Database Query and Control Language

The recipe provides examples and scenarios to illustrate the usage of these statements in PostgreSQL, enabling readers to effectively work with tables, data, transactions, and access control in the database including examples of common scenarios.

### Data manipulation language

DML statements are used to manipulate the data stored in the database. The main DML statements in PostgreSQL are **SELECT**, **INSERT**, **UPDATE**, and **DELETE**:

- **SELECT:** The **SELECT** statement is used to retrieve data from one or more tables in the database. Let us consider an example of retrieving the names and ages of employees from the **employees** table.

```
SELECT name, age FROM employees;
```

- **INSERT:** The **INSERT** statement is used to add new rows of data into a table. Let us consider an example of inserting a new employee into the **employees** table.

```
# Execute insert query on employees table
```

```
INSERT INTO
```

```
employees (name, age, department, salary)
```

```
VALUES ('Sarah Johnson', 32, 'Finance', 55000.00);
```

In the above example, we insert a new row of data into the **employees** table with values for the **name**, **age**, **department**, and **salary** columns.

- **UPDATE:** The **UPDATE** statement is used to modify existing data in a table. Let us consider an example of updating the salary of an employee in the **employees** table.

```
# Execute update query on employees table
```

```
UPDATE employees
```

```
SET salary = 60000.00
```

```
WHERE name = 'John Doe';
```

In the above example, we update the **salary** column of the row in the **employees** table where the **name** column is equal to **John Doe**.

- **DELETE:** The **DELETE** statement is used to remove rows of data from a table. Let us consider an example of deleting an employee from the **employees** table.

```
# Execute delete query on employees table
```

```
DELETE FROM employees
```

```
WHERE name = 'Sarah Johnson';
```

In the above example, we delete the row from the **employees** table where the **name** column is equal to **Sarah Johnson**.

## Data definition language

DDL statements are used to define and manage the structure of database objects, such as tables, indexes, and views. The main

DDL statements in PostgreSQL are **CREATE**, **ALTER**, and **DROP**:

- **CREATE:** The **CREATE** statement is used to create new database objects, such as tables, indexes, and views. Let us consider an example of creating a new table to store information about products:

```
# Create products table
```

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    price NUMERIC(10,2) NOT NULL
);
```

In the above example, we create a table named **products** with columns **id**, **name**, and **price**, and their respective data types and constraints.

- **ALTER:** The **ALTER** statement is used to modify the structure of existing database objects, such as adding or dropping columns, modifying constraints, and changing data types. Let us consider an example of adding a new column to the **products** table:

```
# Execute alter query on employee table
```

```
ALTER TABLE
products
ADD COLUMN description TEXT;
```

- **DROP:** The **DROP** statement is used to remove existing database objects, such as tables, indexes, and views. Let us consider an example of dropping the **products** table:

```
# Drop products table
```

```
DROP TABLE products;
```

In the above example, we drop the **products** table from the database.

## Transaction control language

TCL statements are used to manage transactions in the database. Transactions are used to group one or more DML statements into a single unit of work that can be either committed or rolled back as a whole. The main TCL statements in PostgreSQL are **COMMIT**, **ROLLBACK**, and **SAVEPOINT**:

- **COMMIT:** The **COMMIT** statement is used to permanently save changes made during a transaction. Let us consider an example of committing a transaction:

```
BEGIN; -- start a transaction
```

```
UPDATE employees
```

```
    SET salary = salary + 1000
```

```
    WHERE department = 'IT';
```

```
COMMIT; -- commit the transaction and save the changes
```

In the above example, we start a transaction, update the salaries of employees in the **IT** department, and then commit the transaction to permanently save the changes.

- **ROLLBACK:** The **ROLLBACK** statement is used to undo changes made during a transaction and roll back the transaction to its starting point. Let us consider an example of rolling back a transaction:

```
BEGIN; -- start a transaction
```

```
UPDATE employees
```

```
    SET salary = salary + 1000
```

```
    WHERE department = 'Finance';
```

```
ROLLBACK; -- roll back the transaction and undo the  
changes
```

In the above example, we start a transaction, update the salaries of employees in the **Finance** department, and then roll back the transaction to undo the changes made during the transaction.

- **SAVEPOINT:** The **SAVEPOINT** statement is used to create a save point within a transaction, to which you can later roll back. Let us consider an example of using a savepoint:

```
BEGIN; -- start a transaction
```

```

    UPDATE employees
        SET salary = salary + 1000
    WHERE department = 'HR';
    SAVEPOINT my_savepoint; -- create a savepoint
    UPDATE employees
        SET salary = salary + 2000
    WHERE department = 'Finance';
    ROLLBACK TO my_savepoint; -- roll back to the savepoint and
    undo the changes made after the savepoint
    COMMIT; -- commit the transaction and save the changes

```

In the above example, we start a transaction, update the salaries of employees in the **HR** department, create a savepoint, update the salaries of employees in the **Finance** department, and then roll back to the savepoint to undo the changes made after the savepoint.

## Data control language

DCL statements are used to manage permissions and access control in the database. The main DCL statements in PostgreSQL are **GRANT**, **REVOKE**, and **ALTER DEFAULT PRIVILEGES**:

For an in-depth exploration of these critical aspects, refer [Chapter 9, Server Controls and Auditing with Recipe Database Privileges](#). In this upcoming chapter, we will delve into the intricate details of data control language, offering comprehensive insights into the effective management of permissions and privileges.

Understanding these different types of SQL statements and their usage in PostgreSQL is crucial for effectively working with tables, data, transactions, and access control in the database. In the following recipe, we will delve into each type of statement in detail, providing examples and scenarios for better understanding.

## Recipe 61: DDL adaptation

Data definition language in PostgreSQL refers to the set of SQL commands that allow you to manage the structure of your database, including creating, altering, and dropping tables, indexes, and other database objects. In PostgreSQL, there have been updates and improvements related to DDL operations. This recipe explores DDL adaptation in PostgreSQL and provides a practical scenario to illustrate these enhancements.

Prerequisite:

- PostgreSQL 15 (installed and running)
- A text editor or **integrated development environment (IDE)**

Imagine you are working on a project to build a web application that tracks inventory for a retail business. You have already created a database called **inventory\_db** and defined several tables, including **products** and **suppliers**. Now, you need to make some adjustments to the database schema using the new DDL features in PostgreSQL:

## 1. Add a new column to the **products** Table:

You want to add a new column, **discount\_percentage**, to the **products** table to store information about discounts on products. In PostgreSQL, you can use the **ADD COLUMN** option with the **ALTER TABLE** statement to add a new column to an existing table.

```
# Connect to the database
```

```
\c inventory_db
```

```
# Add the new column
```

```
ALTER TABLE products
```

```
ADD COLUMN discount_percentage decimal(5, 2);
```

## 2. Rename a table:

You realize that the table name **suppliers** is too generic and should be more specific. PostgreSQL introduces the **RENAME** option for the **ALTER TABLE** statement to rename tables easily.

```
# Rename the "suppliers" table to
```

```
"product_suppliers"
```

```
ALTER TABLE suppliers RENAME TO product_suppliers;
```

### 3. Change the data type of a column:

The **discount\_percentage** column was mistakenly created as a decimal. You want to change its data type to an integer. In PostgreSQL, you can use the **SET DATA TYPE** option with the **ALTER TABLE** statement to modify the data type of a column.

```
# Change the data type of the  
"discount_percentage" column to integer
```

```
ALTER TABLE products
```

```
ALTER COLUMN discount_percentage SET DATA TYPE integer;
```

An alternative way to change the data type of a column in PostgreSQL is by utilizing the **USING** clause within the **ALTER TABLE** statement. This clause allows you to specify a conversion expression for more complex type transformations.

Let us say you have a table named sales with a column named **order\_date** that was mistakenly created as text instead of a date. You want to convert this **order\_date** column from text to a date format. Here is how you could do that using the **USING** clause:

```
# Sample table structure and data
```

```
CREATE TABLE sales (
```

```
    order_id serial PRIMARY KEY,
```

```
    order_date text
```

```
);
```

```
INSERT INTO sales (order_date) VALUES
```

```
('2023-01-15'),
```

```
('2023-02-20'),
```

```
('2023-03-25');
```

In the example below, we demonstrate the use of **USING** to convert a decimal column (**discount\_percentage**) to integer:

```
# Change the data type of the
```

```
"discount_percentage" column to integer
```

```
ALTER TABLE sales  
    ALTER COLUMN order_date TYPE DATE  
        USING order_date::date;
```

In this command, **order\_date::date** is the conversion expression applied to each value in the **order\_date** column to cast it into the desired date format. This method is particularly useful for handling more intricate data type conversions.

If there are any values in the **order\_date** column that cannot be successfully cast to a **DATE**, the operation will fail, and PostgreSQL will raise an error. In such cases, you may need to address or clean the data before attempting the data type conversion.

#### 4. Drop a column:

You decide that the **product\_suppliers** table no longer needs the **address** column. You can use the **DROP COLUMN** option with the **ALTER TABLE** statement to remove the column.

```
# Drop the "address" column from the  
"product_suppliers" table
```

```
ALTER TABLE product_suppliers  
    DROP COLUMN address;
```

#### 5. Manage constraints:

PostgreSQL also provides improved constraint management. Let us say you want to add a unique constraint to the **product\_name** column in the **products** table to ensure that product names are unique.

```
# Add a unique constraint to the "product_name"  
column in the "products" table
```

```
ALTER TABLE products  
    ADD CONSTRAINT unique_product_name UNIQUE (product_name);
```

PostgreSQL enhances DDL operations, making it easier to adapt your database schema to changing requirements. The new

features allow you to add and modify columns, rename tables, change data types, and manage constraints with greater flexibility. This recipe demonstrated how to use these DDL features in a practical scenario. These capabilities will help you maintain a well-structured database as your application evolves.

## Recipe 62: Working with dataset export/import

Working with dataset export/import is an essential aspect of managing databases. In PostgreSQL, there are multiple methods to export and import datasets. In this recipe, we will cover two commonly used methods: using the **pg\_dump** and **pg\_restore** commands and using the **COPY** command.

Suppose you have a PostgreSQL database named **mydb** that contains a table named **sales**. You want to export the contents of the **sales** table to a dump/CSV file and import it into another PostgreSQL database named **newdb**.

### Method 1: Using pg\_dump and pg\_restore

1. Export the dataset. The **pg\_dump** command is used to export a PostgreSQL database or specific tables to a file. To export the **sales** table from the **mydb** database, run the following command:

```
# Execute pg_dump to export dataset
```

```
pg_dump -U postgres -Fc -t sales mydb > sales.dump
```

This command will export the **sales** table from the **mydb** database and save it to a file named **sales.dump**:

```
[postgres@pgmaster2 ~]$ pg_dump -U postgres -v -Fc -t sales mydb > sales.dump
pg_dump: last built-in OID is 16383
pg_dump: reading extensions
pg_dump: identifying extension members
pg_dump: reading schemas
pg_dump: reading user-defined tables
pg_dump: reading user-defined functions
pg_dump: reading user-defined types
pg_dump: reading procedural languages
pg_dump: reading user-defined aggregate functions
pg_dump: reading user-defined operators
pg_dump: reading user-defined access methods
pg_dump: reading user-defined operator classes
pg_dump: reading user-defined operator families
pg_dump: reading user-defined text search parsers
pg_dump: reading user-defined text search templates
pg_dump: reading user-defined text search dictionaries
pg_dump: reading user-defined text search configurations
pg_dump: reading user-defined foreign-data wrappers
pg_dump: reading user-defined foreign servers
pg_dump: reading default privileges
pg_dump: reading user-defined collations
pg_dump: reading user-defined conversions
pg_dump: reading type casts
pg_dump: reading transforms
pg_dump: reading table inheritance information
pg_dump: reading event triggers
pg_dump: finding extension tables
pg_dump: finding inheritance relationships
pg_dump: reading column info for interesting tables
```

**Figure 8.11:** Pg\_dump - export table

2. Import the dataset. The **pg\_restore** command is used to restore a PostgreSQL database or specific tables from a file. To import the **sales** table into the **newdb** database, run the following command:

```
# Execute pg_restore to restore sales table and its
dataset
```

```
pg_restore -U postgres -t sales -d newdb sales.dump
```

This command will import the **sales** table from the **sales.dump** file into the **newdb** database.

```
[postgres@pgmaster2 ~]$ pg_restore -U postgres -v -t sales -d newdb sales.dump
pg_restore: connecting to database for restore
pg_restore: creating TABLE "public.sales"
pg_restore: processing data for table "public.sales"
[postgres@pgmaster2 ~]$
```

**Figure 8.12:** pg\_restore – restore table

## Method 2: Using the COPY command

1. Export the dataset. The **COPY** command is used to export the contents of a PostgreSQL table to a file. To export the **customers** table from the **mydb** database to a CSV file, run the following command:

*# Export the dataset to Sales.CSV*

```
psql -U postgres -d mydb -c "COPY customers TO 'customers.csv' CSV
HEADER;"
```

Referring to [Figure 8.13](#), the above command will export the **sales** table from the **mydb** database to a file named **sales.csv**.

```
[postgres@pgmaster2 ~]$ psql -U postgres -d mydb -c "\COPY sales TO 'sales.csv' CSV HEADER;"
COPY 9
[postgres@pgmaster2 ~]$
```

**Figure 8.13:** COPY option – Export table to CSV

2. Import the dataset. The **COPY** command can also be used to import data into a PostgreSQL table from a CSV file. To import the **sales** table from the **sales.csv** file into the **newdb** database, run the following command:

*# Import the dataset from Sales.CSV file*

```
psql -U postgres -a -d newdb -c "\COPY sales FROM 'sales.csv' CSV
HEADER;"
```

Referring to [Figure 8.14](#), the above command will import the **customers** table from the **customers.csv** file into the **newdb** database:

```
[postgres@pgmaster2 ~]$ psql -U postgres -a -d newdb -c "\COPY sales FROM 'sales.csv' CSV HEADER;
COPY sales FROM 'sales.csv' CSV HEADER;
COPY 9
[postgres@pgmaster2 ~]$
```

**Figure 8.14:** COPY option - Import table from CSV

In this recipe, we learned two commonly used methods for exporting and importing datasets in PostgreSQL. By using the **pg\_dump** and **pg\_restore** commands or the **COPY** command, you can easily export data from a PostgreSQL database to a file and import it into another PostgreSQL database. These methods can be useful for backing up databases, migrating data between environments, and sharing data with others.

## Recipe 63: Dataset load from spreadsheet/flat files

Loading datasets from spreadsheet or flat files is a common task in database management. In PostgreSQL, we can load datasets from files such as CSV, TSV, or Excel files using the **COPY** command. In this recipe, we will cover the steps to load a CSV file into a PostgreSQL table.

Suppose you have a CSV file named **employees.csv** that contains employee data, including columns such as **employee\_id**, **name**, **department**, and **salary**. We want to load this data into a PostgreSQL table named **employees**.

1. The initial steps in this recipe involve creating a table. Before you can load the data, you must create a table in your PostgreSQL database to store the data. To create the **employees** table, run the following command in **psql**:

```
# Create an Employees table
```

```
CREATE TABLE employees (
    employee_id INTEGER,
    name VARCHAR(50),
    department VARCHAR(50),
    salary INTEGER
);
```

This command will create a table named **employees** with four columns: **employee\_id**, **name**, **department**, and **salary**.

2. Load the dataset. To load the dataset from the **employees.csv** file into the **employees** table, run the

following command in **psql**:

```
# Login to psql command prompt.
```

```
psql -U postgres -d postgres
```

```
#Load the data-set
```

```
COPY employees FROM 'path/to/employees.csv' CSV HEADER;
```

Replace **path/to/employees.csv** with the actual file path to the **CSV** file. The **CSV** option specifies that the file is a comma-separated value file, and the **HEADER** option specifies that the first row of the **CSV** file contains the column headers.

3. Verify the data. To verify that the data has been loaded successfully, run the following command in **psql**:

```
# Verify the data on employees table
```

```
SELECT * FROM employees;
```

This command will display all the data in the **employees** table.

4. Clean up. After you have loaded the data, you can remove the **employees.csv** file if it is no longer needed.

```
# Clean-up of .csv file
```

```
rm path/to/employees.csv
```

In this recipe, we learned how to load a dataset from a CSV file into a PostgreSQL table using the **COPY** command. By following these steps, you can quickly load data from flat files into your PostgreSQL database.

## Recipe 64: Structuring query access with **psql**

PostgreSQL is a popular open-source relational database management system that supports a wide range of data types, indexes, and complex queries. One of the key features of PostgreSQL is its powerful command-line tool, **psql**, which allows users to interact with the database and perform various operations, including querying, modifying, and creating databases, tables, and indices. In this section, we will explore how to structure query access with **psql** in PostgreSQL.

Before we dive into the details, let us first understand what query access is and why it is essential. Query access refers to the ability to query data from a database. In PostgreSQL, query access is provided through the **SELECT** statement. When working with a database, it is often necessary to restrict the query access to specific users or roles. Restricting query access can help ensure the security and integrity of the data by preventing unauthorized users from accessing or modifying the data.

In PostgreSQL, query access can be restricted by defining a set of rules using the **GRANT** and **REVOKE** commands. These commands allow database administrators to grant or revoke permissions to users or roles for specific database objects, such as tables, views, and sequences. To structure query access with **psql**, we need to use these commands to define the necessary permissions.

Let us consider an example to understand how to structure query access with **psql** in PostgreSQL. Suppose we have a database that contains two tables: **employees** and **departments**. We want to restrict the query access to these tables so that only the HR department can query the **employees** table, and only the finance department can query the **departments** table.

To achieve this, we need to follow the below steps:

1. First, we need to create the **employees** and **departments** tables using the **CREATE TABLE** command. For example:

```
# Create employees Table
```

```
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    department VARCHAR(50)
);
```

```
# Create departments Table
```

```
CREATE TABLE departments (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    manager VARCHAR(50)
```

);

2. Next, we need to create two roles: **HR** and **Finance**. We can create roles using the **CREATE ROLE** command. For example:

```
# Create Roles
```

```
CREATE ROLE hr WITH LOGIN ENCRYPTED PASSWORD  
'Hr@1234';;  
CREATE ROLE fin WITH LOGIN ENCRYPTED PASSWORD  
'Fin@1234';;
```

3. Now, we need to grant the necessary permissions to the roles. We want to restrict the query access to the **employees** table to the **HR** role and the **departments** table to the finance role. We can do this using the **GRANT** command. For example:

```
# Grant roles
```

```
GRANT SELECT ON employees TO hr;  
GRANT SELECT ON departments TO fin;
```

4. Finally, we can verify the permissions by logging in as the **HR** role and trying to query the **departments** table. We should get an error message saying that the **HR** role does not have the necessary permissions. Similarly, we can verify the permissions for the finance role by trying to query the **employees** table.

To log in as a role using **psql**, we need to use the following command:

```
psql -U <role_name> -d <database_name>
```

For example, to log in as the **HR** role to the database **mydb**, we can use the following command:

```
psql -U hr -d mydb
```

In summary, structuring query access with **psql** in PostgreSQL involves defining a set of rules using the **GRANT** and **REVOKE** commands to grant or revoke permissions to users or roles for specific database objects. By restricting query access to specific users or roles, we can

ensure the security and integrity of the data in the database.

## Recipe 65: Exploring PostgreSQL join and subqueries

PostgreSQL provides powerful tools for combining data from multiple tables through joins and subqueries. Joins allow you to retrieve data from multiple tables based on a related column, while subqueries enable you to nest one query within another. In this recipe, we will delve into practical examples of using joins and subqueries in PostgreSQL.

Consider a scenario where you are managing a database for a bookstore. You have two tables: books and authors. The books table contains information about each book, including its title, ISBN, and the author's ID. The authors table contains details about each author, such as their name and ID. We will explore how to retrieve information that combines data from both tables.

### 1. Overview of the tables:

The **authors** table contains unique author details, including **author\_id** and **author\_name**. The **books** table stores book information with a title, ISBN, and a link to the author through the **author\_id** foreign key. These interconnected tables lay the groundwork for showcasing PostgreSQL's potent join and subquery functionalities.

```
# Create the authors table
```

```
CREATE TABLE authors (
    author_id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    author_name VARCHAR(100) NOT NULL
);
```

```
# Create the books table
```

```
CREATE TABLE books (
    book_id INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    isbn VARCHAR(13) NOT NULL,
    author_id INT REFERENCES authors(author_id)
```

```
);

# Insert sample data

INSERT INTO authors (author_name) VALUES
('Tom A'),
('Jerry A'),
('Lego A');

INSERT INTO books (title, isbn, author_id) VALUES
('Mastery SQL', '000-123456', 1),
('Database Essentials', '000-987654', 2),
('PGs Mastery', '000-555555', 3);
```

## 2. Using **INNER JOIN** to combine data:

Suppose you want to retrieve a list of books with their titles and the corresponding author names.

```
SELECT books.title, authors.author_name
FROM books
JOIN authors ON books.author_id = authors.author_id;
```

This query uses the **INNER JOIN** clause to combine data from the books and authors tables based on the common **author\_id** column.

## 3. Using **LEFT JOIN** for inclusive results:

If you want to retrieve all books, including those without an associated author (if any), you can use a **LEFT JOIN**.

```
SELECT books.title, authors.author_name
FROM books
LEFT JOIN authors ON books.author_id = authors.author_id;
```

## 4. Subquery for complex conditions:

Suppose you want to find authors who have written more than one book.

```
SELECT author_name
FROM authors
WHERE author_id IN (
    SELECT author_id
    FROM books
    GROUP BY author_id
```

```
HAVING COUNT(*) > 1  
);
```

In this example, the subquery retrieves author IDs from the books table, and the main query selects author names based on those IDs.

## 5. Correlated subquery:

Let us find books with titles longer than the average length.

```
SELECT title  
FROM books b  
WHERE LENGTH(title) > (  
    SELECT AVG(LENGTH(title))  
    FROM books  
);
```

Here, the subquery calculates the average length of titles, and the main query selects books with titles longer than this average.

In this recipe, we have explored the use of joins and subqueries in PostgreSQL. Joins are powerful for combining data from multiple tables based on related columns, while subqueries provide a way to nest queries for more complex conditions. These tools are fundamental in building sophisticated queries to extract meaningful insights from your PostgreSQL database.

## Recipe 66: Querying JSON data

**JavaScript Object Notation (JSON)** is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. JSON is commonly used in web applications and services for data exchange between systems.

PostgreSQL provides support for JSON data type, allowing you to store and manipulate JSON data in your database. JSON data can be stored in a column with the JSON data type. You can query JSON data in PostgreSQL using the built-in functions and

operators, which provide powerful tools for working with JSON data, making it a valuable addition to PostgreSQL's capabilities.

PostgreSQL supports both the **json** and **jsonb** data types for storing JSON data, and there are indeed important distinctions between the two. Here is the quick summary:

Feature	JSON	JSONB
Storage format	Plain text	Binary
Order of keys	Maintain orders	Does not maintain order
Indexing	Less efficient	Support efficient indexing
Query performance	Generally slower due to reparse each execution	Generally faster due to binary storage, avoiding the need for reparse with each execution.
Query flexibility	More flexible for simple cases	More efficient for complex cases
Storage overhead	Higher overhead due to text	Lower overhead due to binary
Typical use cases	Simple, human-readable data	Complex and performance-critical scenarios
Key considerations	Order of key is important	Performance is a priority

Choose between JSON and JSONB based on your specific requirements. If you need a lightweight, human-readable format with ordered keys, JSON might be sufficient. If you prioritize performance, efficient indexing, and have complex queries, JSONB is often the better choice.

In this recipe, we will explore the steps to query JSON data in PostgreSQL by creating a sample table with a JSON column and executing some sample queries.

Suppose you have a PostgreSQL table named **orders** that contains JSON data in a column named **items**. The **items** column contains JSON data that includes order details such as product name, quantity, price, and total price. You want to query this data to retrieve specific order details.

Before working with Postgres JSON queries, we need to create a table that includes a JSON column:

1. For this example, let us create a table called **employees** with columns for **id**, **name**, **address**, and **details**:

```
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    address TEXT NOT NULL,
    details JSONB
);
```

In this table, the **id** column is an auto-incrementing **primary key**, and the **name**, **address**, and **details** columns are of type **TEXT**, **TEXT**, and **JSON**, respectively.

2. Next, let us insert some sample data into the **employees** table.

```
INSERT INTO employees (name, address, details)
VALUES ('Homper Simpson', '111 St', '{"department": "IT", "salary": 50000}');
INSERT INTO employees (name, address, details)
VALUES ('Charlie Brown', '112 St', '{"department": "HR", "salary": 60000}');
```

In these **INSERT** statements, we insert two rows into the **employees** table. Each row contains values for the **name**, **address**, and **details** columns. The **details** column contains JSON data representing additional details about each employee.

3. **Querying JSON data:** To query JSON data, we can use the **->** operator to extract a specific JSON element from a JSON object or array. For example, to extract the **department** field from the **details** column, we can use the following query:

```
SELECT name, details->>'department' as department FROM
employees;
```

This query selects the **name** column and extracts the **department** field from the **details** column using the **->>**

operator. The `->>` operator returns the value of the JSON element as a string. Please refer to the following figure:

	name text	department text
1	Homper Simpson	IT
2	Charlie Brown	HR

**Figure 8.15:** JSON query - Querying JSONB data

4. **Filtering on JSON data:** We can also filter data based on JSON elements using the `->` operator. For example, to find all employees whose department is **IT**, we can use the following query:

```
# Execute select query on employees table
```

```
SELECT name, details->>'department' as department  
FROM employees  
WHERE details->>'department' = 'IT';
```

This query selects the name column and extracts the department field from the details column. The **WHERE** clause filters the results to only include employees whose department is **IT**.

	name text	department text
1	Homper Simpson	IT

**Figure 8.16:** JSON query - filtering JSONB data

5. **Updating JSON data:** We can update JSON data in a PostgreSQL table using the **jsonb\_set** function. This function takes three arguments: the original JSON value, a JSON path specifying the location to be updated, and the new JSON value to be inserted.

For example, to update the phone number of the first employee, you can use the following query:

```
# Execute update query on employees table
```

```
UPDATE employees
SET details = jsonb_set(details, '{employees,0,phone}', '"000-111-
1234")'
WHERE id = 1;
```

This will update the phone number of the first employee to **000-111-1234**.

6. **Deleting JSON data:** We can delete JSON data from a PostgreSQL table using the **jsonb\_set** function with a null value as the third argument. For example, to delete the email address of the second employee, you can use the following query:

*# Execute update query on employees table*

```
UPDATE employees
SET details = jsonb_set(details, '{employees,1,email}', null)
WHERE id = 1;
```

This will delete the email address of the second employee from the JSON data.

PostgreSQL's support for JSON data makes it easy to store and query JSON data in your database. With the **jsonb** data type and the various JSON functions and operators, you can work with JSON data as if it were native to the database.

## Recipe 67: Working with PostgreSQL CAST operator

The PostgreSQL **CAST** operator is used to convert data from one data type to another. It is a very powerful and useful operator that allows you to manipulate data in various ways. The **CAST** operator is especially useful when working with complex data types, such as arrays and JSON objects.

In this recipe, we will explore how to work with the PostgreSQL **CAST** operator. We will look at an example and see how to use the **CAST** operator to convert data from one data type to another.

Suppose you are working on a project that requires you to manipulate data from a PostgreSQL database. You need to convert the data from a string to a numeric data type so that you

can perform calculations on it. In this scenario, we will use the **CAST** operator to convert a string to a numeric data type.

Before working with the PostgreSQL **CAST** operator, make sure you are connected to your PostgreSQL database using your preferred client:

1. Create a table in your database that contains the string data that you want to convert. For example:

```
CREATE TABLE
```

```
string_data (
    id SERIAL PRIMARY KEY,
    string_value VARCHAR(20)
);
```

2. Insert some data into the table.

```
INSERT INTO string_data (string_value) VALUES ('123'), ('456'),
('789');
```

3. Check the data type of the **string\_value** column using the **pg\_typeof()** function.

```
SELECT pg_typeof(string_value) FROM string_data;
```

This should return character varying as given below:

	pg_typeof regtype	lock
1	character varying	
2	character varying	
3	character varying	

**Figure 8.17:** Cast operator - Varying character

4. Use the **CAST** operator to convert the data type of the **string\_value** column to numeric.

```
SELECT CAST(string_value AS numeric) FROM string_data;
```

This should return a result set that looks like this:

	string_value	numeric
1		123
2		456
3		789

**Figure 8.18:** Cast operator - String value

5. We can also use the **CAST** operator to convert the data type of a column in a **SELECT** statement.

```
# Execute select statement with CAST operator
```

```
SELECT id, CAST(string_value AS numeric) AS numeric_value FROM
string_data;
```

This should return a result set that looks like this:

	id [PK] integer	numeric_value	numeric
1	1	123	
2	2	456	
3	3	789	

**Figure 8.19:** Cast operator - result set

In this recipe, we have shown how to use the PostgreSQL **CAST** operator to convert data from one data type to another. We have provided an example scenario and demonstrated how to convert a string to a numeric data type using the **CAST** operator. The **CAST** operator is a powerful tool that can be used to manipulate data in many ways, and it is an essential part of any PostgreSQL developer's toolkit.

## Recipe 68: Working with database consistency and integrity

Database consistency and integrity are crucial for maintaining the accuracy and reliability of data in a database. PostgreSQL provides a wide range of features and tools to ensure that the data stored in a database is consistent and reliable. This recipe will provide an overview of some of the essential aspects of

working with database consistency and integrity in PostgreSQL, along with an example.

## Unique constraints

In the context of hotel reservations, we can use unique constraints to ensure that each reservation has a unique reservation ID, and that each guest has a unique email address. Let us consider an example schema for a hotel reservation system:

```
# Create reservations table
CREATE TABLE reservations (
    reservation_id SERIAL PRIMARY KEY,
    guest_name TEXT NOT NULL,
    guest_email TEXT NOT NULL,
    check_in_date DATE NOT NULL,
    check_out_date DATE NOT NULL,
    room_number INTEGER NOT NULL
);
```

In this example, we have a **reservations** table with columns for the **reservation ID**, **guest name**, **guest email**, **check-in date**, **check-out date**, and **room number**. We can enforce unique constraints on the **reservation\_id** and **guest\_email** columns using the following **ALTER TABLE** statements:

```
# Add constraint on reservation table
ALTER TABLE reservations ADD CONSTRAINT unique_reservation_id
UNIQUE (reservation_id);
ALTER TABLE reservations ADD CONSTRAINT unique_guest_email
UNIQUE (guest_email);
```

The first statement creates a unique constraint on the **reservation\_id** column, which ensures that each reservation has a unique ID. The second statement creates a unique constraint on the **guest\_email** column, which ensures that each guest has a unique email address.

Now let us consider a scenario where a hotel guest tries to make a reservation using an email address that is already associated with another reservation. When the hotel reservation system tries to insert the new reservation into the **reservations** table, the unique constraint on the **guest\_email** column will be violated, and the insert operation will fail. This prevents data duplication and ensures that each guest has a unique reservation.

Suppose we already have a reservation in the **reservations** table with the following details:

	reservation_id [PK] integer	guest_name text	guest_email text	check_in_date date	check_out_date date	room_number integer
1	1	Bugs Bunny	bugsbunny@pgtest.com	2023-06-10	2023-06-15	102

**Figure 8.20:** Unique constraint – for reservation table

Now, let us try to insert a new reservation with the same guest email:

```
# Insert data into the reservations table
INSERT INTO reservations (guest_name, guest_email, check_in_date,
check_out_date, room_number)
VALUES ('Bugs Bunny', 'bugsbunny@pgtest.com', '2023-06-20', '2023-
06-23', 103);
```

Since the guest email **bugsbunny@pgtest.com** is already associated with reservation ID 1, the unique constraint on the **guest\_email** column will be violated, and the insert operation will fail with an error message like. Please refer to the following figure:

```
ERROR: duplicate key value violates unique constraint "unique_guest_email"
DETAIL: Key (guest_email)=(bugsbunny@pgtest.com) already exists.
SQL state: 23505
```

**Figure 8.21:** Unique constraint – Duplicate key error

This prevents data duplication and ensures that each guest has a unique reservation.

## Foreign key constraints

Foreign key constraints in PostgreSQL ensure that the values inserted into a child table referencing a parent table must exist in the parent table. This maintains referential integrity between the tables and helps to prevent data inconsistencies.

Suppose we have a hotel reservation system that tracks reservations made by customers. We can create two tables: Customers and reservations. The customers table will store information about the customers, such as their name and email address, and the reservations table will store information about their reservations, such as the room number, check-in and check-out dates, and the customer who made the reservation.

Let us consider an example schema for a hotel reservation system:

```
# Create customer table
CREATE TABLE customers (
    customer_id SERIAL PRIMARY KEY,
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE
);

# Create reservation table
CREATE TABLE reservations (
    reservation_id SERIAL PRIMARY KEY,
    room_number INTEGER NOT NULL,
    check_in_date DATE NOT NULL,
    check_out_date DATE NOT NULL,
    customer_id INTEGER NOT NULL,
    CONSTRAINT FK_Rsv_Dpt FOREIGN KEY (customer_id) REFERENCES
        customers(customer_id)
);
```

This creates two tables, customers, and reservations. The customers table has a unique constraint on the email column to ensure that each customer has a unique email address. The reservations table has a foreign key constraint on the **customer\_id** column, which references the **customer\_id** column in the **customers** table.

This ensures that each reservation is associated with a valid customer, and that no reservation can be created for a customer that does not exist in the customers table. If an attempt is made to create a reservation with a non-existent customer ID, the foreign key constraint will prevent it and raise an error.

Here is an example scenario of how these constraints might work in practice, suppose we have the following data in our customers table:

	customer_id [PK] integer	first_name text	last_name text	email text
1	1	Homer	Simpson	momersimpson@pgtest.com
2	2	Bugs	Bunny	bugsbunny@pgtest.com

**Figure 8.22:** Foreign key - Customers table dataset

Now, suppose we want to create a reservation for room 101, with check-in on May 1, 2023 and check-out on May 3, 2023, for a customer with ID 1 (Homer Simpson). We can do this with the following SQL statement:

```
# Insert data into the reservation table
INSERT INTO
    reservations (room_number, check_in_date, check_out_date, customer_id)
VALUES (101, '2023-05-01', '2023-05-03', 1);
```

This will create a new reservation in the reservations table with a foreign key reference to customer ID 1 in the customers table.

However, if we try to create a reservation for a non-existent customer ID, like this:

```
# Insert data into the reservation table with existing
Customer ID
INSERT INTO
```

```
reservations (room_number, check_in_date, check_out_date, customer_id)
VALUES (101, '2023-05-01', '2023-05-03', 1);
```

The foreign key constraint will prevent it and raise an error, because there is no customer with ID 3 in the **customers** table. This helps ensure the integrity of our data and prevent inconsistencies in our database.

## Recipe 69: Getting insight to Python and Java connection

PostgreSQL is a powerful open-source relational database management system that offers support for multiple programming languages, including Python and Java. Both Python and Java are popular programming languages that are widely used for developing web applications and software tools. In this recipe, we will explore how to establish a connection between Python and Java with PostgreSQL and use it to build a sample application.

Recipe Prerequisite:

- PostgreSQL 15 database
- Python 3.9 or higher
- PostgreSQL JDBC driver for Java
- PyCharm (or any other Python IDE)

To connect to PostgreSQL using Python and Java, you will need to install Python and Java on your client system:

**Note: Installing Java and Python is out of the scope of this book. Please refer to the documentation provided by the official websites of Java ([https://www.java.com/en/download/help/linux\\_x64rpm\\_install.html](https://www.java.com/en/download/help/linux_x64rpm_install.html)) and Python (<https://www.python.org/downloads/>) for instructions on how to download and install these languages.**

- Once you have installed Java and Python, verify that they are installed correctly by running the following commands in your terminal:

```
# Verify the java & python version
```

```
java -version
```

```
python --version
```

These commands should display the versions of Java and Python installed on your system respectively.

- Install PostgreSQL JDBC driver for Java and connect to PostgreSQL database. Download the latest PostgreSQL JDBC driver for Java from the official website (<https://jdbc.postgresql.org/>).
- Locate the downloaded jar file **postgresql-42.6.0.jar** to a directory on client/server system.
- Add the path to the **postgresql-42.6.0.jar** file to **CLASSPATH** environment variable. We can do this by executing the following command in the terminal:

```
# Set environment variable
```

```
export CLASSPATH=$CLASSPATH:/pg_jar/postgresql-42.6.0.jar
```

In this command, replace **/pg\_jar/postgresql-42.6.0.jar** with the actual path to the **postgresql.jar** file on your machine.

- We can now use the PostgreSQL JDBC driver in our Java program. To establish a connection to a PostgreSQL database save the following Java program code in the file **JDBC\_PG\_Connection.java**.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class JDBC_PG_Connection {
    public static void main(String[] args) {
        Connection conn = null;
        try {
            conn =
                DriverManager.getConnection("jdbc:postgresql://localhost:5432/mydb",
```

```

    "myusr", "mypassword");
} catch (SQLException e) {
    System.err.println(e);
    System.exit(1);
}
System.out.println("Connected to the PostgreSQL server
successfully.");
// use the connection to execute SQL queries
// ...
try {
    conn.close();
} catch (SQLException e) {
    System.err.println(e);
}
}
}

```

In this code, replace **mydb**, **myuser**, and **mypassword** with the actual database name, username, and password respectively. Also, ensure to replace localhost and 5432 with the actual hostname and port number of PostgreSQL server.

- That is it! We should now be able to use the PostgreSQL JDBC driver in your Java program. Check the PostgreSQL database connection by executing the above Java programme using the following command.

*# Execute java programme*

```
java JDBC_PG_Connection.java
```

The command `java JDBC_PG_Connection` executes a Java program named **JDBC\_PG\_Connection**, which contains the code to connect to a PostgreSQL database using JDBC and print a message indicating whether the connection was successful or not.

If the connection is successful, the program prints the message **Connected to the PostgreSQL server successfully**. If the connection fails, the program prints an

error message to the console and exits. Please refer to the following figure:

```
[root@pgmaster2 ~]# java JDBC_PG_Connection  
Connected to the PostgreSQL server successfully.
```

**Figure 8.23:** Java connection to PostgreSQL DB

Referring to the [Figure 8.23](#), the program was able to successfully connect to the PostgreSQL database and printed the **Connected to the PostgreSQL server successfully** message to the console.

7. **Connect to PostgreSQL using Python:** To connect to a PostgreSQL database using Python, we will use the **psycopg2** module, which provides a Python interface to the PostgreSQL database. The following code snippet that demonstrates how to connect to a PostgreSQL database using Python and the **psycopg2** module and select the data from table sales.
8. Save the following python code snippet in the file **Python\_PG\_Connection.py** file:

```
import psycopg2  
# Connect to the PostgreSQL database  
conn = psycopg2.connect(  
    host="192.168.187.134",  
    database="mydb",  
    user="myusr",  
    password="mypassword"  
)  
print(" ## Connected to PostgreSQL database! MYDB ## ")  
print()  
# Create a cursor object to interact with the  
database  
cur = conn.cursor()  
print(" ## Select dataset from sales table! ## ")  
print("-----")  
# Execute a SQL query
```

```

cur.execute("SELECT * FROM sales")
    # Fetch the results of the query
rows = cur.fetchall()
    # Print the results
for row in rows:
    print(row)
    # Close the cursor and database connection
cur.close()
conn.close()

```

Referring to the *Figure 8.24*, in the code above, we first import the **psycopg2** module, which provides the **connect()** method to establish a connection to a PostgreSQL database:

```

[root@pgmaster2 ~]# python3 Python_PG_Connection.py
## Connected to PostgreSQL database! MYDB ##

## Select dataset from sales table! ##

-----
(483, 'Server_Rack', datetime.date(2021, 1, 1), Decimal('150000'))
(487, 'UPS', datetime.date(2021, 1, 1), Decimal('10000'))
(451, 'Server_hardware', datetime.date(2021, 2, 1), Decimal('300000'))
(452, 'Server_hardware', datetime.date(2021, 3, 1), Decimal('350000'))
(459, 'Server_hardware', datetime.date(2021, 3, 1), Decimal('350000'))
(411, 'Server_hardware', datetime.date(2021, 3, 2), Decimal('350000'))
(412, 'UPS', datetime.date(2021, 2, 5), Decimal('50000'))
(413, 'UPS', datetime.date(2021, 3, 5), Decimal('50000'))
(481, 'Server_hardware', datetime.date(2021, 1, 1), Decimal('100000'))

```

*Figure 8.24: Python connection to PostgreSQL DB*

We then call the **connect()** method with the connection details for your PostgreSQL database, including the **hostname**, **database name**, **username**, and **password**. This method returns a connection object, which you can use to interact with the database.

Next, you create a cursor object using the **cursor()** method of the connection object. This cursor object allows you to execute SQL queries and fetch the results.

We then execute a SQL query using the `execute()` method of the cursor object. In this example, the query selects all rows from sales table in the database.

We fetch the results of the query using the `fetchall()` method of the cursor object, which returns a list of tuples containing the query results.

Finally, you print the results and close the cursor and database connection using the `close()` method.

## Recipe 70: Importing BLOB data types into PostgreSQL

BLOBS are commonly used to store large binary data, such as images, audio files, or other binary files. PostgreSQL offers several methods to import BLOB data into your database, and we will discuss these methods with a step-by-step guide and example scenarios. By the end of this chapter, you will have a comprehensive understanding of how to import BLOB data into PostgreSQL.

In this recipe, we will explore the process of importing image files into a PostgreSQL database. We will create a sample table, populate it with image data, and demonstrate how to retrieve and display these images using SQL commands.

1. Create a table for storing image data:

```
# Create a table to store image data
CREATE TABLE images (
    image_id serial PRIMARY KEY,
    image_name text,
    image_data bytea
);
```

In this step, we create a table called **images** with columns for **image\_id** (an auto-incrementing primary key), **image\_name** (to store the image's name or description), and **image\_data** (a bytea data type to store the binary image data).

2. Prepare image files for import:

Before proceeding, make sure you have your image files ready for import. You can use any image files in common formats like JPEG or PNG.

### 3. Use the **INSERT** command to import image data:

To import image data into the **images** table, you can use the **INSERT** command. Here is an example of how to import an image file into the table:

```
# Import an image file into the "images" table
INSERT INTO images (image_name, image_data) VALUES
('IMG_1495.png',
pg_read_binary_file('/var/lib/pgsql/IMG_1495.png')::bytea);
```

Replace **/var/lib/pgsql/IMG\_1495.png** with the actual path to your image file. This command imports the image's name and binary data into the **images** table.

### 4. Query the database to retrieve and display images:

To retrieve and display an image from a bytea column in PostgreSQL, you can use SQL commands in combination with your application code. Here is a simplified SQL command to retrieve an image based on its **image\_name** from the **images** table:

```
# Retrieve and display a specific image by name
SELECT image_name, image_data FROM images WHERE
image_name = 'IMG_1495.PNG';
```

Referring to [Figure 8.25](#), this SQL query will return the binary image data stored in the bytea column. However, displaying the image in an application typically involves reading the binary data from the query result and rendering it appropriately in your application code. Please refer to the following figure:

2	<code>SELECT image_data FROM images WHERE image_name = 'IMG_1495.jpg';</code>
Data Output    Messages    Notifications	
	image_data bytea
1	[binary data]
2	[binary data]

**Figure 8.25:** Retrieve binary image with SQL query

In a real application, you would use a server-side language (for example, Python, PHP, Java) to fetch the binary data, convert it into an image format (for example, JPEG, PNG), and then display it to the user through a web page or application interface.

Here is a simplified Python example using the **psycopg2** library to retrieve and display an image:

```

import psycopg2
from io import BytesIO
from PIL import Image
# Connect to the PostgreSQL database
conn = psycopg2.connect("dbname=postgres user=postgres
password=postgres host=192.168.187.134")
cur = conn.cursor()
# Retrieve the image data
cur.execute("SELECT image_data FROM images WHERE
image_name = 'IMG_1495.png'")
image_data = cur.fetchone()[0]
# Close the database connection
cur.close()
conn.close()
# Display the image using Python's PIL library
(Pillow)
image = Image.open(BytesIO(image_data))
image.show()
```

Execute the above python code snippet to retrieve and display the image directly to the image browser, allowing you to view it in an image browser window.

*# Execute the python Code snippet*

`python3 retrieve.py`

This Python code connects to the PostgreSQL database, retrieves the image data, and displays the image using the PIL (Pillow) library.

## Conclusion

In this chapter we understood how SQL can be used to access and manipulate data stored in PostgreSQL databases. We began with the basics of SQL syntax, data types, and functions, and progressed to more advanced topics such as joins, subqueries, and indexing. We also explored practical examples of how to use SQL to work with PostgreSQL databases, which will be invaluable for those working in roles such as database administrators, developers, or data analysts.

By mastering SQL and applying it to PostgreSQL databases, users can optimize the way they handle and analyse data, making their work more efficient and effective. With its advanced features and widespread usage, PostgreSQL is a powerful tool for managing large amounts of relational data, and SQL is the key to unlocking its full potential.

The upcoming chapter is dedicated to fortifying the security measures within database environment. We will delve into the foundational concepts of authentication and encryption specific to PostgreSQL, empowering you to establish robust security protocols. By exploring prevalent techniques for auditing and implementing secure authentication practices, you will gain a comprehensive understanding of bolstering the security posture of your PostgreSQL database.

## 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)**



# CHAPTER 9

# Server Controls and Auditing

## Introduction

In the realm of database management, the security of sensitive data is of paramount importance. PostgreSQL 15 offers an array of tools and features to bolster this security. This chapter is dedicated to exploring the foundational concepts underpinning authentication and encryption within the PostgreSQL database server. We will also delve into common methods for encrypting client connections and practical solutions for managing user and group access, thereby enhancing the security of your database. Additionally, there is a dedicated section covering the vital topic of database authentication through SSL.

## Structure

In this chapter, we will cover the following topics:

- Introduction to server control and auditing
- Database privileges
- SSL/TLS authentication
- PostgreSQL auditing
- LDAP authentication

## Objectives

The primary objectives of this chapter revolve around equipping you with a comprehensive understanding of server controls and auditing techniques specific to PostgreSQL. Our key goals include fostering a deep comprehension of the fundamental principles underpinning authentication and encryption within the PostgreSQL database server. We will also guide you through the exploration of prevalent techniques for encrypting client connections and the implementation of secure authentication practices. Moreover, we aim to offer practical solutions for managing user and group control, providing you with the tools to establish versatile access control to your database. Lastly, our objectives encompass shedding light on the intricacies of database authentication through SSL and offering clear guidance on its seamless integration within PostgreSQL 15. Through the accomplishment of these objectives, you will be well-prepared to enhance the security and control of your PostgreSQL 15 database environment.

## **Introduction to server control and auditing**

Server control refers to the ability to control access to the server, limit its resources, and manage its configuration. Auditing, on the other hand, involves tracking and analyzing user activity on the server, which can be useful for compliance, troubleshooting, and security purposes.

In PostgreSQL, administrators continue to benefit from a long-standing and impactful server control feature that enables the restriction of access based on the client's IP address. This capability, accessible through the configuration of the **pg\_hba.conf** file, empowers administrators to specify and limit connections to the server from particular IP addresses or ranges. By utilizing this well-established feature, the server's attack surface is effectively minimized, ensuring that access remains confined to trusted sources and contributing to enhanced overall security.

Alongside IP address filtering, PostgreSQL features a logging system for auditing purposes. For instance, it allows logging details about user activities, such as query duration and the

number of rows returned. This functionality, available in previous releases as well, aids in performance analysis, bottleneck identification, and issue troubleshooting.

In PostgreSQL, administrators can continue leveraging a long-standing auditing feature related to the use of the **COPY** command. This command facilitates the import and export of data in PostgreSQL. By implementing auditing for the **COPY** command, administrators gain the capability to monitor user interactions with data, providing insights into user access and the specific data being accessed.

In summary, the server control and auditing features in PostgreSQL offer administrators enhanced control over server access and greater visibility into user activity. This can help organizations improve the security and reliability of their database management systems, as well as comply with regulatory requirements.

## **Recipe 71: Database privileges**

Database privileges play a crucial role in ensuring the security of a database management system. Database privileges refer to the level of access granted to users or roles for performing specific actions on a database or its objects, such as tables, views, or functions. PostgreSQL 15 provides a robust system for managing database privileges, allowing administrators to grant or revoke privileges as needed.

There are several types of privileges in PostgreSQL, including table privileges, sequence privileges, function privileges, and schema privileges. Table privileges refer to the ability to perform actions on a table, such as selecting, inserting, updating, or deleting data. Sequence privileges allow users to create or modify sequences, which are used for generating unique values in a table. Function privileges refer to the ability to execute a function or procedure. Schema privileges allow users to create or modify schemas, which are used for organizing database objects.

In addition to granting and revoking privileges, administrators can also use roles to manage database access. Roles are used to

group users and assign privileges to them as a group, simplifying the management of database access. This recipe will guide you through understanding and managing database privileges in PostgreSQL.

## 1. List database-level privileges:

To list the user privileges at the database level:

```
# List database privileges
```

```
SELECT
    datname AS database_name,
    pg_roles.rolname AS owner,
    datacl AS privileges
FROM
    pg_database
JOIN
    pg_roles ON pg_database.datdba = pg_roles.oid;
```

In this query, we are retrieving the **datname** (database name), **rolname** (owner), and **datacl** (privileges) from the **pg\_database** catalog table, joined with **pg\_roles** to get the owner's name.

The **datacl** column contains the detailed access control list. In this example, **{=Tc/postgres,postgres=CTc/postgres,test=c/postgres}** indicates that the test user has the **CONNECT** privilege.

## 2. Fetch schema-level privileges:

To list the user privileges at the schema level execute the following **sql**.

```
# Fetch schema-level privileges
```

```
SELECT
```

```
n.nspname AS schema_name,
```

```
pg_roles.rolname AS owner,
```

```
CASE
```

```
    WHEN has_schema_privilege(n.nspname, 'USAGE') THEN 'USAGE'
```

```
    WHEN has_schema_privilege(n.nspname, 'CREATE') THEN 'CREATE'
```

```
    WHEN has_schema_privilege(n.nspname, 'CREATE TEMPORARY')
```

```

TABLE') THEN 'CREATE TEMPORARY TABLE'
WHEN has_schema_privilege(n.nspname, 'CREATE ON SCHEMA')
THEN 'CREATE ON SCHEMA'
-- Add more WHEN clauses for other privileges as
needed
ELSE NULL
END AS privileges,
n.nspacl AS acl
FROM
pg_namespace n
JOIN
pg_roles ON n.nspowner = pg_roles.oid;

```

The output **{postgres=UC/postgres,test=UC/postgres}** suggests that both the postgres and test roles have the **USAGE** and **CREATE** privileges on the schemas being inspected. The format of the output is an **access control list (ACL)** where UC stands for **USAGE** and **CREATE**.

### 3. Fetch object-level privileges (tables, views, and so on):

To list the user privileges at the table level, execute the following **sql**.

```

# Fetch Object-Level Privileges (Tables, Views,
etc.)
SELECT
table_name,
grantee,
privilege_type
FROM
information_schema.table_privileges;

```

- **Fetch specific table privileges:**

Fetching specific table privileges involves querying the system catalog tables to retrieve detailed information about the permissions granted on a particular table.

```
# Fetch specific table Privileges
SELECT
    table_name,
    grantee,
    privilege_type
FROM
    information_schema.table_privileges where
    table_name='table_name' ;
```

#### 4. Fetch object-level privileges (functions, procedures, and so on):

To view the existing privileges on a specific routine using SQL, use the following query:

```
# Fetch Object-Level Privileges (Functions,
Procedures, etc.):
SELECT
    specific_name,
    grantee,
    privilege_type
FROM
    information_schema.routine_privileges;
```

- **Fetch specific routine privileges:**

Fetching specific routine privileges involves querying the system catalog tables to retrieve detailed information about the permissions granted on a particular routine, such as functions or procedures.

```
# Fetch specific routine Privileges
SELECT
    specific_name,
    grantee,
    privilege_type
FROM
    information_schema.routine_privileges where
    specific_name='routine_name' ;
```

## 5. Grant database objects:

Granting database objects in PostgreSQL 15 involves providing specific permissions and privileges to users or roles, enabling them to perform various actions on tables, views, schemas, and other database entities. The **GRANT** command is pivotal in this process, allowing administrators to define who can access or manipulate certain resources and specifying the type of actions permitted, such as **SELECT, INSERT, UPDATE, DELETE**, and more.

- **Grant SELECT on specific table:**

*# Grant SELECT on specific table:*

```
GRANT SELECT, INSERT ON your_table_name TO your_user;
```

- **Grant SELECT, INSERT, UPDATE, DELETE on a specific table:**

*# Grant SELECT, INSERT, UPDATE, DELETE on a specific table:*

```
GRANT SELECT, INSERT, UPDATE, DELETE ON your_table_name  
TO your_user;
```

- **Grant ALL PRIVILEGES on specific table:**

*# Grant ALL PRIVILEGES on specific table:*

```
GRANT ALL PRIVILEGES ON your_table_name TO your_user;
```

- **Grant USAGE on specific schema:**

*# Grant USAGE on specific schema:*

```
GRANT USAGE ON SCHEMA your_schema_name TO your_user;
```

- **Grant ALL PRIVILEGES on specific schema:**

*# Grant ALL PRIVILEGES on specific schema:*

```
GRANT ALL ON SCHEMA ON SCHEMA your_schema_name TO  
your_user;
```

- **Grant ALL PRIVILEGES on all tables under specific schema:**

*# Grant ALL PRIVILEGES on all table under specific*

*schema:*

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA  
your_schema_name TO your_user;
```

## 6. Revoke privileges:

Revoking privileges is a crucial aspect of access management, providing administrators with the means to restrict or remove specific permissions granted to roles or users:

- **Revoke SELECT from specific table:**

*# Revoking UPDATE from specific table*

```
REVOKE SELECT ON your_table_name FROM your_user_or_role;
```

- **Revoke SELECT, INSERT, UPDATE, DELETE from a specific table:**

*# Revoke SELECT, INSERT, UPDATE, DELETE  
privileges from a specific table*

```
REVOKE SELECT, INSERT, UPDATE, DELETE ON your_table_name  
FROM your_user_or_role;
```

- **Revoke ALL PRIVILEGES from specific table:**

*# Revoke ALL PRIVILEGES from specific table:*

```
REVOKE ALL PRIVILEGES ON your_table_name FROM your_user;
```

- **Revoke USAGE from specific schema:**

*# Revoke USAGE from specific schema:*

```
REVOKE USAGE ON SCHEMA your_schema_name FROM your_user;
```

- **Revoke ALL PRIVILEGES from specific schema:**

*# Revoke ALL PRIVILEGES from specific schema:*

```
REVOKE ALL ON SCHEMA ON SCHEMA your_schema_name FROM  
your_user;
```

- **Revoke ALL PRIVILEGES from all tables under specific schema:**

*# Revoke ALL PRIVILEGES from all table under  
specific schema:*

```
REVOKE ALL PRIVILEGES ON ALL TABLES IN SCHEMA  
your_schema_name FROM your_user;
```

## 7. Using default privileges:

In PostgreSQL 15, leveraging default privileges is a powerful mechanism for streamlining access control and permissions within a database. Default privileges allow administrators to set a template for permissions that will be automatically applied to objects created in the future:

- **Grant default privileges to user for any future tables user create:**

```
# Setting default privileges for a specific user  
ALTER DEFAULT PRIVILEGES FOR USER your_user  
GRANT SELECT, INSERT, UPDATE ON TABLES TO your_user;
```

- **Grant Default Privileges in Schema to Specific user:**

```
# Granting default privileges in a schema to a  
specific user  
ALTER DEFAULT PRIVILEGES FOR USER your_user IN SCHEMA  
your_schema  
GRANT SELECT, INSERT, UPDATE ON TABLES TO your_user;
```

## 8. Role authorization:

Authorizing roles in PostgreSQL is a fundamental aspect of managing access and permissions within the database. Roles serve as a mechanism to group users and assign specific privileges to them collectively.

- **Role inheritance:**

```
# Creating a role with inheritance  
CREATE ROLE manager WITH INHERIT;  
# Granting a role to another role  
GRANT manager TO employee;
```

- **Set role attributes:**

```
# Setting role attributes like LOGIN and
```

## SUPERUSER

```
ALTER ROLE your_user LOGIN SUPERUSER;
```

- **Role revocation:**

*# Revoking a role from a user*

```
REVOKE your_role FROM your_user;
```

- **Cascade option for revocation:**

*# Revoking a role and cascading the revocation to dependent objects*

```
REVOKE your_role FROM your_user CASCADE;
```

- **Grant default privileges to role for any future tables user create:**

*# Setting default privileges for a specific user*

```
ALTER DEFAULT PRIVILEGES FOR ROLE your_role
```

```
GRANT SELECT, INSERT, UPDATE ON TABLES TO your_user;
```

- **Grant default privileges in schema:**

*# Granting default privileges in a schema to a specific user*

```
ALTER DEFAULT PRIVILEGES FOR ROLE your_role IN SCHEMA
```

```
your_schema
```

```
GRANT SELECT, INSERT, UPDATE ON TABLES TO your_user;
```

## Recipe 72: PostgreSQL role management and authorization

Managing roles membership, attributes, authentication, and authorizations in PostgreSQL is essential to ensure proper security and access control in the database system. In this recipe, we will go through the steps involved in managing roles in PostgreSQL 15, including creating and modifying roles, assigning attributes, and setting up authentication and authorization.

Suppose we have a database named **company** that stores information about **employees**, **departments**, and **salaries**. We want to create roles for various users and grant them appropriate

privileges to access the data. The roles we need to create are as follows:

- **Admin**: This role will have full access to all the tables and functions in the database.
- **Manager**: This role will have access to the employees and departments table and can view the salaries table.
- **Analyst**: This role will have read-only access to the employees and departments table but cannot view the salaries table.

Here is a table providing information about the predefined roles. Please note that the specifics might change in newer versions of PostgreSQL, and it is recommended to refer to the official documentation for the most up-to-date information.

Role	Description
<b>pg_read_all_stats</b>	Allows reading statistical information.
<b>pg_stat_scan_tables</b>	Allows to execute monitoring functions that may take <b>ACCESS SHARE</b> locks on tables.
<b>pg_write_all_data</b>	Allows writing data to any table.
<b>pg_read_all_data</b>	Allows reading data to any table.
<b>pg_execute_server_program</b>	Allows execution of server-side programs using the <b>COPY</b> .
<b>pg_signal_backend</b>	Enables signalling specific backends, useful for debugging and process management.
<b>pg_monitor</b>	Provides access to monitoring functions, allowing the role to view system views related to performance.
<b>pg_read_server_files</b>	A role that able to read files on the server.
<b>pg_write_server_files</b>	A role that able to write files on the server.
<b>pg_database_owner</b>	Automatically assigned to the role that creates a database. Allows the role to

	modify or drop the database.
<b>pg_checkpoint</b>	Allows execution of checkpoints.
<b>pg_read_all_settings</b>	Allows reading all configuration settings.

**Table 9.1:** predefined roles

1. To create a new role (regular role) in PostgreSQL, we use the **CREATE ROLE** command. We will start by creating the **admin** role using following command:

```
CREATE ROLE admin LOGIN PASSWORD 'Admin@123';
```

The command above creates a new role named **admin** with login privileges and sets the password to **Admin@123**. Replace **password** with your desired password.

Next, we will create the **manager** role:

```
CREATE ROLE manager LOGIN PASSWORD 'Manager@123';
```

And finally, the **Analyst** role:

```
CREATE ROLE analyst LOGIN PASSWORD 'Analyst@123';
```

2. Create **SUPERUSER** role in PostgreSQL. We will start by creating the **admin\_user** role using following command:

*# Creating a superuser role*

```
CREATE ROLE admin WITH SUPERUSER LOGIN PASSWORD  
'admin_password';
```

3. After creating the roles, we need to assign attributes to them and define their privileges. In PostgreSQL, we use the **GRANT** command to assign privileges.

First, we will assign all privileges to the **admin** role:

```
GRANT ALL PRIVILEGES ON DATABASE company TO admin;  
GRANT USAGE ON SCHEMA public TO admin;
```

This grants the **admin** role **all privileges** on the **company** database and **public** schema, including creating and modifying tables, executing functions, and so on.

Next, we will assign privileges to the **manager** role:

```
GRANT SELECT, INSERT, UPDATE ON employees, departments TO  
manager;  
GRANT USAGE ON SCHEMA public TO manager;
```

```
GRANT SELECT ON salaries TO manager;
```

This grants the **manager** role select, insert, and update privileges on the **employees** and **departments** tables and select privileges on the **salaries** table.

Finally, we will assign privileges to the **analyst** role:

```
GRANT SELECT ON employees, departments TO analyst;
```

```
GRANT USAGE ON SCHEMA public TO analyst;
```

This grants the Analyst role select privileges on the **employees** and **departments** tables.

4. Grant predefined built-in role in PostgreSQL, we use the **GRANT** command to assign privileges.

*# Granting specific built-in privileges to a regular role*

```
ALTER ROLE admin WITH pg_stat_scan_tables, pg_write_all_data,  
pg_write_server_files;
```

5. Verify that the **admin** role has full access to all tables and functions in the **company** database by logging in as the **admin** user and attempting to create, modify, and execute various SQL statements.

*# Login to the database with admin user*

```
psql -U admin -d company
```

*# Create a table, insert data and verify the inserted data*

```
CREATE TABLE test_admin (id SERIAL PRIMARY KEY, name TEXT);
```

```
INSERT INTO test_admin (name) VALUES ('Buggy');
```

```
SELECT * FROM test_admin;
```

The first command logs in to the **company** database as the **admin user**. The second command creates a new table named **test\_admin** with a **id** column and a **name** column. The third command inserts a new row with a name of **Buggy** into the **test\_admin** table. The fourth command selects all rows from the **test\_admin** table to verify that the insert was successful. This test ensures that the **admin** role can create and modify tables and execute SQL statements in the **company** database.

6. Verify that the **manager** role has access to the **employees** and **departments** tables and can view the **salaries** table by logging in as the **manager** user and attempting to select, insert, and update records in the **employees** and **departments** tables, and select records from the **salaries** table.

```
# Login to the database with manager user
```

```
psql -U manager -d company  
# Insert, update and select the data  
SELECT * FROM employees;  
INSERT INTO employees (name, department_id) VALUES ('Scooby',  
1);  
UPDATE employees SET salary = 60000 WHERE name = 'Scooby';  
SELECT * FROM salaries;
```

The first command logs in to the **company** database as the **manager user**. The second command selects all rows from the **employees** table to verify that the **manager** role has access to it. The third command inserts a new row into the **employees** table with a name of **Scooby** and a **department\_ID** of **1**. The fourth command updates the salary of the employee named **Scooby** to **60000**. The fifth command attempts to select all rows from the **salaries** table to verify that the **manager** role cannot access it. This test ensures that the **manager** role has the correct privileges in the **company** database.

7. Verify that the **analyst** role has read-only access to the **employees** and **departments** tables and cannot view the **salaries** table by logging in as the **analyst** user and attempting to select records from the **employees** and **departments** tables and the **salaries** table.

```
# Login to the database with analyst user
```

```
psql -U analyst -d company  
# select the data  
SELECT * FROM employees;  
SELECT * FROM departments;  
SELECT * FROM salaries;
```

The first command logs in to the **company** database as the **analyst** user. The second command selects all rows from the **employees** table to verify that the analyst role has read-only access to it. The third command selects all rows from the **departments** table to verify that the analyst role has read-only access to it. The fourth command attempts to select all rows from the **salaries** table to verify that the analyst role cannot access it. This test ensures that the analyst role has the correct privileges in the **company** database.

8. To set up authentication and authorization for the roles, we will use the **pg\_hba.conf** and **pg\_ident.conf** files in PostgreSQL.

In **pg\_hba.conf**, we will add the following lines to enable password authentication for the roles:

#	TYPE	DATABASE	USER	ADDRESS	METHOD
	host	company	admin	0.0.0.0/0	scram-sha-256
	host	company	manager	0.0.0.0/0	scram-sha-256
	host	company	analyst	0.0.0.0/0	scram-sha-256

This allows the **admin**, **manager**, and **analyst** roles to log in from any IP address using the **scram-sha-256** authentication method.

In **pg\_ident.conf**, we will add the following lines to map the roles to the corresponding Unix users.

#	MAPNAME	SYSTEM-USERNAME	PG-USERNAME
	pg_user	admin	admin
	pg_user	manager	manager
	pg_user	analyst	analyst

This maps the Unix users **admin**, **manager**, and **analyst** to the PostgreSQL roles of the same names.

## SSL/TLS authentication

SSL/TLS authentication provides an additional layer of security for database management systems by encrypting data transmitted between the client and server. SSL/TLS authentication ensures that sensitive data, such as passwords and queries, are not transmitted in plaintext, making it more difficult for attackers to intercept and steal this information.

To use SSL/TLS authentication in PostgreSQL 15, administrators must first configure the server to use SSL/TLS. This involves creating an SSL/TLS certificate, which can be self-signed or signed by a trusted certificate authority, and configuring the server to use this certificate for encryption. Once SSL/TLS is enabled on the server, clients must also be configured to use SSL/TLS for authentication.

When a client connects to the server using SSL/TLS authentication, the server sends its SSL/TLS certificate to the client, which verifies that the certificate is valid and issued by a trusted authority. The client then sends its own SSL/TLS certificate to the server, allowing the server to verify the client's identity.

SSL/TLS authentication can be used in conjunction with other authentication methods, such as password authentication or client certificate authentication, to provide an additional layer of security. By encrypting data transmitted between the client and server, SSL/TLS authentication can help protect sensitive data from interception and theft, making it an essential feature for secure database management systems.

## **Recipe 73: Setting up SSL authentication in PostgreSQL**

**Secure Sockets Layer (SSL)** is a cryptographic protocol used to secure communications over a computer network. It provides encryption and authentication to ensure that data transmitted between two devices is secure and cannot be intercepted or tampered with by unauthorized parties.

In the context of PostgreSQL, SSL can be used to provide secure authentication for clients connecting to the database server. In

this recipe, we will cover the steps to set up SSL authentication in PostgreSQL 15.

1. Firstly, assume that PostgreSQL 15 database cluster instance is running and succeed with the connection to the PostgreSQL database cluster.
2. Then create a self-signed certificate, The following command creates a self-signed SSL/TLS certificate for PostgreSQL server using **openSSL**. The command generates a new private key and a self-signed certificate that is valid for **365** days.

```
# Configure PostgreSQL for SSL
```

```
openssl req -new -newkey rsa:2048 -days 365 -nodes -x509 -keyout  
postgresql.key -out postgresql.crt
```

After executing the command, **openSSL** will prompt for additional information that will be included in the certificate. The user can enter the values for the country name, state or province name, locality name, organization name, organizational unit name, common name, and email address fields as per their requirement.

3. Create a server certificate and private key. The following command generates a private key and a **certificate signing request (CSR)** for a server. This certificate is used by the PostgreSQL server to authenticate itself to clients.

```
# Configure PostgreSQL for SSL
```

```
openssl req -new -nodes -out server.csr -keyout server.key
```

This command will create a CSR **server.csr** and a private key **server.key** for the server.

The use of a server certificate is to provide secure communication between a client and a server over a network. When a client connects to a server using SSL/TLS, the server presents its server certificate to the client. The client verifies the server certificate to ensure that it was issued by a trusted CA and that it matches the hostname of the server it is connecting to. If the verification is successful, the client and server can establish a secure encrypted communication channel.

We sign the server certificate using the CA we created earlier using the following command:

```
# Configure PostgreSQL for SSL
```

```
openssl x509 -req -in server.csr -days 365 -CA ca.crt -CAkey ca.key -  
CAcreateserial -out server.crt
```

This command creates a server certificate **server.crt** that is signed by the self-signed certificate **postgresql.crt**. The server certificate can then be used to secure connections to the PostgreSQL server using SSL/TLS encryption.

4. Configure PostgreSQL server. After creating the server certificate, we need to configure the PostgreSQL server to use SSL authentication. To do this, we need to add the following lines to the **postgresql.conf** file:

```
# Configure PostgreSQL for SSL
```

```
ssl = on  
ssl_cert_file = 'server.crt'  
ssl_key_file = 'server.key'  
ssl_ca_file = 'postgresql.crt'
```

5. Restart PostgreSQL server. After making changes to the **postgresql.conf** file, we need to restart PostgreSQL server for the change to take effect.
6. Finally, when SSL is configured on PostgreSQL server, we verify the SSL connection details by running the following SQL command.

```
# Configure PostgreSQL for SSL
```

```
SELECT * FROM pg_stat_ssl;
```

Referring to [Figure 9.1](#), above SQL execution display information about the SSL connection. If the **ssl** column is **true**, then the connection is using SSL. If the **ssl** column is **false**, then the connection is not using SSL. Please refer to the following figure: Should be as below.



The screenshot shows the pgAdmin interface with the 'Data Output' tab selected. The title bar includes 'Data Output', 'Messages', and 'Notifications'. Below the title bar is a toolbar with icons for file operations like Open, Save, and Print. The main area displays a table with the following data:

	pid	ssl	version	cipher	bits	client_dn	client_serial	issuer_dn
	integer	boolean	text	text	integer	text	numeric	text
1	99485	true	TLSv1.3	TLS_AES_256_GCM_SHA384	256	[null]	[null]	[null]
2	99486	true	TLSv1.3	TLS_AES_256_GCM_SHA384	256	[null]	[null]	[null]

**Figure 9.1:** Verify connection using SSL

That is it for the SSL configuration on PostgreSQL server. In SSL authentication, a client certificate is used to authenticate the client to the server. The client presents the server with a certificate signed by a trusted **certificate authority (CA)**, and the server can verify the certificate to authenticate the client.

## Exploring TDE and encryption in PostgreSQL

**Transparent Data Encryption (TDE)** and encryption are techniques used to protect sensitive data stored in a database. TDE encrypts the entire database, while encryption encrypts individual columns or fields within the database. In PostgreSQL, these techniques can be implemented using various extensions and plugins, such as pgcrypto and EDB advanced server.

To get insight into TDE and encryption in PostgreSQL, we need to first understand the concepts and techniques involved in these processes. We also need to understand how to implement TDE and encryption in PostgreSQL. Here are the steps to get insight into TDE and encryption in PostgreSQL:

1. Understand the concepts of TDE and encryption:
  - TDE encrypts the entire database, providing data-at-rest protection.
  - Encryption encrypts individual columns or fields within the database, providing selective data protection.
  - Both techniques use encryption algorithms and keys to encrypt and decrypt data.
2. Choose a TDE or encryption solution:
  - In PostgreSQL, TDE and encryption can be implemented

using various extensions and plugins.

- The pgcrypto extension provides a basic set of encryption functions for individual columns or fields within the database.
- EDB advanced server provides more advanced TDE and encryption capabilities, including support for industry-standard encryption algorithms.

### 3. Implement TDE or encryption:

- To implement TDE, we need to encrypt the entire database using a TDE solution such as EDB advanced server.
- To implement encryption, we need to identify the sensitive columns or fields within the database and use an encryption solution such as pgcrypto to encrypt them.

### 4. Test and monitor TDE or encryption:

- After implementing TDE or encryption, we need to test the solution to ensure that it is working as expected.
- We also need to monitor the solution to ensure that there are no performance issues or security vulnerabilities.

## Recipe 74: Encryption with pgcrypto in PostgreSQL

PostgreSQL supports several encryption methods, including SSL/TLS encryption for securing connections and data at rest encryption for securing data stored in the database. This recipe is all about configuring encryption in PostgreSQL 15, with an example.

Let us say we have a PostgreSQL 15 database running on a Linux server. We want to enable SSL/TLS encryption to secure client connections to the database.

1. To set up PostgreSQL with additional contributed modules, download **postgresql15-contrib** package from the PostgreSQL official website at

[https://download.postgresql.org/pub/repos/yum/15/redhat/rhel-9-x86\\_64/](https://download.postgresql.org/pub/repos/yum/15/redhat/rhel-9-x86_64/) and continue with the installation. We can install this by using the following command:

```
# Install pgaudit package
```

```
yum localinstall postgresql15-contrib-15.2-1PGDG.rhel9.x86_64.rpm
```

This command will install the **postgresql15-contrib** package.

2. Next, we need to enable the **pgcrypto** extension in our PostgreSQL database. We can do this by connecting to our database using the **psql** command-line tool and running the following command:

```
# Create pgcrypto extension
```

```
CREATE EXTENSION pgcrypto;
```

3. Generate a secret key for **pgcrypto**, you can use the **gen\_random\_bytes()** function to create a **random byte** sequence of the desired length. For example, to generate a **32-byte** secret key, you can run the following command:

```
SELECT encode(gen_random_bytes(32), 'hex');
```

Referring to *Figure 9.2*, this will generate a random **32-byte** sequence and convert it to a hexadecimal string, which can be used as a secret key for **pgcrypto**.



The screenshot shows a PostgreSQL terminal window. The command entered is:

```
1 SELECT encode(gen_random_bytes(32), 'hex');
```

The output shows the generated hex string:

1	eab8d2be6136a865fb667f5cb3f4b8995ce63e9ead968b89cb12c2df980467...
---	---

**Figure 9.2:** Generated secret key

4. Once secret key generated, we should store it securely and only provide access to authorized users. We can store the secret key in a secure location, such as a password manager or a secure file storage system.
5. Now, we will create functions to encrypt and decrypt the sensitive data in your database:

```
# Function that encrypt data
```

```

CREATE OR REPLACE FUNCTION encrypt_data(data TEXT)
RETURNS BYTEA AS $$

DECLARE
key TEXT := 'secret-key';

BEGIN
RETURN pgp_sym_encrypt(data, key);
END;

$$ LANGUAGE plpgsql;
# Function that decrypt data

CREATE OR REPLACE FUNCTION decrypt_data(data BYTEA)
RETURNS TEXT AS $$

DECLARE
key TEXT := 'secret-key';

BEGIN
RETURN pgp_sym_decrypt(data, key);
END;

$$ LANGUAGE plpgsql;

```

Here, we have created two functions: **encrypt\_data()** and **decrypt\_data()**. **encrypt\_data()** function takes plain text as input and returns encrypted binary data. **decrypt\_data()** function takes encrypted binary data as input and returns plain text.

Let us assume that we have a database containing sensitive information such as credit card details, social security numbers, and other personal information. We want to ensure that this information is encrypted at the database level using TDE encryption.

We can use the **pgcrypto** extension in PostgreSQL 15 to implement TDE encryption for our sensitive data. We can create a function that encrypts the sensitive data using a secret key, and another function that decrypts the data using the same key. We can then use these functions to encrypt and decrypt the sensitive data when it is stored and retrieved from the database.

For example, we can create a table **customer\_info** with columns such as **id**, **name**, **email**, **credit\_card**, **ssn**, and so on. We can

then create functions to encrypt the **credit\_card** and **ssn** columns using a secret key:

1. Before we dive into encrypting sensitive data in PostgreSQL, let us start by creating a table to store the sensitive information securely. For example, you can create a table **customer\_info** with columns such as **id**, **name**, **email**, **credit\_card**, **ssn**, and so on.

```
CREATE TABLE customer_info (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    email VARCHAR(50),
    credit_card TEXT,
    ssn TEXT
);
```

2. Then, we can create functions to encrypt and decrypt the sensitive data in your database. For example, we create a function to encrypt credit card information using a secret key:

```
CREATE OR REPLACE FUNCTION encrypt_db_data(plaintext text)
RETURNS bytea AS $$

BEGIN
    RETURN pgp_sym_encrypt(plaintext, 'secret-key');
END;

$$ LANGUAGE plpgsql;
```

This function takes the plain text value of the credit card information as input, encrypts it using the **pgp\_sym\_encrypt()** function, and returns the encrypted binary data.

Similarly, you can create a function to decrypt the credit card information using the same secret key:

```
CREATE OR REPLACE FUNCTION decrypt_db_data (ciphertext text)
RETURNS text AS $$

BEGIN
    RETURN pgp_sym_decrypt(cast(ciphertext as bytea), 'secret-key');
END;
```

```
$$ LANGUAGE plpgsql;
```

This function takes the encrypted binary data of the credit card information as input, decrypts it using the **pgp\_sym\_decrypt()** function, and returns the plain text value.

Replace **secret-key** with a secret key generated in step 3 of this recipe.

3. Next, we will insert sensitive data into the **customer\_info** table using the **encrypt\_db\_data()** function to encrypt the credit card information:

```
INSERT INTO customer_info (name, email, credit_card, ssn)
VALUES ('John Doe', 'johndoe@example.com', encrypt_db_data('1234-
5678-9012-3456'), encrypt_db_data('123-45-6789'));
```

Referring to *Figure 9.3*, this will insert a new record into the **customer\_info** table with the **name**, **email**, **credit\_card**, and **ssn** columns encrypted using the **encrypt\_db\_data()** function:

```
postgres=% select * from customer_info;
 [ RECORD 1 ]
 id | 3
 name | Bugs Bunny
 email | bbunny@pgcryptotest.com
 credit_card | \xc30d04070302bb1525f6d7f8bde62d23c815765d86a7cbd8a6612b477cd161fd5cf1ce5sec10f22c12540c15d25fc0b1fc8149cced08cd12421b00568a67d80cdb405522587fe9de7408
```

*Figure 9.3: Encrypted record*

4. Finally, when we retrieve the sensitive data from the **customer\_info** table, we need to decrypt it using the **decrypt\_db\_data()** function:

```
SELECT id, name, email, decrypt_db_data(credit_card) AS
credit_card, decrypt_db_data(ssn) AS ssn FROM customer_info;
```

Referring to *Figure 9.4*, this will retrieve the **id**, **name**, **email**, **credit\_card**, and **ssn** columns from the **customer\_info** table and decrypt the values in the **credit\_card** and **ssn** columns using the **decrypt\_db\_data()** function:

The screenshot shows a PostgreSQL query window with the following content:

```
3 SELECT id, name, email, decrypt_db_data(credit_card) AS credit_card, decrypt_db_data(ssn) AS ssn FROM customer_info;
```

Data Output Messages Notifications

A table with the following data:

	<b>id</b> [PK] integer	<b>name</b> character varying (50)	<b>email</b> character varying (60)	<b>credit_card</b> text	<b>ssn</b> text
1	3	John Doe	johndoe@example.com	1234-5678-9012-3456	123-45-6789

**Figure 9.4:** Decrypted record

In conclusion, **pgcrypto** is a PostgreSQL extension that provides cryptographic functions to secure data stored in the database. By using pgcrypto, you can easily encrypt sensitive information such as credit card numbers and social security numbers, making it harder for unauthorized users to access or misuse the data.

## PostgreSQL auditing

Auditing feature allows administrators to monitor database activity and track changes made to the database. Auditing can help identify security threats, ensure compliance with regulations, and provide insights into database usage patterns.

PostgreSQL 15 provides several built-in tools for auditing, including logging and event triggers. Logging allows administrators to record all database activity in log files, which can be reviewed to identify unauthorized access or other security issues. Event triggers allow administrators to define custom actions to be taken when specific events occur, such as when a table is modified or when a user logs in.

For example, the following command can be used to enable logging in PostgreSQL 15:

```
ALTER SYSTEM SET log_destination = 'csvlog';
```

This command sets the **log destination** to CSV format, which can be easily read and analyzed using spreadsheet software.

In addition to built-in auditing tools, PostgreSQL 15 also supports third-party auditing extensions, such as **pgaudit** and **pgbadger**. These extensions provide additional features and flexibility for auditing, such as customizable log formats and advanced reporting capabilities.

## Recipe 75: Installing and configuring pgaudit

**pgaudit** is a PostgreSQL extension that provides auditing and logging capabilities for database activities. By installing and configuring **pgaudit**, we can audit specific events and log them for analysis and compliance purposes. The installation process involves creating the necessary tables, triggers, and functions required by **pgaudit**, while the configuration process involves setting the level of logging we want to enable and configure **pgaudit** rules to log specific events.

In this recipe, we will cover the steps to install and configure **pgaudit** for PostgreSQL 15.

1. Download **pgaudit** package from the PostgreSQL official website at [https://download.postgresql.org/pub/repos/yum/15/redhat/rhel-9-x86\\_64/](https://download.postgresql.org/pub/repos/yum/15/redhat/rhel-9-x86_64/) and continue with the installation. We can install this by using the following command:

```
# Install pgaudit package
```

```
yum localinstall pgaudit17_15-1.7.0-1.rhel9.x86_64.rpm
```

2. Configure **pgaudit**. Once we have installed the **pgaudit** package, we need to configure it according to our requirements. The **pgaudit** configuration is stored in the **postgresql.conf** file.

To enable **pgaudit**, we need to add the following lines to the **postgresql.conf** file:

```
# Add pgaudit configuration to postgresql.conf file
```

```
shared_preload_libraries = 'pgaudit'
```

The **shared\_preload\_libraries** setting tells PostgreSQL to load the **pgaudit** library during server start-up.

3. Reload PostgreSQL Server after making changes to the **postgresql.conf** file, we need to reload the PostgreSQL server for the changes to take effect. We can do this by using the following command:

```
# Reload the PostgreSQL server
```

```
SELECT pg_reload_conf();
```

4. Connect to the PostgreSQL database using the **psql** command-line tool and run the following statement to load the **pgaudit** extension:

```
# Create pgaudit extension
```

```
CREATE EXTENSION pgaudit;
```

5. Next is to configure the **pgaudit** rules to log specific events. For example, to log all the statements on the database **postgres**, we can add the following line to the **postgresql.conf** file:

```
# Configure pgaudit rules
```

```
pgaudit.log = 'all'
```

```
pgaudit.log_catalog = on
```

```
pgaudit.log_parameter = on
```

```
pgaudit.log_relation = on
```

```
pgaudit.log_statement = 'on'
```

To permanently set **pgaudit.log = ALL** and all the above **pgaudit** rules without modifying the **postgresql.conf** file, we can use the **ALTER SYSTEM** command to modify the configuration, using the following command:

```
# Configure pgaudit rules
```

```
ALTER SYSTEM SET pgaudit.log = 'ALL';
```

```
ALTER SYSTEM SET pgaudit.log_catalog = on;
```

```
ALTER SYSTEM SET pgaudit.log_parameter = on;
```

```
ALTER SYSTEM SET pgaudit.log_relation = on;
```

```
ALTER SYSTEM SET pgaudit.log_statement = on;
```

This will add the configuration parameter **pgaudit** rules to the database, which is read by PostgreSQL at server start-up. Note that you need to have the **pgaudit** extension installed and loaded in order to use the **pgaudit.log** configuration option.

Alternatively, we can also set **pgaudit.log** to **ALL** and all the **pgaudit** rules at runtime using the **SET** command in a PostgreSQL session.

```
# Configure pgaudit rules
```

```
SET pgaudit.log = 'all'
```

```
SET pgaudit.log_catalog = on  
SET pgaudit.log_parameter = on  
SET pgaudit.log_relation = on  
SET pgaudit.log_statement = on
```

This will set the **pgaudit** rules configuration option to **ALL** for the duration of the current PostgreSQL session.

After executing the **ALTER SYSTEM** command, we need to reload the configuration using the following command:

```
# Reload the PostgreSQL server
```

```
SELECT pg_reload_conf();
```

This will reload the configuration and apply any changes made to it, including the changes made using **ALTER SYSTEM**.

6. Next, run the following SQL statement to verify that **pgaudit** appears in the list of installed extensions.

```
# Configure pgaudit rules
```

```
SELECT * FROM pg_available_extensions WHERE name LIKE  
'%audit%';
```

Referring to *Figure 9.5*, the above command lists the installed **pgaudit** extension:

	name	default_version	installed_version	comment
1	pgaudit	1.7	1.7	provides auditing functionality

*Figure 9.5: Installed pgaudit extension*

If **pgaudit** is not listed, it is likely that the installation was not successful or the extension was not properly enabled.

7. Generate some activity on PostgreSQL server, such as running some queries or executing some DDL statements.

```
# Configure pgaudit rules
```

```
CREATE TABLE test_audt (  
    id serial primary key,  
    name varchar(50)
```

```
);
```

```
# Configure pgaudit rules
```

```
INSERT INTO test_audt (name) VALUES ('Popeye');
```

```
INSERT INTO test_audt (name) VALUES ('Popeye1');
```

8. Check the PostgreSQL log file for audit log entries. The location of the log file can be found by running the following command:

```
# Configure pgaudit rules
```

```
SHOW log_directory;
```

The default log file name is **postgresql-%a.log**. Look for lines in the log file that start with **AUDIT**. These lines will contain information about the audit event, including the time of the event, the user who performed the event, the database and schema involved, and the SQL statement that was executed.

For example, we have executed a **CREATE TABLE** and **INSERT** statement in the public schema of the **postgres** database. As shown in *Figure 9.6*, we might see a log entry like this:

```
2023-05-11 15:51:22.709 +08 [54972] LOG: AUDIT: SESSION,1,1,DDL,CREATE SEQUENCE,SEQUENCE,public.test_audt_id_seq,"CREATE TABLE test_audt ("
2023-05-11 15:51:22.709 +08 [54972] LOG: AUDIT: SESSION,1,1,DDL,CREATE TABLE,TABLE,public.test_audt,"CREATE TABLE test_audt (
2023-05-11 15:51:22.709 +08 [54972] LOG: AUDIT: SESSION,1,1,DDL,CREATE INDEX,INDEX,public.test_audt_pkey,"CREATE TABLE test_audt (
2023-05-11 15:51:22.789 +08 [54972] LOG: AUDIT: SESSION,1,1,DDL,ALTER SEQUENCE,SEQUENCE,public.test_audt_id_seq,"CREATE TABLE test_audt (
2023-05-11 15:52:35.020 +08 [54972] LOG: AUDIT: SESSION,3,1,WRITE,INSERT,TABLE,public.test_audt,INSERT INTO test_audt (name) VALUES ('Popeye');,<none>
2023-05-11 16:07:07.774 +08 [56619] LOG: AUDIT: SESSION,3,1,WRITE,INSERT,TABLE,public.test_audt,INSERT INTO test_audt (name) VALUES ('Popeye1');,<none>
2023-05-11 16:08:21.618 +08 [56619] LOG: AUDIT: SESSION,4,1,READ,SELECT,TABLE,public.test_audt,SELECT * FROM test_audt;,<none>
```

**Figure 9.6:** Logged audit record with pgaudit

The above **pgaudit** log entry indicates the following:

With session 1:

- User initiated a session and created a sequence named **test\_audt\_id\_seq** in the public schema, as part of a **CREATE TABLE** statement for a table named **test\_audt**.
- User initiated a session and created an index named **test\_audt\_pkey** on the **test\_audt** table in the public schema, as part of a **CREATE TABLE** statement.
- User initiated a session and altered the **test\_audt\_id\_seq** sequence in the public schema, as part of a **CREATE TABLE** statement for a table named **test\_audt**.

With session 3:

- User performed a **WRITE** operation by inserting a row into the **test\_audit** table in the public schema, with the value **Popeye** in the name column.
- User performed a **WRITE** operation by inserting a row into the **test\_audit** table in the public schema, with the value **Popeye1** in the name column.

With session 4:

- User initiated a session and performed a **READ** operation by selecting all rows from the **test\_audit** table in the public schema. No parameters or values were logged in this case.

Finally, we successfully verified the **pgaudit** installation and configuration by running some test queries and checking the **pgaudit** logs in our previous steps. Let us now head over to next recipe to work around with PostgreSQL trigger for **Audit** logs.

## **Recipe 76: Using audit log with PostgreSQL trigger**

PostgreSQL is a popular open-source **relational database management system (RDBMS)** that provides several built-in features to audit database activities. One such feature is the use of triggers to track database changes and record them in the audit log.

In this recipe, we will dive into the use of audit log with PostgreSQL trigger to track the changes made to a specific table. We will provide a detailed example scenario to make it easy to understand.

Before starting, we need to make sure that you have the following prerequisites:

- PostgreSQL 15 or later installed on your system.
  - A database and a table to work with.
  - Sufficient privileges to create triggers and tables.
1. The first step is to create a table to store the audit log information. The table should have columns to store the

user ID, timestamp, table name, operation type, and old and new values.

We can use the following SQL command to create the audit log table:

```
# Create Audit-log tracking Table
```

```
CREATE TABLE audit_log_tracker (
    id SERIAL PRIMARY KEY,
    user_id VARCHAR(50) NOT NULL,
    table_name VARCHAR(50) NOT NULL,
    operation_type CHAR(1) NOT NULL,
    old_value JSONB,
    new_value JSONB,
    created_at TIMESTAMP NOT NULL DEFAULT NOW()
);
```

2. The next step is to create a trigger function that will be called every time a change is made to the table. The trigger function should insert a new row into the audit log table with the necessary information.

We can use the following SQL command to create the trigger function:

```
# Create trigger function for tracking change
```

```
CREATE OR REPLACE FUNCTION audit_log_tracker()
RETURNS TRIGGER AS $$

DECLARE
    operation_type CHAR(1);

BEGIN
    IF TG_OP = 'INSERT' THEN
        operation_type := 'I';
    ELSIF TG_OP = 'UPDATE' THEN
        operation_type := 'U';
    ELSIF TG_OP = 'DELETE' THEN
        operation_type := 'D';
    ELSE
        RAISE EXCEPTION 'Unknown TG_OP: "%". Should be INSERT,
                        UPDATE, or DELETE.', TG_OP;
    END IF;
    -- Insert the audit log entry here
    -- ...
END;
```

```

    END IF;
    INSERT INTO audit_log_tracker (user_id, table_name,
operation_type, old_value, new_value)
        VALUES (CURRENT_USER, TG_TABLE_NAME, operation_type,
to_jsonb(old), to_jsonb(new));
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

```

The trigger function **audit\_log\_tracker()** takes no arguments and returns a trigger. It starts by declaring a variable **operation\_type** which will hold the type of operation that triggered the function. The **IF** statement checks the value of the **Trigger Operation (TG\_OP)** variable to determine the type of operation. If it is an **INSERT** operation, the value of **operation\_type** is set to **I**, if it is an **UPDATE** operation, the value is set to **U**, and if it is a **DELETE** operation, the value is set to **D**. If none of these conditions are met, an exception is raised.

The function then inserts a new record into the **audit\_log\_tracker** table, with the **CURRENT\_USER** as the **user\_id**, **TG\_TABLE\_NAME** as the **table\_name**, **operation\_type** as the operation type, old and new as **JSONB** objects representing the old and new values of the row respectively.

Finally, the function returns the **NEW** value, which is the updated row after the operation is complete.

3. The final step is to create a trigger on the table that you want to track changes for. The trigger should be fired for every insert, update, or delete operation on the table. We use the following SQL command to create the trigger:

*# Configure trigger for auditing*

```
CREATE TRIGGER audit_table_change_tracker
```

```
AFTER INSERT OR UPDATE OR DELETE
```

```
ON <Table_Name>
```

```
FOR EACH ROW
```

```
EXECUTE FUNCTION audit_log_tracker();
```

Let us consider an example to understand how to use audit log with PostgreSQL trigger. Suppose we have a table called **employees** with the following columns:

- **id** (serial)
- **name** (varchar)
- **age** (integer)
- **email** (varchar)
- **salary** (numeric)

We want to track all the changes made to employees table and record them in the audit log table. To achieve this, we can refer back to *steps 1 and 2* of this recipe:

1. Create a trigger to audit the employee table using the following SQL command:

```
# Create trigger for auditing
```

```
CREATE TRIGGER audit_table_change_tracker
```

```
AFTER INSERT OR UPDATE OR DELETE
```

```
ON employees
```

```
FOR EACH ROW
```

```
EXECUTE FUNCTION audit_log_tracker_trigger();
```

Now, whenever a new row is inserted, updated or deleted from the **employees** table, the trigger function **audit\_log\_tracker\_trigger** will be called, which will insert a new row into the **audit\_log\_tracker** table with the user ID, table name, operation type, old value and new value.

For example, we have the following data in the **employees** table, as shown in [Figure 9.7](#):

Data Output    Messages    Notifications					
	<b>id</b> [PK] integer	<b>name</b> character varying (30)	<b>age</b> integer	<b>email</b> character varying (30)	<b>salary</b> numeric
1	1	Bugs Bunny	30	bbuny@triggaudit.com	5000.00
2	2	Popeye	35	popeye@triggaudit.com	6000.00

**Figure 9.7:** Test data for audit log record

Now, if we execute the following SQL commands on the employees table:

```
# Update employee table
```

```
UPDATE employees SET salary = 6500.00 WHERE id = 1;
```

```
UPDATE employees SET salary = 7000.00 WHERE id = 2;
```

```
DELETE FROM employees WHERE ID = 2;
```

Referring to [Figure 9.8](#), The **audit\_log\_tracker** table will have the following records:

Data Output Messages Notifications					
	id	user_id	table_name	operation_type	old_value
	[PK] integer	character varying	character varying	character(1) jsonb	new_value
1	1	postgres	employees	U	{"id": 1, "age": 30, "name": "Bugs Bunny", "email": "bbunny@triggaudit.com", "salary": 5000.00}
2	2	postgres	employees	U	{"id": 2, "age": 35, "name": "Popeye", "email": "popeye@triggaudit.com", "salary": 6000.00}
3	3	postgres	employees	D	{"id": 2, "age": 35, "name": "Popeye", "email": "popeye@triggaudit.com", "salary": 7000.00}

**Figure 9.8:** Traced audit log record

In conclusion, using audit logging with PostgreSQL triggers can be a powerful way to track changes to your database and maintain data integrity. By implementing the steps outlined in this recipe, you can create a trigger that logs changes to a separate audit table and enables the **pgaudit** extension to provide additional auditing capabilities. With this setup, you can easily monitor and track changes to your database, ensuring that your data remains accurate and secure.

## Recipe 77: Working with **log\_statement**

**log\_statement** is a PostgreSQL configuration parameter that controls whether SQL statements executed by clients are logged or not. It can be set to three possible values, they are as follows:

- **none (off)**: No logging of SQL statements.
- **ddl**: Logging of only **data definition language (DDL)** statements, such as **CREATE**, **ALTER**, and **DROP**.
- **mod**: Logging of only data modification statements, such as **INSERT**, **UPDATE**, and **DELETE**.
- **all**: Logging of all SQL statements, including both DDL and DML statements.

The **log\_statement** parameter controls whether SQL statements are logged or not. This recipe will demonstrate how to configure and use **log\_statement** in PostgreSQL 15. Moreover, we will enable **log\_statement** for a database and perform some SQL statements to generate log messages.

1. The first step to configure **log\_statement** is to edit the PostgreSQL configuration file **postgresql.conf**. This file contains the configuration parameters for the database server. We can open the file using a text editor and locate the following line:

```
#log_statement = 'none'
```

Uncomment the line by removing the **#** and set the value to **all** to enable logging of all SQL statements:

```
log_statement = 'all'
```

To change the value of **log\_statement** without editing the PostgreSQL configuration file, we can use the **ALTER SYSTEM** command:

```
ALTER SYSTEM SET log_statement = 'all';
```

Save the changes and restart the PostgreSQL server to apply the new configuration.

Referring to [Figure 9.9](#), The **pg\_settings** table has been populated with all the changes made in step 1 of this recipe:

```
postgres=# SELECT name,setting FROM pg_settings WHERE name LIKE 'log_statement%';
      name       | setting
-----+-----
log_statement    | all
log_statement_sample_rate | 1
log_statement_stats | off
```

**Figure 9.9:** *log\_statement to all - configuration*

2. We can now perform some SQL statements to generate log messages. In this recipe, we will create a new table and insert some records into it.

```
# Create a new table
```

```
CREATE TABLE characters (
    id SERIAL PRIMARY KEY,
```

```

    name TEXT NOT NULL
);
# Insert some records
INSERT INTO characters (name) VALUES ('Mickey Mouse');
INSERT INTO characters (name) VALUES ('Daffy Duc');

```

3. We can view the log messages generated by the SQL statements by querying the PostgreSQL log files. By default, the log files are located in the **log** subdirectory of the PostgreSQL data directory.

```
tail -f /var/lib/pgsql/15/data/log/postgresql-<day>.log
```

Referring to *Figure 9.10*, we use the tail command to view the contents of the log file. The -f option will cause the command to follow the log file and display new log messages as they are written. Within the log file, we can see the SQL statements executed by the clients, along with other information such as the user, database, and timestamp. Please refer to the following figure:

```

[root@pgsrvdev log]# tail -f /var/lib/pgsql/15/data/log/postgresql-Sun.log
2023-05-14 09:53:27.341 +08 [89450] LOG: parameter "log_statement" changed to "all"
2023-05-14 09:53:29.442 +08 [89487] LOG: statement: SELECT name,setting FROM pg_settings WHERE name LIKE 'log_statement';
2023-05-14 09:54:39.396 +08 [89452] LOG: checkpoint starting: time
2023-05-14 09:54:42.313 +08 [89452] LOG: checkpoint complete: wrote 32 buffers (0.2%); 0 WAL file(s) added, 0 removed, 0 recycled; write=2.913 s, sync=0.002 s,
2023-05-14 09:55:40.252 +08 [89487] LOG: statement: CREATE TABLE characters (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL
);
2023-05-14 09:55:55.764 +08 [89487] LOG: statement: INSERT INTO characters (name) VALUES ('Mickey Mouse');
2023-05-14 09:55:55.765 +08 [89487] LOG: statement: INSERT INTO characters (name) VALUES ('Daffy Duc');

```

**Figure 9.10:** *log\_statement to all - logs*

As shown in *Figure 9.10*, we can see the **CREATE TABLE** statement, along with the two **INSERT** statements, logged in the log file.

In the upcoming steps, we will explore the different values that can be used for the **log\_statement** parameter in PostgreSQL 15. This parameter allows you to control what types of SQL statements and activities are recorded in the PostgreSQL logs.

1. For the possible value mod, edit the PostgreSQL configuration file **postgresql.conf**, locate the line containing the **log\_statement** parameter and change its value to **mod** or add it if it does not exist.

```
log_statement = 'mod'
```

To change the value of **log\_statement** without editing the PostgreSQL configuration file, we can use the **ALTER SYSTEM** command:

```
ALTER SYSTEM SET log_statement = 'mod';
```

Save the changes and reload the configuration file for the change to take effect.

Referring to *Figure 9.11*, the **pg\_settings** table has been populated with all the changes made in step 1 of this recipe:

```
postgres=# SELECT name,setting FROM pg_settings WHERE name LIKE 'log_statement%';
          name           | setting
-----+-----
log_statement      | mod
log_statement_sample_rate | 1
log_statement_stats    | off
(3 rows)
```

*Figure 9.11: log\_statement to mod - configuration*

2. We can now perform some SQL statements to generate log messages. We will create a new table and insert some records.

```
CREATE TABLE characters_new (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL
);
# Insert some records
INSERT INTO characters_new (name) VALUES ('Mickey Mouse');
INSERT INTO characters_new (name) VALUES ('Daffy Duc');
```

3. Check the PostgreSQL log files to verify that the statement was logged. The location of the log files may vary depending on your operating system, but typical locations include **/var/lib/pgsql/<version>/data/log**, look for a log entry containing the statement we executed:

```
tail -f /var/lib/pgsql/15/data/log/postgresql-<day>.log
```

After the configuration was reloaded, we see once **CREATE** table and two **INSERT** statements executed against the

**characters\_new** table, and both of them were logged in the PostgreSQL log file with their respective SQL statements, along with their timestamps and other relevant information.

Referring to [Figure 9.12](#), when **log\_statement='mod'**, PostgreSQL will log all SQL statements that modify data, including **INSERT**, **UPDATE**, and **DELETE** statements, but not **SELECT** statements. The log entries will include the SQL statement, the time the statement was executed, and other relevant information such as the user who executed the statement and the database it was executed against. Please refer to the following figure:

```
[root@pgsrdev log]# tail -f /var/lib/pgsql/15/data/log/postgresql-Sun.log
2023-05-14 11:18:05.086 +08 [89487] LOG:  statement: ALTER SYSTEM SET log_statement TO 'mod';
2023-05-14 11:18:08.278 +08 [89487] LOG:  statement: SELECT pg_reload_conf();
2023-05-14 11:18:08.278 +08 [89450] LOG:  received SIGHUP, reloading configuration files
2023-05-14 11:18:08.279 +08 [89450] LOG:  parameter "log_statement" changed to "mod"
2023-05-14 11:22:44.570 +08 [89487] LOG:  statement: CREATE TABLE characters_new (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL
);
2023-05-14 11:22:55.083 +08 [89487] LOG:  statement: INSERT INTO characters_new (name) VALUES ('Mickey Mouse');
2023-05-14 11:22:55.084 +08 [89487] LOG:  statement: INSERT INTO characters_new (name) VALUES ('Daffy Duck');
```

**Figure 9.12: log\_statement to mod - logs**

In the upcoming steps, we will explore the **log\_statement='ddl'** values that can be used for the **log\_statement** parameter in PostgreSQL 15. This parameter allows you to control what types of SQL statements and activities are recorded in the PostgreSQL logs.

With **log\_statement='ddl'**, the PostgreSQL server will log all SQL statements that modify database schema, such as **CREATE**, **ALTER**, and **DROP** statements.

1. For the possible value **ddl for the log\_statement variable**, edit the PostgreSQL configuration file **postgresql.conf**. Locate the line containing the **log\_statement** parameter and change its value to **ddl** or add it if it does not exist.

```
log_statement = 'ddl'
```

To change the value of **log\_statement** without editing the PostgreSQL configuration file, we can use the **ALTER SYSTEM** command.

```
ALTER SYSTEM SET log_statement = 'ddl';
```

Save the changes and reload the configuration file for the change to take effect.

Referring to [Figure 9.13](#), the **pg\_settings** table has been populated with all the changes made in step 1 of this recipe:

```
postgres=# SELECT name,setting FROM pg_settings WHERE name LIKE 'log_statement%';
          name           | setting
-----+-----
log_statement      | ddl
log_statement_sample_rate | 1
log_statement_stats   | off
(3 rows)
```

**Figure 9.13:** *log\_statement to ddl - configuration*

2. Execute some SQL statements that modify database schema, such as creating or dropping tables or indexes. For example:

```
CREATE TABLE characters_new (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL
);
# Insert some records
INSERT INTO characters_new (name) VALUES ('Mickey Mouse');
INSERT INTO characters_new (name) VALUES ('Daffy Duc');
```

3. Check the PostgreSQL log file to see the logged SQL statements. By default, the PostgreSQL log file is located in the data directory, under the log subdirectory. You can use a text editor or a command-line tool such as less or tail to view the log file. For example:

```
tail -f /var/lib/pgsql/15/data/log/postgresql-<day>.log
```

Referring to [Figure 9.14](#), in the log file, you can see log entries for the SQL statements that modified the database schema, along with their timestamps and other relevant information. For example:

```
[root@pgsrpdev log]# tail -f /var/lib/pgsql/15/data/log/postgresql-Sun.log
2023-05-14 11:38:53.123 +08 [89487] LOG: statement: ALTER SYSTEM SET log_statement TO 'ddl';
2023-05-14 11:38:57.671 +08 [89450] LOG: received SIGHUP, reloading configuration files
2023-05-14 11:38:57.672 +08 [89450] LOG: parameter "log_statement" changed to "ddl"
2023-05-14 11:39:25.521 +08 [89487] LOG: statement: CREATE TABLE characters_ddl_test (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL
);
2023-05-14 11:39:39.705 +08 [89452] LOG: checkpoint starting: time
```

**Figure 9.14:** *log\_statement to ddl - logs*

In the above log output, the **CREATE TABLE** statement is logged because it is a DDL statement that creates a new table named **characters\_ddl\_test**. The **INSERT** statements are not logged because they are not DDL statements, but rather **data manipulation language (DML)** statements that modify data in the database.

- After we have finished using **log\_statement**, we can disable it by editing the **postgresql.conf** file again and setting the value of **log\_statement** back to **none**.

```
#log_statement = 'none'
```

Save the changes and reload the configuration file for the change to take effect.

The **log\_statement** parameter in PostgreSQL 15 provides a way to log SQL statements executed by clients. By enabling **log\_statement** and querying the PostgreSQL log files, we can gain insight into the behaviour of the database and troubleshoot issues. It is important to disable **log\_statement** when it is no longer needed to avoid excessive log file growth and potential performance issues.

## LDAP authentication

**Lightweight directory access protocol (LDAP)** is a standard protocol used for accessing and maintaining distributed directory information services over an IP network. It is a lightweight and efficient way to manage user authentication and authorization in a network environment. LDAP can be used as an authentication method in PostgreSQL to allow users to authenticate using their network credentials instead of a separate PostgreSQL user account. By integrating LDAP with PostgreSQL, you can simplify

user management by centralizing user account information and password policies.

## Recipe 78: Getting ready with LDAP authentication

This recipe provides insights into the configuration steps and prerequisites necessary to seamlessly integrate LDAP authentication into PostgreSQL. By doing so, users can leverage their network credentials for authentication, streamlining the process and centralizing user account information and password policies. This not only enhances security but also simplifies user management in a networked environment, making it an invaluable resource for those seeking to optimize authentication methods in their PostgreSQL setup.

1. To enable LDAP authentication support in PostgreSQL requires the **ldap-utils** modules. You can install these modules using the following command:

```
yum install ldap-utils
```

2. Configure PostgreSQL to use LDAP authentication, edit the **pg\_hba.conf** file located in the PostgreSQL data directory (usually **/var/lib/postgresql/data** on Linux systems) and add the following command:

```
# LDAP Authentication
```

```
host all all <ldap-srv-ip-add>/32 ldap ldapserver=<ldap-srv-name> ldapport=389 ldapbinddn="<bind-dn>" ldapprefix="cn=" ldapsuffix=",<ldap-base-dn>"
```

Replace **<ldap-srv-ip-add>** with the IP address of the LDAP server, **<ldap-srv-name>** with the name of the LDAP server, **<bind-dn>** with the distinguished name of a user that has read access to the LDAP server, and **<ldap-base-dn>** with the base DN of the LDAP directory.

3. After editing the **pg\_hba.conf** file, we need to restart PostgreSQL for the changes to take effect. We can do this using the following command.

```
systemctl restart postgresql-15.service
```

4. Create a PostgreSQL user that corresponds to the LDAP user account, we can create a PostgreSQL user that corresponds to the LDAP user account by running the following command:

```
CREATE USER <ldap-user> LOGIN;
```

Replace **<ldap-user>** with the username of the LDAP user account.

5. Grant database privileges to the PostgreSQL user by running the following command:

```
GRANT ALL PRIVILEGES ON DATABASE <database-name> TO  
<ldap-user>;
```

Replace **<database-name>** with the name of the PostgreSQL database you want to grant access to, and **<ldap-user>** with the username of the LDAP user account. Alternatively, we can grant privileges on specific schemas and/or tables in the database using the **GRANT** command.

6. Test LDAP authentication by connecting to the PostgreSQL instance using an LDAP user account. For example, you can use the **psql** command-line tool to connect to the database as follows:

```
psql -h <postgres-host> -U <ldap-user> <database-name>
```

Replace **<postgres-host>** with the hostname or IP address of the PostgreSQL server, **<ldap-user>** with the username of an LDAP user account, and **<database-name>** with the name of the PostgreSQL database you want to connect to.

## Conclusion

In conclusion, server controls and auditing are crucial elements in ensuring the security of your PostgreSQL 15 database. By implementing these controls and techniques, you can ensure that your data is protected against unauthorized access, manipulation, and theft. We hope that this chapter has provided you with a clear understanding of the fundamental concepts involved in authentication and encryption, as well as the practical solutions for user and group control, client connection encryption,

and database authentication through SSL. With this knowledge, you can implement the necessary security measures to protect your PostgreSQL 15 database.

In the upcoming chapter, we will shift our focus to a crucial aspect of database management - backup and recovery. We will introduce the concept of database backup, guide you through backup planning, explore SQL dumps, file system-level backups, and ways to improve backup performance. Furthermore, we will delve into various backup tools and present real-world backup use cases to prepare you for comprehensive database backup strategies.

## **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)



# CHAPTER 10

# Backup

## Introduction

In this chapter, we will delve into the world of database backup in PostgreSQL. As data is a critical asset for any organization, it is essential to have a robust backup strategy in place to ensure the availability and integrity of the database. This chapter will provide you with a comprehensive understanding of database backup concepts, techniques, and tools, accompanied by practical examples to guide you through the process. By the end of this chapter, you will possess the knowledge and skills necessary to perform efficient backups and recoveries, enhancing the reliability and security of your PostgreSQL databases.

## Structure

In this chapter, we will cover the following topics:

- Introduction to database backup
- Exploring SQL dump and file system level backup
- Exploring backup tools
- Improving backup performance

## Objectives

The primary objective of this chapter is to provide a comprehensive understanding of backup techniques within PostgreSQL. Our focus encompasses several key objectives:

Firstly, we aim to elucidate the significance of database backups, elucidating their pivotal role in data protection and disaster recovery. We delve into how these backups enhance the overall reliability and security of a PostgreSQL database.

Furthermore, we explore diverse backup strategies and techniques, such as full backups, incremental backups, and point-in-time recovery. This involves a thorough examination of their respective advantages, limitations, and the process of selecting the most appropriate strategy based on distinct scenarios.

To facilitate a practical understanding, we introduce the backup tools and utilities available in PostgreSQL, including but not limited to **pg\_dump**, **pg\_dumpall**, and **pg\_basebackup**. Through this, readers become acquainted with the features, options, and effective utilization of these tools for creating backups.

Finally, the chapter concludes with a hands-on approach, allowing readers to actively engage in backup procedures within PostgreSQL. Step-by-step instructions and real-world examples guide the reader through the process of backing up databases, considering different availability scenarios.

## Introduction to database backup

Database backup is an essential component of any database management system, including PostgreSQL. Backups serve as a means of protection against potential data loss, which can occur due to various reasons such as system crashes, hardware failures, and other disasters. PostgreSQL offers several types of backup options, including logical, physical, and continuous archiving backups, to create backups of databases.

Logical backups create a text file containing SQL commands that can be used to recreate the database objects and data, while physical backups create a binary copy of the database files at the file system level. Continuous archiving backups create a continuous stream of archived WAL files that can be used to restore the database to a specific point in time. PostgreSQL provides backup tools such as **pg\_dump**, **pg\_basebackup**, **pg\_restore**, and **pg\_ctl**, which allow users to create and restore backups with ease.

For example, if a user has a database named **products**, they can use **pg\_dump** to create a logical backup of the database by running the command **pg\_dump -U postgres products | products\_backup.sql**

in the command line. This creates a backup file named **products\_backup.sql** that contains the SQL commands to recreate the database objects and data. Overall, understanding and implementing database backups is an essential part of database management to protect data and ensure its safety.

## Exploring SQL dump and file system level backup

SQL dump and file system level backup are two popular backup methods used in PostgreSQL. SQL dump backup involves creating a logical backup of the database in SQL format, while file system level backup involves copying the entire PostgreSQL data directory to a backup location.

Exploring SQL dump backup involves understanding its advantages and disadvantages. SQL dump backups are portable and can be restored on any platform that supports PostgreSQL, making them ideal for migrating data between different PostgreSQL versions or platforms. However, they can be slower to restore than file system level backups, especially for large databases.

Exploring file system level backup involves understanding its advantages and disadvantages. File system level backups are faster to restore than SQL dump backups and provide a complete backup of the database cluster. However, they are platform-specific and cannot be restored on different platforms or PostgreSQL versions. Additionally, file system level backups can only be used for point-in-time recovery, meaning that they cannot be used to restore individual database objects.

Sarah is a database administrator for a financial company that uses PostgreSQL. Sarah is responsible for creating backups of the database and decides to explore SQL dump and file system level backup methods.

Sarah begins by creating a SQL dump backup of the database. She uses the **pg\_dump** tool provided by PostgreSQL to create a logical backup in SQL format. Sarah tests the backup by restoring it to a test environment, ensuring that the backup is reliable and usable in case of data loss. She notes that SQL dump backups are portable, making them ideal for migrating data between different PostgreSQL versions or platforms.

Sarah then explores file system level backup. She uses the **pg\_basebackup** tool provided by PostgreSQL to copy the entire

PostgreSQL data directory to a backup location. Sarah tests the backup by restoring it to a test environment, ensuring that the backup is reliable and usable in case of data loss. She notes that file system level backups are faster to restore than SQL dump backups and provide a complete backup of the database cluster.

In conclusion, exploring SQL dump and file system level backup methods in PostgreSQL involves understanding their advantages and disadvantages. SQL dump backups are portable and ideal for migrating data between different PostgreSQL versions or platforms, while file system level backups are faster to restore and provide a complete backup of the database cluster. By exploring both backup methods, you can choose the one that best suits your database size, availability, and recovery time requirements.

## Recipe 79: Working with logical backup

Logical backups in PostgreSQL provide a convenient way to backup and restore databases using a text-based format that is portable and easy to understand. In this recipe, we will explore the process of working with logical backups in PostgreSQL:

**Prerequisite:** Ensure that PostgreSQL is installed and running, and have a specific database set up before proceeding with working on logical backups.

### Set up the backup directory:

1. Create a directory on your storage medium to store the backup files. For example, create a folder named **pg\_backup** on an external hard drive.
2. Create a logical backup, open a terminal or command prompt and use the **pg\_dump** command to create a logical backup of your database. The basic syntax is as follows:

```
pg_dump -U <username> -d <db_name> -F p -f <backup_file.sql>
```

Let us break down the details of each option used in the above command:

- **-U <username>**: Specifies the username used to connect to the PostgreSQL database. Replace **<username>** with the actual username you want to use for the backup operation.
- **-d <db\_name>**: Specifies the name of the database to be backed up. Replace **<database\_name>** with the name of

the database you want to backup.

- **-F p**: Specifies the output file format. In this case, **p** indicates the plain-text format. This format produces a human-readable SQL script that can be easily understood and modified if needed.
- **-f <backup\_file.sql>**: Specifies the filename and path for the backup file to be created. Replace **<backup\_file.sql>** with the desired name and location for your backup file.

Replace **<username>** with your PostgreSQL username, **<db\_name>** with the name of the database you want to backup, and **<backup\_file.sql>** with the desired name for your backup file.

PostgreSQL will prompt you for your password, wait for the backup process to complete. This may take some time depending on the size of your database.

### Scenario: Working with logical backups

Imagine a software developer working on a web application that utilizes a PostgreSQL database. The application has been running smoothly for months, but recently made some significant changes to the database schema and want to ensure to have a backup before deploying the changes to the production environment.

1. Create a logical backup, open a terminal on your development machine and run the following command to create a logical backup of the database:

```
# Backup PostgreSQL without Compression
```

```
pg_dump -U postgres -d pg -F p -f Logical_Bkp_202305.sql
```

```
# Alternatively, Backup PostgreSQL with native Compression
```

```
pg_dump -U postgres -Fc -d pg Logical_Bkp_202305
```

This command connects to the database named **pgs** using the username **postgres** and creates a logical backup in a file named **Logical\_Bkp.sql**.

2. Enter your PostgreSQL password when prompted.
3. Wait for the backup process to complete.
4. Once the backup process is finished, store the **Logical\_Bkp.sql** file in a safe location. We can choose to use a cloud storage service, an external hard drive, or another secure location.

Ensure that the backup file is easily accessible and properly labeled for future reference.

5. Proceed with deploying the new schema changes to the production environment, using appropriate migration or deployment techniques. Test the application thoroughly to ensure that the changes have been implemented correctly.

By following these steps, we can create a logical backup of your PostgreSQL database, ensuring that you have a reliable copy of your data before making significant changes. In case any issues arise during the deployment process, we can always rely on the logical backup to restore the database to its previous state.

## Recipe 80: Working with physical backup

Physical backups in PostgreSQL involve taking a snapshot of the entire database directory, including data files, transaction logs, and configuration files. In this recipe, we will explore the process of working with physical backups in PostgreSQL.

**Prerequisite:** Ensure that PostgreSQL is installed and running, and have a specific database set up before proceeding with working on logical backups.

### Preparing for a physical backup:

1. Determine the target directory to store the backup. Ensure that there is enough disk space to accommodate the backup.
2. Stopping PostgreSQL service:
  - a. Stop the PostgreSQL service to ensure a consistent backup.

```
systemctl stop postgresql-15.service
```

3. Creating a physical backup:

- a. Use a file system-level backup tool or utility to create a snapshot or copy of the entire PostgreSQL data directory.

Following are the commonly available tools or options for PostgreSQL physical backups:

Tools/Options	Description
<b>rsync</b>	Although not a PostgreSQL-specific tool, rsync is a widely used utility for efficiently synchronizing files and directories between different locations. It can be utilized for physical backups by copying the PostgreSQL data directory to a backup location.  The basic rsync command for backing up a PostgreSQL data directory would be:

	<pre>rsync -avz --progress /path/to/bkp/location/</pre> <p>This command recursively copies the files and directories from the PostgreSQL data directory to the backup location while preserving file permissions, ownership, and timestamps.</p>
<b>pg_basebackup</b>	<p>A built-in tool provided by PostgreSQL for taking base backups at the file level. It performs a streaming backup of the entire database cluster, including data files and transaction logs.</p> <p>To use <b>pg_basebackup</b>, you would typically execute a command like.</p> <pre>pg_basebackup -D /path/to/bkp/location -Ft -X stream</pre> <p>This command instructs <b>pg_basebackup</b> to stream the backup in the tar format (-Ft) to the specified backup location (-D). The -X stream option ensures that the backup is streamed rather than written to a file on the server.</p>

**Table 10.1:** PostgreSQL backup tool options

The location of the data directory varies depending on the operating system and PostgreSQL installation. The following command will identify the data directory path:

```
select setting from pg_settings where name = 'data_directory';
```

4. Ensure that the backup captures all the necessary files, including data files, transaction logs, and configuration files.

### Scenario: Physical backups:

Imagine a database administrator for a financial company that heavily relies on a PostgreSQL database. Due to regulatory compliance requirements, DBA needs to regularly perform physical backups of the database to ensure data integrity and disaster recovery.

### Preparing for a physical backup:

Before proceeding with the steps for recipe scenario it is crucial to assess the available disk space to ensure its sufficiency for storing the backup. Additionally, identify a suitable location or external storage medium where the backup will be stored. For this recipe, we use **/pg\_backup/Physical\_bkp** file system as an example of a designated storage location. This preliminary evaluation ensures a smooth and successful execution of the subsequent steps in the physical backup process:

#### 1. Stopping PostgreSQL service:

- a. Stop the PostgreSQL service to ensure a consistent backup.

```
Systemctl stop postgresql-15.service
```

## 2. Creating a physical backup:

- a. Utilize a file system-level backup tool or utility, such as **rsync**, **tar**, or a specialized backup solution, to create a copy of the entire PostgreSQL data directory.
- b. Run the command specific to your chosen backup tool. For example, using **rsync**:

```
rsync -avz --progress /var/lib/pgsql/15/data /pg_backup/Physical_bkp
```

Here is a brief description of the execution with **rsync**:

- The command initiates the **rsync** utility.
- **-a** enables the archive mode, which preserves file attributes such as permissions, ownership, and timestamps.
- **-v** activates verbose mode, providing detailed information about the files being copied.
- **-z** enables compression during the transfer to reduce network bandwidth usage.
- **--progress** displays the progress of the backup process, showing the transfer rate and estimated completion time.
- **/var/lib/pgsql/15/data** represents the source directory where the PostgreSQL data files are located. You may need to adjust the path based on your PostgreSQL installation.
- **/pg\_backup/Physical\_bkp** is the destination directory where the backup files will be stored. Modify the path according to your desired backup location.

After execution, we will have a copy of the PostgreSQL data directory in the specified backup location. This backup can be used for restoring the database in case of data loss or for setting up a new PostgreSQL instance with the same data.

In addition to using **rsync**, we can also create a physical backup of your PostgreSQL database using the **pg\_basebackup** utility.

Open a terminal or command prompt and run the following command to create a physical backup using **pg\_basebackup**:

```
pg_basebackup -U postgres -D /pg_backup/phy_bkp -Ft -z -P -Xs -c fast
```

Here is a brief description of the execution with **pg\_basebackup**:

- The command initiates the **pg\_basebackup** utility.
- **-U postgres** specifies the PostgreSQL user (**postgres**) to connect and perform the backup.
- **-D /pg\_backup/phy\_bkp** sets the destination directory where the backup files will be stored. Adjust the path as per your desired backup location.
- **-Ft** specifies the backup format as **tar (t)**, which is a human-readable format that allows easy inspection of backup contents.
- **-z** enables **gzip** compression during the backup process to reduce the size of the backup files.
- **-P** displays the progress of the backup, providing information on the transfer rate and estimated completion time.
- **-Xs** sets the backup mode to stream.
- **-c fast** Enables fast checkpointing, this mode performs checkpoint writes quickly, which can result in faster backup completion times.

When this command is executed, **pg\_basebackup** connects to the PostgreSQL server using the specified user (**postgres**), retrieves the data files, and streams them directly to the destination directory (**/pg\_backup/phy\_bkp**) in a compressed tar format.

After the execution, we will have a physical backup of the PostgreSQL database in the specified location. This backup can be used to restore the database or set up a new PostgreSQL instance with the same data.

## Recipe 81: Automating backup

Automating backups in PostgreSQL involves setting up a system to perform regular backups without manual intervention. By automating the backup process, you can ensure that your PostgreSQL database is regularly backed up, reducing the risk of data loss and simplifying your backup workflow.

The process typically involves creating a backup script that contains the necessary commands to perform the backup, such as using the **pg\_dump** utility. This script is then scheduled to run at specific intervals using cron, a time-based job scheduler. With the backup script and cron in place, the system will automatically execute the backup

script according to the defined schedule, creating backups of your PostgreSQL database without requiring manual effort.

This automation provides peace of mind, knowing that your critical data is consistently protected and easily recoverable in the event of any unforeseen issues or disasters. In this recipe, we will explore how to automate backups in PostgreSQL using a shell script and cron job scheduling.

Imagine a DBA working for an e-commerce company that relies on a PostgreSQL database to store customer and order data. To ensure data safety and minimize downtime, DBA wants to automate the backup process for the database:

1. Create a backup script with a text editor or code editor and create a new file and save it with a suitable name, such as **backup\_script.sh**.
  - a. Write a shell script to perform the backup operations. Here is an example script to get you started:

```
#!/bin/bash
# PostgreSQL backup script
# Variables
PG_USER="postgres"
PG_DB="postgres"
BACKUP_DIR="/pg_backup/Daily_backup"
BACKUP_FILE="$BACKUP_DIR/backup_$(date +%Y%m%d%H%M%S).sql"
# Perform the backup
pg_dump -U $PG_USER -d $PG_DB -F p -f $BACKUP_FILE
# Optional: Compress the backup file
gzip $BACKUP_FILE
# Optional: Remove backups older than X days
find $BACKUP_DIR -name "*.gz" -mtime +7 -delete
```

Here is a brief description of the execution of **backup\_script.sh** script:

- The script is written in the bash scripting language and starts with the shebang **#!/bin/bash**, indicating that it should be interpreted using the bash shell.
- The script sets up variables:
  - **PG\_USER**: Specifies the PostgreSQL username to use for the backup.

- **PG\_DB**: Specifies the name of the database to be backed up.
- **BACKUP\_DIR**: Specifies the directory where the backups will be stored.
- **BACKUP\_FILE**: Defines the filename for the backup, using the current date and time.
- The backup operation is performed using the **pg\_dump** command with the following options:
  - **-U \$PG\_USER**: Specifies the PostgreSQL username for the connection.
  - **-d \$PG\_DB**: Specifies the database to be backed up.
  - **-F p**: Sets the backup format to plain SQL script.
  - **-f \$BACKUP\_FILE**: Specifies the output file for the backup.

**Optional:** After the backup is created, the script compresses the backup file using the **gzip** command. The script includes a line that uses the find command to locate and remove backup files older than 7 days (**-mtime +7**) in the specified **\$BACKUP\_DIR**.

This script, when executed, will create a backup of the specified PostgreSQL database in the defined backup directory. It also provides the option to compress the backup file and remove older backups to manage storage space efficiently. The next step is to schedule the execution of the script at regular intervals using a tool like cron. This allows you to automate the backup process for your PostgreSQL database:

## 1. Making the script executable:

- a. Navigate to the directory where the backup script is saved and run the following command to make the script executable:

```
chmod +x backup_script.sh
```

## 2. Automating the backup with cron:

- a. To schedule the backup script, you need to add an entry to the cron jobs list. Use the **crontab -e** command to edit the cron jobs list for the current user:

In the cron jobs list, add a new line specifying the schedule and the path to the backup script. For example:

```
0 0 * * * /path/to/backup_script.sh
```

Save the cron jobs list and exit the text editor, this example schedules the backup script to run every day at midnight (00:00).

## Exploring backup tools

When it comes to managing backups for your PostgreSQL database, there are several reliable tools available that can simplify the process and ensure the safety of your data. In this section, we will explore some popular backup tools that you can consider for your backup strategy:

**pg\_dump:** This is a built-in backup tool that comes with PostgreSQL itself. It allows you to create logical backups, generating a script containing SQL statements that can be used to recreate the database. While pg\_dump is simple and easy to use, it can be slower for large databases and may cause downtime during the backup process.

**pgBackRest:** pgBackRest is a powerful, open-source backup and restore tool specifically designed for PostgreSQL. It offers features like parallel backup and restore, incremental backups, and compression. With pgBackRest, you can efficiently manage backups, handle point-in-time recovery, and perform granular restores.

**Barman:** Barman is another popular backup and recovery manager for PostgreSQL. It provides advanced features like incremental backups, retention policies, and disaster recovery solutions. Barman supports both physical and logical backups, making it flexible for different backup scenarios.

**WAL-E:** WAL-E is a tool that focuses on continuous archiving of PostgreSQL's **write-ahead logs (WAL)**. It is designed to work with cloud storage services like Amazon S3 and Google Cloud Storage. WAL-E allows for point-in-time recovery and can be an excellent choice for environments where data durability and high availability are crucial.

**Bacula:** Bacula is a comprehensive backup solution that supports PostgreSQL and other database systems. It offers a wide range of features, including client-server architecture, backup scheduling, encryption, and tape library management. Bacula is a scalable solution suitable for large and complex backup environments.

**Pg\_probackup:** pg\_probackup is an open-source backup and recovery tool developed by the PostgreSQL community. It provides features like parallel backups, incremental backups, and validation of backups. Pg\_probackup can be an excellent choice for users who prefer a command-line interface with advanced options.

These are just a few of the backup tools available for PostgreSQL. Each tool has its unique features and capabilities, so it is essential to evaluate your requirements and choose the one that best suits your backup strategy. Whether you prefer simplicity, performance, or advanced functionalities, there is a backup tool out there to meet your needs and ensure the safety of your PostgreSQL database.

## **Recipe 82: Working with pg\_probackup, pgBackRest tool**

pg\_probackup and pgBackRest are popular backup and restore tools specifically designed for PostgreSQL databases. These tools provide efficient and reliable solutions for managing database backups, ensuring data integrity, and facilitating disaster recovery.

pg\_probackup is an open-source backup and recovery tool developed by the PostgreSQL community. It supports both full and incremental backups, making it suitable for large databases with frequent data modifications. pg\_probackup uses PostgreSQL's WAL mechanism to create consistent backups, enabling point-in-time recovery. It offers various features like parallel backup and restore, retention policies, compression, and encryption.

On the other hand, pgBackRest is a powerful and flexible backup and restore tool specifically designed for PostgreSQL. It is developed by a third-party company called **Crunchy Data** and is available as an open-source solution. pgBackRest provides advanced features like parallel backup and restore, differential and incremental backups, compression, encryption, and checksum verification. It also supports various backup storage options, including local filesystems, remote servers, and cloud storage providers.

Both pg\_probackup and pgBackRest offer command-line interfaces for easy integration into scripting and automation workflows. These tools provide comprehensive documentation and have active communities, ensuring continuous development and support.

## **Recipe 83: Installing and configuring pg\_probackup for PostgreSQL**

pg\_probackup serves as a centralized solution for efficiently managing database backups, implementing retention policies, and ensuring a reliable disaster recovery strategy. By automating key tasks, pg\_probackup simplifies the backup process, providing administrators with a comprehensive set of features to maintain data integrity and availability.

The tool supports both full and incremental backups, optimizing disk space usage and expediting restore operations. Leveraging PostgreSQL's streaming and write-ahead logging (WAL) capabilities, pg\_probackup guarantees the creation of consistent backups, crucial for maintaining data consistency.

Key advantages of pg\_probackup for PostgreSQL include centralized backup management, streamlined administration, data integrity and recovery, **point-in-time recovery (PITR)**, and monitoring and reporting capabilities. It facilitates easy scheduling of regular backups, retention policy definition, and automated backup pruning. Ensuring data integrity, pg\_probackup utilizes PostgreSQL's streaming and WAL capabilities for consistent backups and efficient restore operations. The tool supports PITR, allowing administrators to restore databases to specific points in time, minimizing data loss in case of failures. Additionally, pg\_probackup offers monitoring features to track backup progress, validate backups for data integrity, and generate reports on backup operations. Following are the steps to install and configure pg\_probackup for PostgreSQL:

1. Begin by downloading and installing the necessary packages for **pg\_probackup**.

- a. Download pg\_probackup from **postgrespro** official repositories

[https://repo.postgrespro.ru/pg\\_probackup/keys/pg\\_probackup-repo-rhel.noarch.rpm](https://repo.postgrespro.ru/pg_probackup/keys/pg_probackup-repo-rhel.noarch.rpm). Find the desired release version (for example, release/15):

```
[root@pgsrvdev log]# rpm -ivh https://repo.postgrespro.ru/pg_probackup/keys/pg_probackup-repo-rhel.noarch.rpm
Retrieving https://repo.postgrespro.ru/pg_probackup/keys/pg_probackup-repo-rhel.noarch.rpm
warning: /var/tmp/rpm-tmp.EiWQ8L: Header V4 RSA/SHA256 Signature, key ID 636d717e: NOKEY
Verifying... ################################################ [100%]
Preparing... ################################################ [100%]
Updating / installing...
 1:pg_probackup-repo-2.5.12-1 ################################################ [100%]
[root@pgsrvdev log]#
```

**Figure 10.1:** pg\_probackup – Official repository

- b. Install pg\_probackup:

```
yum install pg_probackup-15*
```

Referring to [Figure 10.2](#), the above command will download and install the **pg\_probackup** package along with its dependencies:

```
[root@pgsrvdev log]# yum install pg_probackup-15*
Updating Subscription Management repositories.
Unable to read consumer identity

This system is not registered with an entitlement server. You can use subscription-manager to register.

Last metadata expiration check: 0:00:40 ago on Sat 27 May 2023 01:15:14 PM +08.
Dependencies resolved.
=====
Package           Architecture      Version          Repository      Size
=====
Installing:
pg_probackup-15      x86_64        2.5.12-1.d6721662ec76257d9470b1d20d75b7bc6bb1501c      pg_probackup      208 k
pg_probackup-15-debuginfo  x86_64        2.5.12-1.d6721662ec76257d9470b1d20d75b7bc6bb1501c      pg_probackup      440 k
pg_probackup-15-debugsource x86_64        2.5.12-1.d6721662ec76257d9470b1d20d75b7bc6bb1501c      pg_probackup      453 k

Transaction Summary
=====
Install 3 Packages

Total download size: 1.1 M
Installed size: 3.7 M
Is this ok [y/N]: y
```

**Figure 10.2:** pg\_probackup - Installation from YUM

- Verify the installation by checking the version of **pg\_probackup**

```
pg_probackup-15 --version
```

Referring to [Figure 10.3](#), the output indicates that we have version 2.5.12 of **pg\_probackup-15** installed, which is compatible with PostgreSQL version 15.0:

```
pgsrvdev [log]# pg_probackup-15 --version
pg_probackup-15 2.5.12 (PostgreSQL 15.0)
[root@pgsrvdev log]#
```

**Figure 10.3:** pg\_probackup - Version check

- Once installed, configure **pg\_probackup** by executing the following commands:

- Create the backup directory and set the appropriate permissions:

```
sudo mkdir /pro_backup
```

```
sudo chown postgres:postgres /pro_backup
```

- Initialize the backup catalog:

```
pg_probackup init -B /pro_backup
```

Referring to [Figure 10.4](#), the output message **INFO: Backup catalog '/pro\_backup' successfully initialized** confirms that the backup catalog has been successfully initialized and created at the specified location (**/pro\_backup**):

```
[postgres@pgsrvdev pro_backup]$ pg_probackup-15 init -B /pro_backup
INFO: Backup catalog '/pro_backup' successfully initialized
[postgres@pgsrvdev pro_backup]$
[postgres@pgsrvdev pro_backup]$ tree /pro_backup/
/pro_backup/
└── backups
└── wal

2 directories, 0 files
```

**Figure 10.4:** pg\_probackup - Initialize backup catalog

c. initialize the backup instance:

```
pg_probackup-15 add-instance -B /pro_backup --instance=pgbackup
-D $PGDATA
```

Referring to [Figure 10.5](#), the output message **INFO: Instance pgbackup successfully initialized** confirms that the new instance named **pgbackup** has been successfully added to the backup catalog. The instance is now ready to be used for backup and restore operations using **pg\_probackup**.

```
[postgres@pgsrvdev pro_backup]$ pg_probackup-15 add-instance -B /pro_backup --instance=pgbackup -D $PGDATA
INFO: Instance 'pgbackup' successfully initialized
[postgres@pgsrvdev pro_backup]$ tree /pro_backup/
/pro_backup/
└── backups
    └── pgbackup
        └── pg_probackup.conf
└── wal
    └── pgbackup

4 directories, 1 file
[postgres@pgsrvdev pro_backup]$
```

**Figure 10.5:** pg\_probackup - Add Instance

### 3. Configure PostgreSQL WAL Archiving for pg\_probackup:

To configure the **archive\_command** for **pg\_probackup**, **archive-push** is the common option used. Here is an explanation of the **archive-push** archival option:

#### Using the archive-push option for archiving:

The **archive\_command** configuration in PostgreSQL specifies the command to execute for archiving WAL files:

```
archive_command =
    'pg_probackup-15 archive-push -B </Backup_path>
     --instance <Inst_Name>
     --wal-file-path=%p -wal-file-name=%f'
```

It utilizes the **pg\_probackup-15** utility with the **archive-push** subcommand. The configuration includes the following options:

- **-B </Backup\_path>**: Sets the path to the backup directory where archived files will be stored.
- **--instance <Inst\_Name>**: Specifies the name of the PostgreSQL instance being backed up.
- **--wal-file-path=%p**: Represents the path of the current WAL file to be archived.
- **--wal-file-name=%f**: Represents the filename of the current WAL file to be archived.

When triggered, PostgreSQL executes the **archive\_command** by replacing the placeholders with the appropriate values. **Pg\_probackup-15** is invoked to perform the archiving process by copying the WAL file to the designated backup directory.

Let us continue with the PostgreSQL archive configuration for **pg\_probackup**:

Open the PostgreSQL configuration file (**postgresql.conf**) using a text editor and locate the following configuration:

- a. Set the **archive\_mode** and **archive\_command** parameter configuration in **postgresql.conf** file.

```
Archive_command = 'pg_probackup-15 archive-push -B /pro_backup  
--instance pgbackup  
--wal-file-path=%p --wal-file-name=%f'  
  
archive_mode = on
```

- b. Both the parameters can also be set by running the **ALTER SYSTEM** command:

```
alter system set archive_command =  
    'pg_probackup-15 archive-push -B /pro_backup  
    --instance pgbackup  
    --wal-file-path=%p --wal-file-name=%f';  
  
alter system set archive_mode = on;
```

- c. Restart the PostgreSQL service for the changes to take effect:

```
systemctl restart postgresql-15.service
```

4. Verify the **pg\_probackup** configuration:

- a. Check the configuration settings of **pg\_probackup** by running the following command:

```
pg_probackup-15 show-config -B <backup_path> --instance=<instance_name>
```

Replace **<backup\_path>** with the path to your backup location, and **<instance\_name>** with the name of your PostgreSQL instance, as shown in the following figure:

```
[postgres@pgsrpdev pro_backup]$ pg_probackup-15 show-config -B /pro_backup/ --instance=pgbackup
# Backup instance information
pgdata = /var/lib/pgsql/15/data
system-identifier = 7231531554300609974
xlog-seg-size = 16777216
# Connection parameters
pgdatabase = postgres
# Replica parameters
replica-timeout = 5min
# Archive parameters
archive-timeout = 5min
# Logging parameters
log-level-console = INFO
log-level-file = OFF
log-format-console = PLAIN
log-format-file = PLAIN
log-filename = pg_probackup.log
log-rotation-size = 0TB
log-rotation-age = 0d
# Retention parameters
retention-redundancy = 0
retention-window = 0
wal-depth = 0
# Compression parameters
compress-algorithm = none
compress-level = 1
# Remote access parameters
remote-proto = ssh
[postgres@pgsrpdev pro_backup]$
```

**Figure 10.6:** pg\_probackup - Config detail

Referring to [Figure 10.6](#), the above command will display the current configuration settings of **pg\_probackup** for the specified backup path and instance. Verify that the settings align with your desired backup configuration, including retention policies, compression options, and other relevant parameters.

In the provided example, the output shows the configuration settings for the backup catalog located at **/pro\_backup/** and the instance named **pgbackup**. The individual configuration parameters are displayed along with their respective values.

**Note:** The output does not include values for all configuration settings. If any setting is missing, it means that the default value is being used.

4. Execute backups with **pg\_probackup**:

- a. Execute full backups with **pg\_probackup** by executing the **pg\_probackup** command:

```
pg_probackup-15 backup -B /pro_backup --instance=pgbackup full -b full
```

Referring to *Figure 10.7*, the above command performs a full backup of a PostgreSQL database using **pg\_probackup**:

```
[postgres@pgsrdev pro_backup]$ pg_probackup-15 backup -B /pro_backup --instance=pgbackup full -b full
INFO: Backup start, pg_probackup version: 2.5.12, instance: pgbackup, backup ID: RVCQF7, backup mode: FULL, wal mode: ARCHIVE
WARNING: This PostgreSQL instance was initialized without data block checksums. pg_probackup have no way to detect data block corruption.
WARNING: Current PostgreSQL role is superuser. It is not recommended to run pg_probackup under superuser.
INFO: Database backup start
INFO: wait for pg_backup_start()
INFO: Wait for WAL segment /pro_backup/wal/pgbackup/00000001000000000000004B to be archived
INFO: PGDATA size: 428MB
INFO: Current Start LSN: 0/4B00028, TLI: 1
INFO: Start transferring data files
INFO: Data files are transferred, time elapsed: 2s
INFO: wait for pg_stop_backup()
INFO: pg_stop backup() successfully executed
INFO: stop_lsn: 0/4C000088
INFO: Getting the Recovery Time from WAL
INFO: Syncing backup files to disk
INFO: Backup files are synced, time elapsed: 0
INFO: Validating backup RVCQF7
INFO: Backup RVCQF7 data files are valid
INFO: Backup RVCQF7 resident size: 428MB
INFO: Backup RVCQF7 completed
postgres@pgsrdev pro_backup$
```

**Figure 10.7:** pg\_probackup - Execute full backup

- b. List the available backups using the **pg\_probackup-15 show** command:

```
pg_probackup-15 show -B /pro_backup --instance=pgbackup
```

Referring to *Figure 10.8*, the execution of the above command will display a list of available backups with their respective backup IDs, modes, statuses, and other details. The command lists the backup we took in the preceding step of this recipe for the backup id **RVCQF7**. Here, the **Mode** column shows **DELTA**, indicating it is an incremental backup for the backup id **RVCQF7**. Please refer to the following figure:

Instance	Version	ID	Recovery Time	Mode	WAL Mode	TLI	Time	Data	WAL	Zratio	Start LSN	Stop LSN	Status
pgbackup	15	RVCQF7	2023-05-28 12:42:46+08	FULL	ARCHIVE	1/0	4s	428MB	16MB	1.00	0/4B00028	0/4C000088	OK

**Figure 10.8:** pg\_probackup - List available backup

- c. Verify the backup status in detail for a specific backup by executing the following command:

```
pg_probackup-15 show -B /pro_backup --instance=pgbackup --
```

```
backup-id=RVCQF7
```

Referring to *Figure 10.9*, this output provides a summary of the full backup performed using **pg\_probackup**. It confirms that the backup was completed successfully (**OK**) and provides details about its characteristics, such as the size of the data and **WAL** files, compression ratio, and the **LSN** range covered by the backup. Please refer to the following figure:

```
[postgres@pgsrdev pro_backup]$ pg_probackup-15 show -B /pro_backup --instance=pgbackup --backup-id=RVCQF7
#Configuration
backup-mode = FULL
stream = false
compress-alg = none
compress-level = 1
from-replica = false

#Compatibility
block-size = 8192
xlog-block-size = 8192
checksum-version = 0
program-version = 2.5.12
server-version = 15

#Result backup info
timelineid = 1
start-lsn = 0/4B000028
stop-lsn = 0/4C0000B8
start-time = '2023-05-28 12:42:43+08'
end-time = '2023-05-28 12:42:47+08'
recovery-xid = 53910
recovery-time = '2023-05-28 12:42:46+08'
data-bytes = 448843260
wal-bytes = 16777216
uncompressed-bytes = 448410828
pgdata-bytes = 448410515
status = OK
primary_conninfo = 'user=postgres channel_binding=prefer port=5432 sslmode=prefer sslcompression=0 sslnsi=1 ssl_min_r_content-crc = 4124005719
[postgres@pgsrdev pro_backup]$
```

**Figure 10.9:** pg\_probackup – Get backup detail

We have now successfully configured **pg\_probackup** for PostgreSQL. We can use the **pg\_probackup** command-line tool to perform backups, restore databases, and manage your PostgreSQL backups.

## Recipe 84: Installing and configuring pgBackRest for PostgreSQL

PgBackrest is an advanced backup and recovery tool tailored specifically for PostgreSQL databases. Similar to Barman, pgBackRest serves as a centralized solution for managing database backups, implementing retention policies, and supporting disaster recovery efforts. It simplifies the backup process by automating tasks and offering a feature-rich toolkit to guarantee data integrity and availability.

Administrators using pgBackRest can effortlessly schedule and execute regular backups for their PostgreSQL databases. The tool accommodates both full and incremental backups, optimizing disk space usage and expediting restore operations. Drawing upon PostgreSQL's streaming and **write-ahead logging (WAL)** capabilities, pgBackRest ensures the creation of coherent backups, a critical aspect for maintaining data consistency.

The distinct advantages of pgBackRest for PostgreSQL are summarized in the following key points:

- **Centralized backup management:** pgBackRest provides a centralized platform for managing backups across multiple PostgreSQL servers and databases, streamlining the backup process from a unified location.
- **Streamlined administration:** Administrators can easily schedule regular backups, define retention policies, and automate backup pruning, reducing the complexity of administrative tasks.
- **Data integrity and recovery:** Similar to Barman, pgBackRest prioritizes data integrity by leveraging PostgreSQL's streaming and WAL capabilities. This ensures the creation of consistent backups and facilitates efficient restore operations.
- **Point-in-time recovery (PITR):** pgBackRest empowers administrators to perform point-in-time recovery, enabling them to restore databases to specific moments in time. This capability minimizes data loss in the event of failures.
- **Monitoring and reporting:** pgBackRest includes monitoring features to track backup progress, validate backups for data integrity, and generate detailed reports on backup operations. This monitoring functionality enhances visibility into the backup process, facilitating proactive data management.

For a comprehensive guide on installing and configuring pgBackRest for PostgreSQL, this recipe will provide step-by-step instructions, ensuring a smooth and effective integration of PgBackrest into your database management strategy.

1. Begin by downloading and installing the necessary packages for pgBackRest:
  - a. Download pgBackRest from PGDG repositories [https://download.postgresql.org/pub/repos/yum/reporpm/EL-9-x86\\_64/pgdg-redhat-repo-latest.noarch.rpm](https://download.postgresql.org/pub/repos/yum/reporpm/EL-9-x86_64/pgdg-redhat-repo-latest.noarch.rpm). Find

the desired release version (for example, **release/2.46**).

b. Add the PostgreSQL Yum repository:

```
sudo dnf install  
https://download.postgresql.org/pub/repos/yum/repor  
pms/EL-9-x86_64/pgdg-redhat-repo-  
latest.noarch.rpm
```

c. Enable the PostgreSQL and **pgdg-common** repositories:

```
sudo dnf -qy module disable postgresql  
sudo dnf config-manager --set-enabled pgdg-common
```

d. Install pgBackRest:

```
sudo dnf install pgbackrest
```

2. Once installed, configure pgBackRest repositories by executing the following commands:

a. Create the backup directory and set the appropriate permissions:

```
# Create Backup directory  
sudo mkdir /pgbr_backup  
# Change ownership of Backup directory  
sudo chown postgres:postgres /pgbr_backup
```

b. Create the **log** directory and set the appropriate permissions:

```
# Create Log directory  
sudo mkdir /pgbr_log  
# Change ownership of Log directory  
sudo chown postgres:postgres /pgbr_log
```

3. Configure pgBackRest for PostgreSQL database, the default location for the pgBackRest configuration file (**pgbackrest.conf**) is **/etc/pgbackrest.conf**. This is the standard location where pgBackRest looks for its configuration file by default. However, it is important to note that the actual location of the configuration file may vary depending on how pgBackRest was installed or if a custom configuration path was specified during installation. Open the **pgbackrest** configuration file using a text editor and modify the following parameters in the configuration file:

```
[global]  
repo1-path=/pgbr_backup
```

```
repo1-retention-full=2
log-level-console=detail
log-level-file=detail
archive-async=y
archive-get-queue-max=2GiB
[pgbr_DB_Backup]
pg1-path=/var/lib/pgsql/15/data/
```

Let us delve into the provided configuration in more detail:

**[global] section:** It contains global configuration settings that apply to the entire pgBackRest setup. It sets up parameters that are common to all the databases being backed up. Here is a breakdown of the settings within the **[global]** section:

- **repo1-path:** Specifies the path to the backup repository. In this case, backups will be stored in the **/pgbr\_backup directory**.
- **repo1-retention-full:** Sets the retention policy for full backups. The value 2 means that only the two most recent full backups will be retained.
- **log-level-console:** Sets the console log level. The value detail indicates that detailed log information will be displayed in the console.
- **log-level-file:** Sets the log level for log files. Again, the value detail indicates detailed logging in the log files.
- **archive-async:** Enables asynchronous archiving. When set to **y**, it allows the archiving process to happen asynchronously.
- **archive-get-queue-max:** Specifies the maximum queue size for getting archives. The value 2GiB sets the maximum queue size to 2 gigabytes.
- **[pgbr\_DB\_Backup] section:** A database-specific section within the **pgbackrest.conf** file. It allows you to define settings specifically for a particular database. In the given example, the section is named **[pgbr\_DB\_Backup]**. Here is the setting within the **[pgbr\_DB\_Backup]** section:

- **pg1-path:** Specifies the path to the PostgreSQL data directory for the specific database being backed up. In the provided configuration, it is set to `/var/lib/pgsql/15/data/`.

These settings provide a basic configuration for pgBackRest, refer to the official pgBackRest documentation for more detailed configuration options based on your specific requirements.

#### 4. Configure PostgreSQL WAL archiving for pgBackRest:

To configure the **archive\_command** for pgBackRest, **archive-push** is the common options used. Here is an explanation of **archive-push** archival option:

Using **archive-push** option for archiving:

```
archive_command =
  'pgbackrest --stanza=<Database_Backup_Section>
    archive-push %p'
```

The **archive\_command** configuration in PostgreSQL specifies the command to execute for archiving **write-ahead log (WAL)** files. It utilizes the pgBackRest utility with the **archive-push** subcommand.

The configuration includes the following options:

- **pgbackrest:** This specifies the command-line utility to be executed, which is pgBackRest.
- **--stanza=<Database\_Backup\_Section>:** This option specifies the database backup section or stanza within the pgBackRest configuration. Replace **<Database\_Backup\_Section>** with the actual name of the database backup section defined in the pgBackRest configuration file (**pgbackrest.conf**). This allows PostgreSQL to invoke pgBackRest with the appropriate configuration for archiving.
- **archive-push:** This is a pgBackRest subcommand that performs the actual archival of the WAL file.
- **%p:** This is a placeholder for the path of the WAL file being archived. PostgreSQL will replace **%p** with the actual path of the WAL file when invoking the

## **archive\_command**

When triggered, PostgreSQL executes the **archive\_command** by replacing the placeholders with the appropriate values. **pgBackRest** is invoked to perform the archiving process by copying the WAL file to the designated backup directory.

Let us continue with the PostgreSQL archive configuration for pgBackRest:

Open the PostgreSQL configuration file (**postgresql.conf**) using a text editor and locate the following configuration:

- a. Set the **archive\_mode** and **archive\_command** parameter configuration **postgresql.conf** file.

```
archive_command = 'pgbackrest --stanza=pgbr_DB_Backup
                  archive-push %p'
archive_mode = on
```

- b. Both parameters can also be set by running the **ALTER SYSTEM** command:

```
alter system set archive_command =
  'pgbackrest --stanza=pgbr_DB_Backup
   archive-push %p'
alter system set archive_mode = on;
```

- c. Restart the PostgreSQL service for the changes to take effect:

```
systemctl restart postgresql-15.service
```

## 5. Create stanza for database:

- a. Open a terminal and run the following command to create a stanza for database:

```
pgbackrest --stanza=pgbr_DB_Backup --log-level-console=detail stanza-
```

```
create
```

Referring to [\*Figure 10.10\*](#), the above command initializes the necessary metadata and directory structure for the specified stanza:

```
[postgres@pgsrdev ~]$ pgbackrest --stanza=pgbr_DB_Backup --log-level-console=detail stanza-create
2023-05-28 18:30:23.800 P00 INFO: stanza-create command begin 2.46: --exec-id=9283-9b955ela --log-level-console=detail
--log-level-file=detail --log-path=/pgbr_log --pgl-path=/var/lib/pgsql/15/data/
--rep0l-path=/pgbr_backup --stanza=pgbr_DB_Backup

2023-05-28 18:30:24.408 P00 INFO: stanza-create for stanza 'pgbr_DB_Backup' on repo1
2023-05-28 18:30:24.417 P00 INFO: stanza-create command end: completed successfully (621ms)
[postgres@pgsrdev ~]$
```

**Figure 10.10:** pgBackRest - Create stanza

Once the stanza is created, we can proceed with further configuration and customization of pgBackRest as needed, such as defining additional database-specific sections and setting backup retention policies.

Note that after creating the stanza, we may need to update the **[db]** section in the **pgbackrest.conf** file with the appropriate settings for the specific database being backed up. The db-path should point to the PostgreSQL data directory for that particular database.

## 6. Check the pgBackRest configuration:

- Check the configuration settings of pgBackRest by running the following command:

```
pgbackrest check --stanza=pgbr_DB_Backup
```

Referring to the [Figure 10.11](#), the output displays various information and log messages generated during the execution of the **pgbackrest check** command. The **pgbackrest check** command is used to verify the integrity of the backup repository and perform various checks on the configuration and archived **WAL** files. It ensures that the backup setup is functioning correctly and can be relied upon for backup and recovery operations. Please refer to the following figure:

```
[postgres@pgsrdev ~]$ pgbackrest check --stanza=pgbr_DB_Backup
2023-05-28 18:42:12.714 P00 INFO: check command begin 2.46:
--exec-id=9898-9ce1d4bc --log-level-console=detail --log-level-file=detail
--log-path=/pgbr_log --pgl-path=/var/lib/pgsql/15/data/
--rep0l-path=/pgbr_backup --stanza=pgbr_DB_Backup

2023-05-28 18:42:13.321 P00 INFO: check rep0l configuration (primary)
2023-05-28 18:42:13.523 P00 INFO: check rep0l archive for WAL (primary)
2023-05-28 18:42:13.624 P00 INFO: WAL segment 000000010000000000000058 successfully archived to
'/pgbr_backup/archive/pgbr_DB_Backup/15-1/000000010000000000000058-8c3c30f0e1c7b91bd3d3b0b1df21745dcaaee00.gz' on rep0l

2023-05-28 18:42:13.624 P00 INFO: check command end: completed successfully (912ms)
[postgres@pgsrdev ~]$
```

**Figure 10.11:** pgBackRest - Verify configuration

## 7. Execute backups with pgBackRest:

- a. Execute full backups with pgBackRest by executing pgBackRest command:

```
pgbackrest backup --stanza=pgbr_DB_Backup --type=full
```

Referring to *Figure 10.12*, the above command performs a full backup of a PostgreSQL database using pgBackRest:

```
[postgres@pgsrdev ~]$ pgbackrest backup --stanza=pgbr_DB_Backup --type=full
ERROR:  '031': option '-typefull' must begin with --
[postgres@pgsrdev ~]$ pgbackrest backup --stanza=pgbr_DB_Backup --type=full
2023-05-28 18:51:31.745 P00 INFO: backup command begin 2.461 --exec-id=10233-1a38a8b4 --log-level-console=detail --log-level-file=detail --log-path=/pgbr_log --pg1-path=/var/lib/pgsrdev/pgbr/pg1
2023-05-28 18:51:32.453 P00 INFO: execute non-exclusive backup start: backup begins after the next regular checkpoint completes
2023-05-28 18:51:33.556 P00 INFO: backup start archive = 000000010000000000000000005A, lsn = 0/5A00002B
2023-05-28 18:51:33.556 P00 INFO: check archive for prior segment 0000000100000000000000000059
2023-05-28 18:52:06.806 P01 DETAIL: backup file /var/lib/postgresql/15/data/log/postgresql Sat.csv (3.8GB, 79.68%) checksum 4Bd2fef004e5be5dd0574506f5ba7e667aac50f5
2023-05-28 18:52:24.956 P01 DETAIL: backup file /var/lib/postgresql/15/data/base/5/21006 (387.2MB, 89.79%) checksum 05c9555c744ed56e8ac02f4c3cadde4793792
<<<<< Output Truncated >>>>
2023-05-28 18:52:33.076 P00 INFO: execute non-exclusive backup stop and wait for all WAL segments to archive
2023-05-28 18:52:33.277 P00 INFO: backup stop archive = 000000010000000000000000005A, lsn = 0/5A000000
2023-05-28 18:52:33.278 P00 DETAIL: write 'backup_label' file returned from backup stop function
2023-05-28 18:52:33.278 P00 INFO: check archive for segments: 00000001000000000000005A:00000001000000000000005A
2023-05-28 18:52:33.291 P00 INFO: new backup label = 20230528-185132F
2023-05-28 18:52:33.341 P00 INFO: full backup size = 3.7GB, file total = 2413
2023-05-28 18:52:33.341 P00 INFO: backup command end: completed successfully (61599ms)
2023-05-28 18:52:33.341 P00 INFO: expire command begin 2.461 --exec-id=10233-1a38a8b4 --log-level-console=detail --log-level-file=detail --log-path=/pgbr_log --repo1-path=/pgbr
2023-05-28 18:52:33.344 P00 INFO: expire command end: completed successfully (ms)
```

**Figure 10.12:** pgBackRest - Execute full backup

- b. List the available backups using the **pgbackrest info** command:

```
pgbackrest info --stanza=pgbr_DB_Backup
```

Referring to *Figure 10.13*, the output provides information about the backups for the specified stanza (**pgbr\_DB\_Backup**). The **pgbackrest info** command provides a comprehensive overview of the backup status and details for the specified stanza. It includes information about the backups' timestamps, WAL segments, database sizes, and repository information. Please refer to the following figure:

```
[postgres@pgsrdev ~]$ pgbackrest info --stanza=pgbr_DB_Backup
stanza: pgbr_DB_Backup
status: ok
cipher: none

db (current)
wal archive min/max (15): 000000010000000000000058/00000001000000000000005A

full backup: 20230528-185132F
timestamp start/stop: 2023-05-28 18:51:32 / 2023-05-28 18:52:33
wal start/stop: 00000001000000000000005A / 00000001000000000000005A
database size: 3.7GB, database backup size: 3.7GB
repo1: backup set size: 194MB, backup size: 194MB
[postgres@pgsrdev ~]$
```

**Figure 10.13:** pgBackRest - List backup information

The command lists the backup we took in the preceding step of this recipe for the stanza (**pgbr\_DB\_Backup**). The presence of a **full backup** line indicates that a full backup

has been taken. In this case, the backup ID is **20230528-185132F**.

Please note that the backup ID is a combination of the date and time when the backup was taken. The format typically follows the pattern **YYYYMMDD-HHMMSS** or includes additional characters to ensure uniqueness.

That is it! We have now successfully configured pgBackRest for PostgreSQL. We can use the **pgbackrest** command-line tool to perform backups, restore databases, and manage your PostgreSQL backups.

## Recipe 85: Installing and configuring Barman

Barman is a powerful and efficient backup and recovery manager tool specifically designed for PostgreSQL databases. It provides a centralized solution for managing database backups, implementing retention policies, and facilitating disaster recovery. Barman simplifies the backup process by automating various tasks and offering a range of features to ensure data integrity and availability.

With Barman, administrators can easily schedule and perform regular backups of their PostgreSQL databases. The tool supports both full and incremental backups, allowing for efficient disk space usage and faster restore operations. Barman utilizes PostgreSQL's streaming and **write-ahead logging (WAL)** capabilities to create consistent backups and ensure data consistency.

The key advantage of Barman for PostgreSQL can be summarized in the following steps:

- **Centralized backup management:** Barman provides a centralized solution for managing backups across multiple PostgreSQL servers and databases from a single location.
- **Streamlined administration:** Administrators can easily schedule and perform regular backups, define retention policies, and automate backup pruning.
- **Data integrity and recovery:** Barman ensures data integrity by utilizing PostgreSQL's streaming and WAL capabilities, allowing for consistent backups and efficient restore operations.
- **Point-in-time recovery (PITR):** Barman enables administrators to perform point-in-time recovery, allowing them to restore

databases to specific moments in time, minimizing data loss in case of failures.

- **Monitoring and reporting:** Barman offers monitoring capabilities to track backup progress, validate backups for data integrity, and generate reports on backup operations.

This recipe will guide through the process of installing and configuring Barman for PostgreSQL:

1. Begin by ensuring that you have PostgreSQL already installed on your system. If not, install PostgreSQL using the appropriate package for your operating system.
2. Next, install Barman by following these steps:

- a. Download Barman from PGDG repositories  
[https://download.postgresql.org/pub/repos/yum/reporpm\\_s/EL-9-x86\\_64/pgdg-redhat-repo-latest.noarch.rpm](https://download.postgresql.org/pub/repos/yum/reporpm_s/EL-9-x86_64/pgdg-redhat-repo-latest.noarch.rpm). Find the desired release version (for example, **release/3.5.0**).

- b. Add the PostgreSQL Yum repository:

```
sudo dnf install  
https://download.postgresql.org/pub/repos/yum/reporpm_s/EL-9-x86_64/pgdg-redhat-repo-latest.noarch.rpm
```

- c. Enable the PostgreSQL and **pgdg-common** repositories:

```
sudo dnf -qy module disable postgresql  
sudo dnf config-manager --set-enabled pgdg-common
```

- d. Install Barman:

```
sudo dnf install barman
```

3. After installing Barman, we need to configure it to work with your PostgreSQL database. The default location for the Barman configuration file (**barman.conf**) is **/etc/barman.conf**. This is the standard location where Barman looks for its configuration file by default. However, it is important to note that the actual location of the configuration file may vary depending on how Barman was installed or if a custom configuration path was specified during the installation.

- a. Open the Barman configuration file using a text editor and modify the following parameters in the configuration file:

```
[MY_PG_SERVER_DEV]  
description = "PostgreSQL Development Database Server"
```

```

ssh_command = ssh postgres@pgsrvdev
conninfo = host=pgsrvdev user=postgres port=5432 dbname=postgres
backup_options = concurrent_backup
backup_method = rsync
parallel_jobs = 2
network_compression = true
archiver = on

```

- b. Save the configuration file and exit the text editor.
- c. The configuration settings in the **barman.conf** file can be explained as follows:

<b>Configurations</b>	<b>Descriptions</b>
[MY_PG_SERVER_DEV]	This section header represents a specific server configuration for Barman and serves as an identifier for the server configuration settings.
description	<ul style="list-style-type: none"> <li>This optional setting provides a description of the PostgreSQL server being backed up.</li> <li>It helps identify the server in the Barman configuration.</li> </ul>
ssh_command	<ul style="list-style-type: none"> <li>Specifies the SSH command to connect to the PostgreSQL server.</li> <li>In this example, it connects to the server pgsqldev as the postgres user. Barman utilizes SSH for secure communication with the PostgreSQL server.</li> </ul>
conninfo	<ul style="list-style-type: none"> <li>Defines the connection information for the PostgreSQL server.</li> <li>In this example, it specifies the host (pgsqldev), the username (postgres), the port number (5432), and the name of the default database (postgres).</li> </ul>
backup_options	<ul style="list-style-type: none"> <li>Sets the backup options to enable concurrent backups.</li> <li>The concurrent_backup option allows barman to perform concurrent backups, enabling parallel processing and potentially faster backup times.</li> </ul>

Configurations	Descriptions
backup_method	<ul style="list-style-type: none"> <li>Specifies the backup method to be used, in this case, rsync, which utilizes the rsync utility for efficient file synchronization between the PostgreSQL server and the backup storage.</li> </ul>
parallel_jobs	<ul style="list-style-type: none"> <li>Determines the number of parallel jobs for the backup process.</li> <li>In this example, 2 parallel jobs are configured, allowing for concurrent backup tasks.</li> </ul>
network_compression	<ul style="list-style-type: none"> <li>Enables network compression during backups.</li> <li>Barman compresses the data transmitted over the network, reducing bandwidth usage and potentially improving backup performance.</li> </ul>
Archiver	<ul style="list-style-type: none"> <li>Turns on the archiving feature for managing WAL segments.</li> <li>The archiver is responsible for managing and storing the WAL (Write-Ahead Log) segments that are essential for point-in-time recovery and backup consistency.</li> </ul>

**Table 10.2:** Barman configuration options

4. Set up SSH key-based authentication, this recipe plan is to perform remote server backups using SSH, set up key-based authentication between the Barman server and the PostgreSQL server. This will enable secure and passwordless authentication using the following steps:

a. Generate an SSH key pair on the Barman server and PostgreSQL server:

**On barman server using barman user:** Execute the following command:

```
# Login to the Barman Server with barman user
```

```
sudo su - barman
```

```
# Generate SSH Keypair
```

```
ssh-keygen -t rsa
```

**On PostgreSQL server using postgres user:** Execute the following command:

```
# Login to the PostgreSQL Server with postgres user
sudo su - postgres
# Generate SSH Keypair
ssh-keygen -t rsa
```

- b. Copy the public key to the PostgreSQL and Barman server:

**On barman server using barman user:** Execute the following command:

```
# Execute on Barman Server under barman user
sudo ssh-copy-id postgres@pgsrvdev
```

**On PostgreSQL server using postgres user:** Execute the following command:

```
# Execute on PostgreSQL Server under postgres
user
sudo ssh-copy-id barman@pgsrvdev
```

- c. Test the **passwordless** SSH setup:

**On barman server:** Execute the following command:

```
# Login to the Barman Server with barman user
ssh postgres@pgsrvdev
```

**On PostgreSQL server:** Execute the following command:

```
# Login to the Barman Server with barman user
ssh postgres@pgsrvdev
```

If the configuration was successful, we should be able to establish an SSH connection without being prompted for a password.

## 5. Configure PostgreSQL WAL archiving for Barman:

To configure the **archive\_command** for Barman, there are two common options available: using **rsync** or using the **barman-wal-archive** command. Here is an explanation of each option:

**Using rsync:**

```
archive_command      =      'rsync      -av      %p
barman@barman_srv:/path/to/barman/receive-wal/%f'
```

With this option, the **archive\_command** utilizes the **rsync** utility to transfer the WAL files from the PostgreSQL server to the Barman server. The **-av** options instruct **rsync** to preserve file attributes and use verbose mode for logging the transfer. The **%p**

placeholder represents the path of the current WAL file, and **%f** represents the file name of the WAL file.

### Using barman-wal-archive:

```
archive_command = 'barman-wal-archive SERVER_NAME %p'
```

With this option, the **archive\_command** directly invokes the **barman-wal-archive** command provided by Barman. The **SERVER\_NAME** parameter should be replaced with the name of your Barman server configuration specified in the **barman.conf** file. The **%p** placeholder represents the path of the current WAL file.

The **barman-wal-archive** utility requires the **barman-cli** package to be installed. This package provides the necessary dependencies and functionality for **barman-wal-archive** to work properly.

To install **barman-cli** and its dependencies, we can use the package manager. Here are examples for installation on RedHat:

```
yum install barman-cli
```

Once **barman-cli** is installed, you should be able to use the **barman-wal-archive** command and configure the **archive\_command** in PostgreSQL accordingly.

Let us continue with the archive configuration for PostgreSQL:

Open the PostgreSQL configuration file (**postgresql.conf**) using a text editor and locate the following configuration:

- Set the **archive\_mode** and **archive\_command** parameter configuration **postgresql.conf** file.

```
archive_command = 'barman-wal-archive pgsvrdev  
MY_PG_SERVER_DEV %p'  
archive_mode = on
```

- Both the parameter can also be set by running the **ALTER SYSTEM** command:

```
alter system set archive_command = 'barman-wal-archive pgsvrdev  
MY_PG_SERVER_DEV %p';  
alter system set archive_mode = on;
```

- Restart the PostgreSQL service for the changes to take effect:

```
systemctl restart postgresql-15.service
```

## 6. Test Barman server configuration:

- To verify that Barman is working correctly, perform a configuration test by executing the following command:

```
barman check MY_PG_SERVER_DEV
```

When we execute the above command, Barman performs several checks on the specified PostgreSQL server. Referring to [Figure 10.14](#), these checks include verifying the connectivity to the server, checking the availability of required directories, and validating the configuration settings:

```
[barman@pgsrvdev .ssh]$ barman check MY_PG_SERVER_DEV
Server MY_PG_SERVER_DEV:
  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@pgsrvdev .ssh]$
```

**Figure 10.14:** Barman - List server detail

The **barman check** command helps to ensure that the PostgreSQL server is properly configured and ready for backup and recovery operations with Barman.

After executing the command, Barman will provide a summary of the health check results, indicating if the server configuration passes or fails the checks.

## 7. Perform a sample backup:

- Connect to the PostgreSQL server and switch to the **barman** user and perform a sample backup using the following command:

```
# Switch to barman user
```

```
sudo su - barman
```

```
# Perform sample backup
```

```
barman backup MY_PG_SERVER_DEV
```

Replace **MY\_PG\_SERVER\_DEV** with the name of PostgreSQL server configured in **/etc/barman.conf** file.

- b. Barman will start the backup process and create a backup of the PostgreSQL server. It will store the backup files in the configured **backup\_directory**. Please refer to the following figure:

```
[barman@pgsrdev .ssh]$ barman backup MY_PG_SERVER_DEV
Starting backup using rsync-concurrent method for server MY_PG_SERVER_DEV in /var/lib/barman/MY_PG_SERVER_DEV/base/20230527T105132
Backup start at LSN: 0/9000028 (000000100000000000000009, 00000028)
This is the first backup for server MY_PG_SERVER_DEV
WAL segments preceding the current backup have been found:
    000000010000000000000006 from server MY_PG_SERVER_DEV has been removed
    000000010000000000000007 from server MY_PG_SERVER_DEV has been removed
Starting backup copy via rsync/SSH for 20230527T105132 (2 jobs)
Copy done (time: 1 second)
This is the first backup for server MY_PG_SERVER_DEV
Asking PostgreSQL server to finalize the backup.
Backup size: 43.6 MiB
Backup end at LSN: 0/9000100 (000000100000000000000009, 00000100)
Backup completed (start time: 2023-05-27 10:51:33.500948, elapsed time: 2 seconds)
Processing xlog segments from file archival for MY_PG_SERVER_DEV
    000000010000000000000008
    000000010000000000000009
    000000010000000000000009.00000028.backup
[barman@pgsrdev .ssh]$
```

**Figure 10.15:** Barman – Backup database

Referring to [Figure 10.15](#), The barman backup **MY\_PG\_SERVER\_DEV** command execution output can be summarized as follows:

- Backup initiated for server **MY\_PG\_SERVER\_DEV** using the **rsync-concurrent** method and the directory where the backup files will be stored (**/var/lib/barman/MY\_PG\_SERVER\_DEV/base/20230527T105132**).
- Backup started at **log sequence number (LSN) 0/9000028**.
- This is the first backup for server **MY\_PG\_SERVER\_DEV**: This line indicates that this backup is the first backup for the specified server (**MY\_PG\_SERVER\_DEV**). If it were not the first backup, it would show information about the previous backups.
- Preceding WAL segments **000000010000000000000006** and **000000010000000000000007** have been found and

removed.

- This line signifies that some WAL segments have been found and removed. It lists the LSNs of the removed segments.
- Backup files are being copied via **rsync/SSH** with 2 concurrent jobs and the copying process is completed in **1 second** with the message **This is the first backup for the server MY\_PG\_SERVER\_DEV**.
- PostgreSQL server is being asked to finalize the backup with its backup size **43.6 MiB**.
- Backup completed at LSN **0/9000100**. Backup process took **2 seconds**. Xlog segments **000000010000000000000008**, **000000010000000000000009**, and **0000000100000000000009.00000028.backup** are being processed from file archival.

Overall, this output provides a detailed summary of the backup process, including start and end LSNs, backup size, completion status, and the processing of transaction log segments.

This backup can now be used for disaster recovery or restoring your PostgreSQL database when needed.

## Recipe 86: Incremental/differential backup

Incremental and differential backups are essential strategies for backing up PostgreSQL databases, providing efficient data protection while optimizing storage space.

In an incremental backup scenario, let us say you performed a full backup of your PostgreSQL database on day 1. On day 2, only a few changes were made to the database, such as adding new records or modifying existing ones. Instead of creating a new full backup, an incremental backup captures only the changes since the last backup. This results in a smaller backup file and reduces the time required for backup and restoration.

On the other hand, in a differential backup scenario, let us consider the same full backup performed on day 1. On day 2, multiple changes were made to the database, including new records, modifications, and

deletions. A differential backup captures all changes made since the last full backup. Therefore, the backup file contains the accumulated changes since Day 1, providing a balance between storage efficiency and ease of restoration.

Using incremental and differential backups allows you to maintain a comprehensive backup history while minimizing the storage requirements. This recipe will guide you through the process of performing incremental and differential backups for a PostgreSQL database:

1. Create a directory on your storage medium to store the backup files. For example, create a folder named **pg\_backup** on an external hard drive.
2. Enable WAL archiving:
  - a. Open the PostgreSQL configuration file (**postgresql.conf**) and locate the **wal\_level** parameter and set it to **archive** to enable WAL archiving.
  - b. Specify a directory where the WAL archives will be stored by setting the **archive\_command** parameter. For example, **archive\_command = 'cp %p /path/to/archive/%f'**.
3. Perform the initial full backup:
  - a. Open a terminal or command prompt and navigate to the PostgreSQL installation directory and execute the following command to perform a full backup and store it in the backup directory.

```
pg_dump -Fc -f /pg_backup/full_backup_202305.dump -d postgres
```

The above command performs a full backup of the **postgres** database. The backup is stored in a custom format (**-Fc**) and saved in the file named **full\_backup\_202305.dump** located in the **/pg\_backup** directory.

This command captures the entire database, including all schemas, tables, and data. It ensures a comprehensive backup of the **postgres** database, allowing for full restoration if needed.

4. Perform incremental backups:

**Note: pg\_dump is a backup and restore tool used for logical backups, which means it creates a text-based representation of the database objects and data. While it**

**is a reliable tool for creating full backups and restoring them, it does not have built-in functionality for incremental and delta backups.**

Incremental and delta backups involve capturing only the changes (deltas) that have occurred since the last backup, reducing the backup size and improving backup efficiency. This type of backup is typically achieved using tools specifically designed for incremental backups, such as **pg\_probackup**, **pg\_basebackup**, **barman**.

### **Incremental backup with barman tool:**

In the following steps, we will guide you through the process of performing incremental backups using the Barman tool in PostgreSQL.

- a. The initial base backup is required before running incremental backups with Barman. Refer to the previous recipe in this chapter *Installing and configuring Barman* to make the initial basic backup (full backup).
- b. After initial full backup, we can perform incremental backups to capture changes since the last backup. Prior to that, we need to configure the Barman for incremental backup in PostgreSQL.

Execute the following command to configure barman for incremental backup:

- i. **Locate the barman.conf file:** Typically, the **barman.conf** file is located in the **/etc/barman/** directory. Use a text editor to open the file.
- ii. **Add or modify the reuse\_backup option:** Search for the **[<barman-server-name>]** section in the **barman.conf** file. Inside the section, add or modify the **reuse\_backup** option and set its value to link as shown following configuration section:

```
[MY_PG_SERVER_DEV]
description = "PostgreSQL Development Database Server"
ssh_command = ssh postgres@pgsrvdev
conninfo = host=pgsrvdev user=postgres port=5432 dbname=postgres
backup_options = concurrent_backup
```

```
backup_method = rsync
parallel_jobs = 2
network_compression = true
reuse-backup = link
archiver = on
```

By adding or modifying the **reuse\_backup** option with the value **link** in the specific **<barman-server-name>** section, you instruct Barman to use hard links for incremental backups.

After making the configuration change, Barman will utilize hard links for the incremental backup process, reducing disk space usage and backup time by reusing existing unchanged files from previous backups.

- c. Execute incremental backup PostgreSQL database instance from **barman**:

```
# Switch to barman user
sudo su - barman
# Perform sample backup
barman backup MY_PG_SERVER_DEV
```

In order to make incremental backup with Barman, we do not need to specify additional options during the **backup** command. Barman automatically performs incremental backups by utilizing the WAL files.

### **Incremental backup with pg\_probackup:**

Incremental backups with **pg\_probackup** tool allow you to efficiently back up only the changes made since the last full or incremental backup, reducing backup time and storage requirements. In the following steps, we will guide you through the process of performing incremental backups using the **pg\_probackup** tool in PostgreSQL.

- a. Perform full backup with **pg\_probackup** (if not already done). Refer to the previous recipe in this chapter *Install and configure pg\_probackup for PostgreSQL* to make the initial base backup (full backup).
- b. Execute the following command to perform an incremental backup using **pg\_probackup**.

```
pg_probackup-15 backup -B /pro_backup --instance=pgbackup -b delta
```

Referring to [\*\*Figure 10.16\*\*](#), the execution involved starting an incremental backup with **pg\_probackup-15** using the

**DELTA** backup mode. The backup process included waiting for certain checkpoints, transferring data files, validating the backup, and obtaining necessary information for recovery. Warnings regarding **data block checksums** and the usage of **pg\_probackup** as a superuser were also mentioned. Please refer to the following figure:

```
[postgres@pgsrdev pro_backup]$ pg_probackup-15 backup -B /pro_backup --instance=pgbackup -b delta
INFO: Backup start, pg_probackup version: 2.5.12, instance: pgbackup, backup ID: RVCYEV, backup mode: DELTA
WARNING: This PostgreSQL instance was initialized without data block checksums. pg_probackup have no way to verify them.
WARNING: Current PostgreSQL role is superuser. It is not recommended to run pg_probackup under superuser.
INFO: Database backup start
INFO: wait for pg_backup_start()
INFO: Parent backup: RVCYEV
INFO: Wait for WAL segment /pro_backup/wal/pgbackup/000000010000000000000051 to be archived
INFO: PGDATA size: 431MB
INFO: Current Start LSN: 0/51000028, TLI: 1
INFO: Parent Start LSN: 0/4E000028, TLI: 1
INFO: Start transferring data files
INFO: Data files are transferred, time elapsed: 1s
INFO: wait for pg_stop_backup()
INFO: pg_stop backup() successfully executed
INFO: stop_lsn: 0/520000B8
INFO: Getting the Recovery Time from WAL
INFO: Syncing backup files to disk
INFO: Backup files are synced, time elapsed: 0
INFO: Validating backup RVCYEV
INFO: Backup RVCYEV data files are valid
INFO: Backup RVCYEV resident size: 8857kB
INFO: Backup RVCYEV completed
[postgres@pgsrdev pro_backup]$
```

**Figure 10.16:** pg\_probackup - Execute Delta backup

- c. List the available backups using the **pg\_probackup-15 show** command:

```
pg_probackup-15 show -B /pro_backup --instance=pgbackup
```

Referring to [Figure 10.7](#), the execution of above command will display a list of available backups with their respective backup IDs, modes, statuses, and other details. The command lists the backup we took in the preceding step of this recipe for the backup id **RVCYEV**. Here, the **Mode** column shows **DELTA**, indicating it is an incremental backup for the backup id **RVCYEV**. Please refer to the following figure:

Instance	Version	ID	Recovery Time	Mode	WAL Mode	TLI	Time	Data	WAL	Zratio	Start LSN	Stop LSN	Status
pgbackup	15	RVCYEV	2023-05-28 15:35:21+08	DELTA	ARCHIVE	1/1	3s	8857kB	16MB	1.00	0/51000028	0/520000B8	OK
pgbackup	15	RVCYEV	2023-05-28 15:34:59+08	DELTA	ARCHIVE	1/1	3s	1204kB	16MB	1.00	0/4E000028	0/4F0000B8	OK
pgbackup	15	RVCQF7	2023-05-28 12:42:46+08	FULL	ARCHIVE	1/0	4s	428MB	16MB	1.00	0/4B000028	0/4C0000B8	OK

**Figure 10.17:** pg\_probackup - Verify delta backup

### Incremental backup with pgBackRest:

Incremental backups are an efficient way to reduce storage space and backup duration by only capturing changes made since the last full backup. pgBackRest is a powerful open-source tool specifically designed for PostgreSQL database backup and restore operations. In the following steps, we will guide you through the process of performing incremental backups using the pgBackRest tool in PostgreSQL.

- a. Perform Full Backup with pgBackRest (if not already done). Refer to the previous recipe in this chapter *Install and configure pgBackRest for PostgreSQL* to make the initial base backup (full backup).
  - b. Execute the following command to perform an incremental backup using **pg\_probackup**.

```
pgbackrest backup --stanza=pgbr_DB_Backup --type=diff
```

Referring to [Figure 10.18](#), the above command performs an incremental backup of a PostgreSQL database using pgBackRest using the **diff in** command as the backup type:

**Figure 10.18:** pgBackRest - Execute Incremental backup

## Recipe 87: Working with schema level backup

Schema-level backups provide the flexibility to selectively backup and restore specific database schemas, allowing for efficient data management and recovery.

In a scenario where you manage a comprehensive database for an organization, imagine that you have multiple schemas representing different departments such as HR, Finance, and Sales. Each schema contains specific data relevant to its respective department. However, you may only need to focus on backing up and restoring a particular schema, such as HR, which contains critical employee information.

By working with schema-level backups, you can execute a targeted backup strategy. For example, you can perform a schema-level backup

using the **pg\_dump** utility and specifying the desired schema to be included. This results in a backup file that solely contains the data from the specified schema, reducing the backup size and streamlining the restoration process.

This recipe will guide you through the process of performing schema-level backups in PostgreSQL. Let us consider a scenario where you are managing an e-commerce platform that uses a PostgreSQL database. The database consists of multiple schemas, including **customers**, **orders**, **products**, and **payments**. You want to perform a backup of only the **customers** and **orders** schemas, as they contain critical data:

1. Set up the backup directory:
  - a. Create a directory on your storage medium to store the backup files. For example, create a folder named **pg\_backup** on an external hard drive.
2. Identify the schemas to backup:
  - a. We determine that the **customers** and **orders** schemas are the ones we need to back up to ensure the recovery of essential customer and order information:

```
SELECT schema_name  
FROM information_schema.schemata;
```

Following the execution output of the above statement, referring to [\*Figure 10.19\*](#), that displays the list of schema names retrieved from the information schema in PostgreSQL:

	schema_name	name	lock
1	public		
2	cust_info		
3	transactions		
4	payments		
5	products		
6	orders		
7	customers		
8	information_schema		
9	pg_catalog		
10	pg_toast		

**Figure 10.19:** Schema level backup - List schemas

### 3. Perform the schema level backup:

- Open a terminal or command prompt and navigate to the PostgreSQL installation directory.

Execute the following command to perform a schema-level backup and store it in the backup directory:

```
pg_dump -Fc -f /pg_backup/schema_bkp_202305.dump -n customers -n
orders -d postgres
```

The above command performs a schema-level backup for the **customers** and **orders** schemas in the **postgres** database. The backup is saved in a custom format (**-Fc**) and stored in the file named **schema\_bkp\_202305.dump** located in the **/pg\_backup** directory.

By using the **-n flag** followed by the schema names, the command ensures that only the specified schemas are included in the backup. This selective approach allows for targeted restoration of specific schemas when needed. The backup is performed on the **postgres** database, indicated by the **-d postgres** option.

If need to include additional schemas in the backup, we can simply add more **-n flag** followed by the schema names to the command.

#### 4. Storage and retention:

- a. Store the backup file in a secure location, preferably offsite or on a separate server.
- b. Implement a retention policy that suits your requirements. For example, you may choose to keep multiple copies of schema-level backups for a certain period.

By working with schema-level backups, you have effectively backed up and safeguarded the specific schemas containing crucial customer and order data. This targeted approach allows for more efficient backup and restoration processes while reducing storage requirements.

## Recipe 88: Monitoring backup

A backup is a copy of the data that can be used to restore the database in case of data loss. Planning a backup strategy involves identifying the critical data, determining the backup frequency and retention policy, choosing the appropriate backup method, and testing the backups regularly. Identifying the critical data involves determining the data that is essential for the business operations and prioritizing it based on its importance.

Determining the backup frequency and retention policy involves deciding how often backups are created and how long they are retained. Choosing the appropriate backup method involves selecting the backup method that suits the database size, availability, and recovery time requirements. Testing the backups regularly involves checking them for their reliability and usability in case of data loss. PostgreSQL offers several tools and techniques to create and manage backups, such as **pg\_dump**, **pg\_basebackup**, **pg\_restore**, and **pg\_ctl**, which users can use to implement their backup strategy.

To plan a backup strategy for PostgreSQL, the following are the general steps:

1. **Identify critical data:** Determine the data that is essential for the business operations and prioritize it based on its importance.
2. **Determine backup frequency and retention policy:** Decide how often backups are created and how long they are retained.

The frequency and retention policy may vary depending on the business needs and data recovery objectives.

3. **Choose the appropriate backup method:** Select the backup method that suits the database size, availability, and recovery time requirements. PostgreSQL offers several backup methods, including logical, physical, and continuous archiving backup.
4. **Test backups regularly:** Regularly test the backups to ensure their reliability and usability in case of data loss.

## Improving backup performance

In the realm of database management systems, backups play a crucial role in safeguarding data integrity and facilitating disaster recovery. PostgreSQL, one of the most popular open-source relational database systems, continuously evolves to meet the growing demands of modern data-intensive applications. In PostgreSQL, several enhancements have been introduced to improve backup performance, enabling database administrators to efficiently create and restore backups. This recipe topic explores the various techniques and strategies that can be employed to optimize backup processes in PostgreSQL, providing readers with the knowledge and insights to enhance their backup performance.

Strategies to improve backup performance in PostgreSQL:

Strategies	Description
Utilize parallelism	<ul style="list-style-type: none"><li>• Enable parallel backup and restore operations in PostgreSQL.</li><li>• Run multiple backup or restore processes simultaneously.</li></ul>
Adjust checkpoint settings for efficient data flushing	<ul style="list-style-type: none"><li>• Configure checkpoint-related parameters, such as <b>checkpoint_timeout</b> and <b>max_wal_size</b>.</li><li>• Optimize the timing and frequency of checkpoints during backup operations.</li><li>• Ensures efficient flushing of modified data from memory to disk.</li></ul>
Utilize compression	<ul style="list-style-type: none"><li>• Enable backup compression during backup operations.</li><li>• Reduces the backup size and transfer time.</li><li>• Improves backup performance by minimizing the amount of data to be stored and transferred.</li></ul>

Strategies	Description
Separate data and backup storage	<ul style="list-style-type: none"> <li>Store backups on separate disks or storage systems from the production database.</li> <li>Prevents backup operations from interfering with regular database operations.</li> </ul>
Implement incremental backups	<ul style="list-style-type: none"> <li>Perform periodic incremental backups instead of full backups.</li> <li>Capture only the changes made since the last full backup.</li> <li>Significantly reduces backup duration, especially for large databases with frequent data modifications.</li> <li>Requires a combination of full backups and incremental backups for complete data restoration.</li> </ul>

**Table 10.3:** Backup performance strategies

## Conclusion

In conclusion, this chapter has provided a comprehensive exploration of database backup in PostgreSQL. We started by understanding the importance of backup and its role in ensuring data availability and integrity. We then explored various backup methods, such as SQL dump and file system level backup, along with techniques to improve backup performance.

Additionally, we discussed a range of backup tools available in PostgreSQL and their use cases. By following the practical recipes provided, you gained hands-on experience in backing up PostgreSQL databases using logical and physical backup methods.

Furthermore, we introduced automation techniques to streamline the backup process and presented advanced tools like pg\_probackup, pgBackRest, and Barman for more robust backup management. Finally, we covered incremental and differential backup techniques and highlighted the advantages of schema level backups. With the knowledge and skills gained from this chapter, you are now equipped to design and implement a reliable backup strategy for your PostgreSQL databases, ensuring data protection and facilitating efficient recovery when necessary.

As we transition to the next chapter, readers will immerse themselves in the critical realm of database recovery in PostgreSQL. This chapter is a comprehensive guide that navigates through various facets of recovery, ranging from crash recovery and dropped table/database

recovery to damaged table/database recovery and point-in-time recovery.

### **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)



# CHAPTER 11

# Recovery

## Introduction

The chapter recovery focuses on the aspect of recovering a database. This comprehensive guide navigates through various aspects of recovery, encompassing crash recovery, dropped table/database recovery, damaged table/database recovery, and **point-in-time recovery (PITR)**. Through practical examples and step-by-step instructions, readers will acquire an understanding of the processes involved in restoring a PostgreSQL database.

The journey begins with an *Introduction to database recovery*, setting the stage by unraveling the foundational principles governing the art of restoration. Next, there is a key point in the journey with a section on *Planning Recovery*, where readers delve into the planning required for orchestrating a seamless database recovery process. The chapter further advances into the details of *Understanding Crash Recovery*, equipping reader with a clear understanding of the steps essential for database restoration post-system failure.

## Structure

In this chapter, we will cover the following topics:

- Introduction to database recovery
- Planning recovery
- Understanding crash-recovery
- Recovering with PITR

- Explore dropped table vs damaged table. database recovery

## Objectives

This chapter aims to provide a clear understanding of restore techniques in PostgreSQL. The objectives encompass introducing the concept of database recovery and its significance in maintaining data integrity. Additionally, the chapter delves into crash recovery, and the detailed steps required to recover a database post-system failure or crash. Various scenarios of dropped table or database recovery will be explored, offering efficient strategies for data retrieval. The chapter also provides insights into the recovery of a database or specific tables affected by factors like hardware failures. Furthermore, it explicates the concept of Point-in-time recovery and underscores its significance in restoring a database to a specific point. The ultimate goal is to guide readers through the configuration and implementation of PITR in a PostgreSQL environment.

### Introduction to database recovery

In PostgreSQL, understanding the concept of database restore is crucial for effectively managing and recovering from data loss scenarios. Database restore refers to the process of recovering a database to a previous state or a specific point in time using a backup. This feature plays a vital role in maintaining data integrity and ensuring the availability of data in PostgreSQL. With PostgreSQL, users have access to various techniques and tools to perform database restores. These include logical backups, physical backups, and continuous archiving backups. Logical backups generate a text file containing SQL commands that can be used to recreate the database objects and data. Physical backups create a binary copy of the database files at the file system level, simplifying the restoration process. Continuous archiving backups generate a continuous stream of archived **write-ahead log (WAL)** files, allowing for restoration to a specific point in time.

PostgreSQL provides several tools to facilitate database backup and restore, such as **pg\_dump**, **pg\_basebackup**, and **pg\_restore**. These tools enable users to create and restore backups efficiently, ensuring data safety and minimizing downtime. By understanding the concepts and utilizing the tools available, administrators and

users of PostgreSQL can confidently perform database restores, protecting their data and ensuring the continuity of their operations.

## Planning recovery

Recovery planning is an important aspect of managing a PostgreSQL database. It involves anticipating and preparing for potential disasters or unexpected events that could result in data loss or database unavailability. Having a well-defined recovery plan ensures that you can restore your database to a stable state efficiently and minimize downtime.

To begin planning recovery, start by identifying the various types of failures that can occur, such as hardware failures, software crashes, or human errors. Assess the potential impact of these failures on your database and prioritize them based on their severity and likelihood.

Next, determine the appropriate backup strategy for your environment. Decide whether you will use logical backups, for example, using the **pg\_dump** utility or physical backups, for example, using the **pg\_basebackup** tool based on factors like database size, **Recovery Time Objectives (RTO)**, and available resources. Implement a backup schedule that aligns with your business requirements and ensure backups are stored securely in an offsite location.

Consider incorporating incremental backups or continuous archiving methods, such as WAL archiving, to supplement your backup strategy. These techniques allow for point-in-time recovery and can significantly reduce data loss in case of a failure.

By engaging in comprehensive recovery planning, you can proactively safeguard your PostgreSQL database against potential failures. A well thought-out recovery plan coupled with regular testing and continuous improvement will confidence in ability to restore data and ensure business continuity in the face of adversity.

## Recipe 89: Recovery from logical backup

Recovering data from a logical backup is a way in PostgreSQL to restore a database to a specific point in time or recover from data loss. In this recipe, we will explore the steps to recover from a logical backup in PostgreSQL using the **pg\_restore** utility.

Imagine an IT administrator responsible for managing the PostgreSQL database of a large e-commerce website. Due to an accidental operation, the customer table data was impacted and needs to be recovered from a logical backup, the forthcoming steps are crafted to guide through the process of precise recovery from a logical backup. Let us delve into the recipe steps of this recovery process.

- 1. Prepare the database backup files:** Referencing the preceding chapter, *Chapter 10, Backup*, and the recipe titled *Working with Logical backup*, we outlined the steps for creating logical backups tailored to the database and to obtain a list of logical backups.
- 2. Stop PostgreSQL service:** Before initiating the restore process, stop the PostgreSQL service to prevent conflicts or data corruption during the restore operation. Execute the following command to stop the PostgreSQL service:

```
systemctl stop postgresql-15.service
```

- 3. Restoring from logical backup:** Open a terminal or command prompt on your system and run the following command to restore the database from the logical backup file:

```
pg_restore -U postgres -Fc -d pg5 Logical_Bkp_202305.sql
```

The **pg\_restore** utility will execute the SQL commands from the backup file and restore the database to its original state.

- 4. Start the PostgreSQL service:** Start the PostgreSQL service using the following command and this will bring the database back online after the restore process:

```
systemctl start postgresql-15.service
```

- 5. Verify the restore:** Connect to the PostgreSQL database and check if the restore was successful. Query the **customers** table to verify if the impacted table are recovered.

Additionally, perform any other necessary verification steps specific to the e-commerce website, such as testing functionality that relies on the customer data or checking for data consistency in related tables.

In the scenario, the accidental deletion caused significant data loss in the database. The specific items that were deleted may include customer records, order details, or any other critical data associated

with the e-commerce website. The verification steps mentioned above should be tailored to the specific data and functionality impacted by the deletion.

## Recipe 90: Recovery from physical backup

In the realm of PostgreSQL database management, the process of recovering from a physical backup holds significant importance for ensuring data integrity and system resilience. This recipe unfolds a step-by-step guide to navigate the intricacies of recovering from a physical backup, providing users with a robust strategy to restore their database to a specific point in time or recover from critical data loss scenarios.

Let us explore the essential steps and techniques to master the art of recovery from a physical backup in PostgreSQL.

- 1. Prepare the backup files:** Referencing the preceding chapter, [Chapter 10, Backup](#), and the recipe titled *Working with Physical backup*, we outlined the steps for creating physical backups tailored to the database and to obtain a list of physical backups.
- 2. Stop PostgreSQL service:** Before initiating the restore process, stop the PostgreSQL service to prevent conflicts or data corruption during the restore operation. Execute the following command to stop the PostgreSQL service:

```
systemctl stop postgresql-15.service
```

- 3. Delete the existing data directory:** Delete the existing data directory using the following command. The data directory location may vary depending on your PostgreSQL installation:

```
sudo rm -rf /var/lib/pgsql/15/
```

**Note: Deleting the data directory ensures that the restored database will not have any remnants of previous data or configuration that may interfere with the recovery process. It also helps to avoid any conflicts or inconsistencies that may arise if the existing data directory is not fully removed before restoring the backup.**

- 4. Restoring from physical backup:** We can use a backup tool like **rsync**, **tar**, or **pg\_basebackup** (a specific tool for

PostgreSQL) to restore the entire PostgreSQL database that you backed up in *step 1* of this recipe. These tools help you bring back all the data and settings of the database to its original state.

Run the command specific to chosen tool for restore. For example, using **tar**:

```
tar -xvC /var/lib/pgsql/15/data/ -f /pg_backup/Physical_bkp/base.tar.gz
```

The above command restores a **pg\_basebackup** using **tar**. The **tar** command executes with the provided options and paths, initiating the restoration process. It transfers the files from the source directory (**/pg\_backup/Physical\_bkp/**) to the destination directory (**/var/lib/pgsql/15/data**).

5. **Start the PostgreSQL service:** Start the PostgreSQL service using the following command and this will bring the database back online after the restore process:

```
systemctl start postgresql-15.service
```

6. **Verify the restore:** Connect to the PostgreSQL database and check if the restore was successful. Query the **customers** table to verify if the impacted table are recovered.

Additionally, perform any other necessary verification steps specific to the e-commerce website, such as testing functionality that relies on the customer data or checking for data consistency in related tables.

## Understanding crash recovery

Crash recovery is a critical aspect of database management systems, including PostgreSQL. It refers to the process of restoring a database to a consistent state after an unexpected system failure or crash. During normal operations, PostgreSQL maintains a WAL that records all changes made to the database. This log acts as a safety net, allowing the database to recover to a consistent state in the event of a crash.

When a crash occurs, PostgreSQL's crash recovery mechanism is triggered upon the next start-up of the database. The recovery process involves several steps, including analysis of the WAL, identifying the last checkpoint, and applying the necessary changes to restore the database's consistency.

During crash recovery, PostgreSQL performs the following steps:

1. **Redo**: The first step is the redo phase, where PostgreSQL applies changes from the WAL to the database. This ensures that any committed transactions that were not yet written to disk are recovered.
2. **Undo**: After redoing the changes, the undo phase takes place. In this phase, PostgreSQL identifies any incomplete transactions or uncommitted changes in the WAL and rolls them back to maintain database integrity.
3. **Checkpoint**: Once the redo and undo phases are completed, PostgreSQL performs a checkpoint. A checkpoint is a snapshot of the database state at a specific point in time. It updates the control files to indicate the completion of recovery up to a certain point. Checkpoints are crucial for optimizing crash recovery time.
4. **Recovery completion**: After the checkpoint, PostgreSQL completes the recovery process and brings the database online, allowing users and applications to access it.

Understanding crash recovery is essential for database administrators to ensure the reliability and data integrity of PostgreSQL databases. It is crucial to regularly monitor the system for any signs of crashes or failures and have a backup and recovery strategy in place to minimize potential data loss or downtime.

## Recovering with PITR

Recovering with **point-in-time recovery (PITR)** in PostgreSQL provides a mechanism to restore a database to a specific point in time, allowing for granular recovery and data restoration. PITR involves a combination of a full backup and archived WAL files to achieve the desired recovery point. To initiate PITR, one must first obtain a recent full backup of the database. This full backup serves as the starting point for the recovery process.

Additionally, the archived WAL files, which contain the transaction logs capturing all changes made to the database, are necessary for PITR. By specifying the target time, PostgreSQL automatically replays the necessary WAL files and restores the database to the desired point in time.

PITR provides a reliable solution for scenarios such as accidental data deletion, system failures, or database corruption. It allows organizations to recover their data precisely to a specific moment, minimizing data loss and ensuring data integrity. Whether utilizing physical backups for complete cluster restoration or leveraging logical backups for selective table recovery, PITR in PostgreSQL is a robust feature that empowers database administrators to swiftly and accurately restore their databases to a previous state.

## Recipe 91: Point-in-time recovery

Performing point-in-time recovery in PostgreSQL is essential for restoring a database to a specific point in time or recovering from various types of failures. This recipe will guide through the steps to perform point-in-time recovery in PostgreSQL.

**Scenario:** A large e-commerce website has experienced data corruption in their customer **testings** table. To recover the data, point-in-time recovery will be performed using the Barman, pg\_probackup and PgBackRest tools. For an in-depth understanding of these tools, refer to the topic titled *Exploring Backup Tools* in [Chapter 10, Backup](#).

### PITR Recovery with PgBackRest:

- 1. Initiate PostgreSQL database backup: Execute the command below to Perform a full backup of the PostgreSQL database using Pgbackrest:**

```
pgbackrest backup --stanza=pgbr_DB_Backup --type=full
```

Verify the available backups using the **pgbackrest info** command as follows:

```
pgbackrest info --stanza=pgbr_DB_Backup
```

Refer to [Figure 11.1](#), The execution shows that there is a full back-up with the ID **20230528-185132F**, a differential backup with the ID **20230528-185132F\_20230528-192018D** and a full backup with **ID 20230618-094303F** available for restore and PITR recovery. The WAL archive range is from **00000001000000000000005A** to **000000030000000000000061**. Please refer to the following:

```
[postgres@pgsrvdev 15]$ pgbackrest info --stanza=pgbr_DB_Backup
stanza: pgbr_DB_Backup
status: ok
cipher: none

db (current)
wal archive min/max (15): 0000000100000000000005A/000000040000000000000061

full backup: 20230528-185132F
timestamp start/stop: 2023-05-28 18:51:32 / 2023-05-28 18:52:33
wal start/stop: 0000000100000000000005A / 0000000100000000000005A
database size: 3.7GB, database backup size: 3.7GB
repo1: backup set size: 194MB, backup size: 194MB

diff backup: 20230528-185132F_20230528-192018D
timestamp start/stop: 2023-05-28 19:20:18 / 2023-05-28 19:21:40
wal start/stop: 0000000100000000000005C / 0000000100000000000005C
database size: 3.8GB, database backup size: 533.3MB
repo1: backup set size: 195.8MB, backup size: 70.4MB
backup reference list: 20230528-185132F

full backup: 20230618-094303F
timestamp start/stop: 2023-06-18 09:43:03 / 2023-06-18 09:44:03
wal start/stop: 00000004000000000000061 / 00000004000000000000061
database size: 3.8GB, database backup size: 3.8GB
repo1: backup set size: 195.8MB, backup size: 195.8MB

[postgres@pgsrvdev 15]$
```

**Figure 11.1: PgBackRest- list backup**

2. **Perform a dummy transaction:** Connect to the PostgreSQL database using a client tool or **psql** and execute a dummy transaction to create a reference point for recovery:

```
DO $$

BEGIN

FOR r IN 1..1000 LOOP
    INSERT INTO testings(id) VALUES (r);
END LOOP;

END;

$$;
```

This transaction will serve as a reference point during the PITR process.

**Note: The inclusion of a dummy transaction in the scenario is for illustrative purposes to demonstrate the importance of having a reference point for recovery during the PITR process. It is essential to understand the concept of having a recovery reference point, but the actual execution of a dummy transaction may not be necessary in a real-world scenario.**

3. **Identify the desired recovery WAL segment:** Look for the WAL segment that corresponds to the recovery point to restore by executing the following command:

```
SELECT * FROM pg_ls_waldir();
```

Referring to [Figure 11.2](#), by examining this output, we can identify the WAL segment names and their respective modification timestamps. This information can be used to determine the recovery target time for performing a point-in-time recovery:

name	size	modification
00000003.history	85	2023-06-18 07:45:24+08
000000010000000000000005C	16777216	2023-06-18 07:45:24+08
00000002.history	42	2023-06-18 07:45:24+08
000000010000000000000005D	16777216	2023-06-18 07:45:24+08
000000020000000000000005E	16777216	2023-06-18 07:45:24+08
000000030000000000000005F	16777216	2023-06-18 07:45:25+08
0000000400000000000000060	16777216	2023-06-18 09:43:04+08
00000004.history	128	2023-06-18 07:45:25+08
0000000400000000000000061	16777216	2023-06-18 09:44:03+08
0000000400000000000000061.00000028.backup	374	2023-06-18 09:44:03+08
0000000400000000000000062	16777216	2023-06-18 10:04:50+08
0000000400000000000000063	16777216	2023-06-18 10:08:12+08
(12 rows)		

**Figure 11.2:** PgBackRest- WAL detail

Among the listed WAL files, we are specifically choosing the WAL file **0000000400000000000000062** and a modification timestamp of **2023-06-18 10:04:50+08**.

By selecting this WAL file as the recovery target, we aim to restore the database up to the state captured at the time represented by the chosen WAL segment. The timestamp, **2023-06-18 10:04:50+08**, indicates the specific recovery target time. This means that during the PITR process, the database will be restored to the state just before this timestamp, including all transactions and changes up until that point.

4. **Set the recovery target:** Open the PostgreSQL configuration file **postgresql.conf** and add the following

line to specify the recovery target time:

```
recovery_target_time = '2023-06-18 10:04:50+08'
```

By setting **recovery\_target\_time** to **2023-06-18 10:04:50+08**, we are instructing PostgreSQL to restore the database up to the state just before the specified timestamp.

5. **Stop the PostgreSQL service:** Execute the following command to stop the PostgreSQL service and this will ensure that the database is not running during the recovery process:

```
systemctl stop postgresql-15.service
```

6. **Clean-up PGDATA directory:** Before initiating the restore process, it is important to clean up the PGDATA directory to remove any existing database files. This can be done by deleting all the files and directories inside the PGDATA directory. However, exercise caution and ensure you have a backup of the necessary files before proceeding:

```
# Backup the Data directory
```

```
mv data Data_Bkp_Before_PITR
```

```
# Create a new data directory for PostgreSQL and  
assign permission
```

```
mkdir data
```

```
chmod 700 data
```

7. **Specify recovery target and perform the PITR Recover:** To perform a PITR recovery with the **pgbackrest** tool and restore the database to a specific point in time, we need to include the **--type=time** and **--target options** in the **pgbackrest** command:

```
pgbackrest --stanza=pgbr_DB_Backup --type=time --target="2023-  
06-18 10:04:50+08" restore
```

Referring to *Figure 11.3*, the above command executes a restore operation for PITR using the **pgbackrest** tool. By specifying **--target** to **2023-06-18 10:04:50+08**, we are instructing PostgreSQL to restore the database up to the state just before the specified timestamp:

```

| postgres@pgsrpdev 15|$ pgbackrest --stanza=pgbr_DB_Backup --type=time --target="2023-06-18 10:04:50+08" restore
2023-06-18 11:17:45.777 P00  INFO: restore command begin 2.46: --exec-id=162405-3bf0aa37 --log-level-console=detail --log-level-file=detail --log-path=/pgd
2023-06-18 11:17:45.795 P00  INFO: repol: restore backup set 20230618-094303F, recovery will start at 2023-06-18 09:43:08
2023-06-18 11:17:45.795 P00  DETAIL: check '/var/lib/pgsql/15/data' exists
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/base'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/base/1'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/base/1/21588'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/base/4'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/base/5'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/global'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/log'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/pg_commit_ts'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/pg_dynshmem'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/pg_logical'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/pg_logical/mappings'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/pg_logical/snapshots'
2023-06-18 11:17:45.796 P00  DETAIL: create path '/var/lib/pgsql/15/data/pg_multixact'
<<<<<< Output Truncated >>>>>>
2023-06-18 11:18:01.500 P00  DETAIL: sync path '/var/lib/pgsql/15/data/pg_stat_tmp'
2023-06-18 11:18:01.500 P00  DETAIL: sync path '/var/lib/pgsql/15/data/pg_subtrans'
2023-06-18 11:18:01.500 P00  DETAIL: sync path '/var/lib/pgsql/15/data/pg_tblspc'
2023-06-18 11:18:01.500 P00  DETAIL: sync path '/var/lib/pgsql/15/data/pg_twophase'
2023-06-18 11:18:01.500 P00  DETAIL: sync path '/var/lib/pgsql/15/data/pg_wal'
2023-06-18 11:18:01.500 P00  DETAIL: sync path '/var/lib/pgsql/15/data/pg_wal/archive_status'
2023-06-18 11:18:01.500 P00  DETAIL: sync path '/var/lib/pgsql/15/data/pg_xact'
2023-06-18 11:18:01.500 P00  INFO: restore global/pg_control (performed last to ensure aborted restores cannot be started)
2023-06-18 11:18:01.500 P00  DETAIL: sync path '/var/lib/pgsql/15/data/global'
2023-06-18 11:18:01.501 P00  INFO: restore size = 3.8GB, file total = 2429
2023-06-18 11:18:01.501 P00  INFO: restore command end: completed successfully (15727ms)
| postgres@pgsrpdev 15|

```

**Figure 11.3: PgBackRest- PITR recovery**

8. **Start the PostgreSQL service:** Start the PostgreSQL service using the following command and this will bring the database back online after the restore process:

```
systemctl start postgresql-15.service
```

9. **Verify the restore:** Connect to the PostgreSQL database and check if the restore was successful. You can query the testings table to verify if the dummy transaction records are present.

**Note:** It is important to have a valid backup and WAL archives available for the desired recovery point. Additionally, make sure to adjust the commands and file paths based on your specific PostgreSQL setup.

**Note:** The provided steps are a general outline and may need to be adapted to your specific environment and configurations.

## Incremental/differential restore

Incremental/differential restore is a technique used to restore a PostgreSQL database to a specific point in time by applying a series of backups that capture incremental or differential changes. This approach allows you to restore the database efficiently without having to restore all the data from the initial full backup.

For the incremental/differential restore process in PostgreSQL, we will utilize several reliable tools mentioned in the previous chapter on *Backup*. We will refer to the backup taken in the previous chapter as our restore point. The tools we will use for the Incremental/differential restore process are pgBackRest, pg\_probackup, and Barman.

By following the steps outlined in the recipe, we can restore the full backup first and then apply the incremental or differential backups in the order they were created. This process ensures that your database is brought back to the desired state, incorporating all the incremental or differential changes made since the full backup. Incremental/differential restore is a valuable for minimizing downtime and data loss in critical database environments.

## Recipe 92: Incremental/differential restore with Barman

In this recipe, we explore an advanced approach to database recovery using Barman. Focusing on incremental and differential restoration techniques, these steps guide you through a strategic process to efficiently restore your PostgreSQL database, minimizing downtime and ensuring data consistency. Let us delve into the precise steps for an effective incremental/differential restore with Barman:

- 1. Prepare the database backup files:** The initial full backup file taken with Barman. Based on the [Figure 11.4](#), refer to the recipe *Install and configure Barman* in the previous chapter and *Backup* to obtain a list of full backups created using the Barman tool.

```
[barman@pgsrdev .ssh]$ barman backup MY_PG_SERVER_DEV
Starting backup using rsync-concurrent method for server MY_PG_SERVER_DEV in /var/lib/barman/MY_PG_SERVER_DEV/base/20230527T105132
Backup start at LSN: 0/9000028 (000000010000000000000009, 00000028)
This is the first backup for server MY_PG_SERVER_DEV
WAL segments preceding the current backup have been found:
    000000010000000000000006 from server MY_PG_SERVER_DEV has been removed
    000000010000000000000007 from server MY_PG_SERVER_DEV has been removed
Starting backup copy via rsync/SSH for 20230527T105132 (2 jobs)
Copy done (time: 1 second)
This is the first backup for server MY_PG_SERVER_DEV
Asking PostgreSQL server to finalize the backup.
Backup size: 43.6 MiB
Backup end at LSN: 0/9000100 (000000010000000000000009, 00000100)
Backup completed (start time: 2023-05-27 10:51:33.500948, elapsed time: 2 seconds)
Processing xlog segments from file archival for MY_PG_SERVER_DEV
    000000010000000000000008
    000000010000000000000009
    000000010000000000000009.00000028.backup
[barman@pgsrdev .ssh]$
```

**Figure 11.4:** Barman- full backup

The incremental or differential backup files taken with Barman, capturing changes made after the full backup. Refer to the recipe *incremental/differential backup* in the previous chapter and backup to obtain a list of incremental backups created using the Barman tool.

2. **Stop PostgreSQL service:** Before initiating the restore process, stop the PostgreSQL service to prevent conflicts or data corruption during the restore operation. Execute the following command to stop the PostgreSQL service:

```
systemctl stop postgresql-15.service
```

3. **Identify the restore point:** Execute the following command to retrieve a list of backups for the PostgreSQL server taken from barman tool:

```
barman list-backup MY_PG_SERVER_DEV
```

Refer to [Figure 11.5](#). Each backup entry includes the backup ID, timestamp, size of the backup, and the size of the WAL associated with the backup.

```
[barman@pgsrvdev ~]$ barman list-backup MY_PG_SERVER_DEV
MY_PG_SERVER_DEV 20230619T221636 - Mon Jun 19 22:16:39 2023 - Size: 443.6 MiB - WAL Size: 0 B
MY_PG_SERVER_DEV 20230527T194307 - Sat May 27 19:43:56 2023 - Size: 443.5 MiB - WAL Size: 144.0 MiB
MY_PG_SERVER_DEV 20230527T194211 - Sat May 27 19:42:14 2023 - Size: 440.1 MiB - WAL Size: 32.0 MiB
MY_PG_SERVER_DEV 20230527T193527 - Sat May 27 19:40:02 2023 - Size: 440.1 MiB - WAL Size: 32.0 MiB
MY_PG_SERVER_DEV 20230527T192954 - Sat May 27 19:34:13 2023 - Size: 405.5 MiB - WAL Size: 80.0 MiB
MY_PG_SERVER_DEV 20230527T191436 - Sat May 27 19:14:40 2023 - Size: 59.7 MiB - WAL Size: 640.0 MiB
MY_PG_SERVER_DEV 20230527T191341 - Sat May 27 19:13:42 2023 - Size: 59.6 MiB - WAL Size: 32.0 MiB
MY_PG_SERVER_DEV 20230527T105132 - Sat May 27 10:51:35 2023 - Size: 59.6 MiB - WAL Size: 32.0 MiB
[barman@pgsrvdev ~]$
```

**Figure 11.5:** Barman- list backup

To determine the specific restore point for database restoration using Barman, we can consider the available full back-up or incremental backups. It is important to note that Barman can be instructed to use links configuration for incremental backups through the reuse-backup configuration option.

Referring to [Figure 11.6](#) and [Figure 11.7](#), assuming the issue in the database that occurred after May 27, 2023, we can use the backup taken on May 27, 2023, at 19:43:56 to restore the database.

The most recent full backup that we have is as follows:

```
[barman@pgsrvdev ~]$ barman show-backup MY_PG_SERVER_DEV 20230527T191436
Backup 20230527T191436:
  Server Name      : MY_PG_SERVER_DEV
  System Id        : 7231531554300609974
  Status           : DONE
  PostgreSQL Version : 150002
  PGDATA directory : /var/lib/pgsql/15/data

  Base backup information:
    Disk usage       : 43.7 MiB (59.7 MiB with WALs)
    Incremental size : 43.7 MiB (-0.00%)
    Timeline          : 1
    Begin WAL        : 000000010000000000000000000000D
    End WAL          : 000000010000000000000000000000D
    WAL number       : 1
    Begin time       : 2023-05-27 19:14:36.325393+08:00
    End time         : 2023-05-27 19:14:40.306704+08:00
    Copy time        : less than one second
    Estimated throughput : 51.7 MiB/s (2 jobs)
    Begin Offset     : 40
    End Offset       : 256
    Begin LSN        : 0/D000028
    End LSN          : 0/D000100

  WAL information:
    No of files      : 40
    Disk usage        : 640.0 MiB
    WAL rate          : 125.80/hour
    Last available    : 00000001000000000000000000000035

  Catalog information:
    Retention Policy   : not enforced
    Previous Backup    : 20230527T191341
    Next Backup         : 20230527T192954
[barman@pgsrvdev ~]$
```

**Figure 11.6:** Barman- show full backup detail

The most recent incremental backup that we have is:

```
[barman@pgsrvdev ~]$ barman show-backup MY_PG_SERVER_DEV 20230527T194307
Backup 20230527T194307:
  Server Name      : MY_PG_SERVER_DEV
  System Id        : 7231531554300609974
  Status           : DONE
  PostgreSQL Version : 150002
  PGDATA directory : /var/lib/pgsql/15/data

  Base backup information:
    Disk usage       : 427.5 MiB (443.5 MiB with WALs)
    Incremental size : 384.3 MiB (-10.12%)
    Timeline          : 1
    Begin WAL        : 000000010000000000000000003E
    End WAL          : 000000010000000000000000003E
    WAL number       : 1
    Begin time       : 2023-05-27 19:43:07.674028+08:00
    End time         : 2023-05-27 19:43:56.778562+08:00
    Copy time        : 3 seconds
    Estimated throughput : 121.1 MiB/s (2 jobs)
    Begin Offset     : 40
    End Offset       : 38528
    Begin LSN        : 0/3E000028
    End LSN          : 0/3E009680

  WAL information:
    No of files      : 9
    Disk usage       : 144.0 MiB
    WAL rate         : 0.02/hour
    Last available   : 0000000100000000000000000047

  Catalog information:
    Retention Policy : not enforced
    Previous Backup  : 20230527T194211
    Next Backup       : 20230619T221636
[barman@pgsrvdev ~]$
```

*Figure 11.7: Barman- show incremental backup detail*

**Note:** By enabling the reuse-backup = link option in the Barman configuration, Barman utilizes hard links when creating incremental backups. This means that the incremental backups will share unchanged files with the previous backup, optimizing storage usage and reducing backup time.

4. **Restore the incremental or differential backups:** Next, proceed to restore the database backups in the order they were created after the full backup. Employing the Barman tool once

again, specify the restore point, backup name, and target directory to apply each incremental backup:

```
barman recover --remote-ssh-command "ssh postgres@pgsrvdev"  
MY_PG_SERVER_DEV 20230527T194307 /var/lib/pgsql/15/data
```

In this scenario, we use the above command to restore the database to the existing data path, which is **/var/lib/pgsql/15/data**. However, we can replace the existing data path with a new path if needed.

5. **Start PostgreSQL service:** Upon completing the restore process for all the database backups, start the PostgreSQL service, making the restored database accessible again:

```
systemctl stop postgresql-15.service
```

With the help of Barman's efficient backup and restore capabilities, we have successfully recovered your critical PostgreSQL database with the desired backup, ensuring data integrity and minimizing downtime.

## Incremental/differential restore with pgBackRest

In this recipe, we focus on an adept approach to database recovery using pgBackRest. Specifically delving into incremental and differential restoration methods, these steps guide you through an efficient process for restoring your PostgreSQL database. By adopting this strategy, you can minimize downtime and selectively recover data changes, ensuring a precise and streamlined recovery process. Let us explore the detailed steps for an effective incremental/differential restore with pgBackRest.

1. **Prepare the database backup files:** The initial full backup file taken with pgBackRest. Based on the [Figure 11.8](#), refer to the recipe *Install and configure pgBackRest for PostgreSQL* in the previous chapter and backup to obtain a list of full backups created using the pgBackRest tool.

The incremental or differential backup files taken with pgBackRest, capturing changes made after the full backup. Based on the [Figure 11.8](#), refer to the recipe *Incremental/Differential Backup* in the previous chapter and backup to obtain a list of incremental/differential backups created using the pgBackRest tool. Please refer to the following figure:

```
[postgres@pgsrvdev ~]$ pgbackrest info --stanza=pgbr_DB_Backup
stanza: pgbr_DB_Backup
status: ok
cipher: none

db (current)
wal archive min/max (15): 000000010000000000000058/00000001000000000000005C

full backup: 20230528-185132F
timestamp start/stop: 2023-05-28 18:51:32 / 2023-05-28 18:52:33
wal start/stop: 00000001000000000000005A / 00000001000000000000005A
database size: 3.7GB, database backup size: 3.7GB
repo1: backup set size: 194MB, backup size: 194MB

diff backup: 20230528-185132F_20230528-192018D
timestamp start/stop: 2023-05-28 19:20:18 / 2023-05-28 19:21:40
wal start/stop: 00000001000000000000005C / 00000001000000000000005C
database size: 3.8GB, database backup size: 533.3MB
repo1: backup set size: 195.8MB, backup size: 70.4MB
backup reference list: 20230528-185132F

[postgres@pgsrvdev ~]$
```

**Figure 11.8:** pgBackRest- list backup

2. **Stop PostgreSQL service:** Before initiating the restore process, stop the PostgreSQL service to prevent conflicts or data corruption during the restore operation. Execute the following command to stop the PostgreSQL service:

```
systemctl stop postgresql-15.service
```

3. **Identify the restore point:** Locate the initial full backup taken with Barman and identify the incremental backups capturing changes made since that full backup. Determine the specific restore point we wish to restore the database to, based on the incremental or differential backups available.

4. **Restore the backup:** Next, proceed to restore the database backups. By executing the following command, Employ the pgBackRest tool to start the database restore in the order they were created.

```
pgbackrest --log-level-console=detail --stanza=pgbr_DB_Backup restore
```

Referring to the output in the [Figure 11.9](#), in the command **pgbackrest --log-level-console=detail --stanza=pgbr\_DB\_Backup restore**, the restore command is executed without specifying a specific backup set or restore point. Instead, **pgBackRest** automatically selects the appropriate backup set for restoration based on its internal metadata and the configuration of the system.

The line **repo1: restore backup set 20230528-**

**185132F\_20230528-192018D**, recovery will start at **2023-05-28 19:20:18** indicates that pgBackRest has identified the **backup set 20230528-185132F\_20230528-192018D** as the appropriate set to restore from. The restore process will begin at the specified time, automatically determined by pgBackRest based on the backup metadata. Please refer to the following figure:

```
postgres@pgs:~$ pgbackrest --log-level-console=detail --stanza=pgbr_DB_Backup restore
2023-05-17 16:24:13.449 P00 INFO: restore command begin 2.46: --exec-id=152292-1lb8d3 --log-level-console=detail --log-level-file=detail --log-path=/pgbr_log
2023-05-17 16:24:13.465 P00 INFO: repool: restore backup set 20230528 185132F_20230528 192018D, recovery will start at 2023 05 28 19:20:18
WARN: unknown user 'root' in backup manifest mapped to current user
WARN: unknown group 'root' in backup manifest mapped to current group
2023-05-17 16:24:13.466 P00 DETAIL: check '/var/lib/pgsql/15/dato' exists
2023-05-17 16:24:13.466 P00 DETAIL: create path '/var/lib/pgsql/15/data/base/1'
2023-05-17 16:24:13.466 P00 DETAIL: create path '/var/lib/pgsql/15/data/base/1588'
2023-05-17 16:24:13.466 P00 DETAIL: create path '/var/lib/pgsql/15/data/base/4'
2023-05-17 16:24:13.466 P00 DETAIL: create path '/var/lib/pgsql/15/data/base/5'
2023-05-17 16:24:13.466 P00 DETAIL: create path '/var/lib/pgsql/15/data/global'
2023-05-17 16:24:13.466 P00 DETAIL: create path '/var/lib/pgsql/15/data/log'
2023-05-17 16:24:13.466 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_commtx_ts'
2023-05-17 16:24:13.466 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_csynshmem'
2023-05-17 16:24:13.467 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_logical'
2023-05-17 16:24:13.467 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_logical/mappings'
2023-05-17 16:24:13.467 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_logical/snapshots'
2023-05-17 16:24:13.467 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_multixact'
<<<<<< Output Truncated >>>>>>
2023-05-17 16:24:30.015 P00 DETAIL: sync path '/var/lib/pgsql/15/data/pg_stat'
2023-05-17 16:24:30.015 P00 DETAIL: sync path '/var/lib/pgsql/15/data/pg_stat_tmp'
2023-05-17 16:24:30.015 P00 DETAIL: sync path '/var/lib/pgsql/15/data/pg_subtrans'
2023-05-17 16:24:30.015 P00 DETAIL: sync path '/var/lib/pgsql/15/data/pg_16spu'
2023-05-17 16:24:30.015 P00 INFO: restore_global/pg_control (performed last to ensure aborted restores cannot be started)
2023-05-17 16:24:30.015 P00 DETAIL: sync path '/var/lib/pgsql/15/data/global'
2023-05-17 16:24:30.015 P00 INFO: restore size = 3.86B, file total = 2413
2023-05-17 16:24:30.016 P00 INFO: restore command end: completed successfully (16570ms)
[postgres@pgs:~$]
```

**Figure 11.9: PgBackRest- restore database**

Therefore, in this scenario, the restore point was automatically selected by the pgBackRest tool, ensuring the correct backup set is restored without the need for manual intervention.

5. **Restore the specific backups set:** Ensure to have the full backup (**20230528-185132F**) available for restoration set. To obtain the full backup set ID, please refer to *Step 1* of this recipe and [Figure 11.1](#). Use the following command to restore only the full backup and skip the differential backup.

```
pgbackrest --set=20230528-185132F --log-level-console=detail --
stanza=pgbr_DB_Backup restore
```

Explanation of the above command options:

- **--set=20230528-185132F** command option designates the backup set for restoration, specifically targeting the full backup (**20230528-185132F**) generated in *Step 1* of this recipe.
- **--log-level-console=detail** sets the console log level to display detailed information during the restore process.

- **--stanza=pgbr\_DB\_Backup** specifies the backup configuration stanza to use for the restore.

```
[postgres@pgsqlvdev 15]$ pgbackrest --sel=20230528-185132F --log-level=console+detail --stanza=pgbr_DB_Backup restore
2023-05-17 17:26:48.390 P00 INFO: restore command begin 2.46: --exec-id=153064-0249a6a5 --log-level-console=detail --log-path=/pgsql.log
2023-05-17 17:26:48.407 P00 INFO: rep01: restore backup set 20230528-185132F, recovery will start at 2023-05-28 18:51:32
WARN: unknown user 'root' in backup manifest mapped to current user
WARN: unknown group 'root' in backup manifest mapped to current group
2023-05-17 17:26:48.407 P00 DETAIL: check '/var/lib/pgsql/15/data' exists
2023-05-17 17:26:48.407 P00 DETAIL: create path '/var/lib/pgsql/15/data/base'
2023-05-17 17:26:48.407 P00 DETAIL: create path '/var/lib/pgsql/15/data/base/1'
2023-05-17 17:26:48.407 P00 DETAIL: create path '/var/lib/pgsql/15/data/base/12588'
2023-05-17 17:26:48.407 P00 DETAIL: create path '/var/lib/pgsql/15/data/base/12588/1'
2023-05-17 17:26:48.407 P00 DETAIL: create path '/var/lib/pgsql/15/data/base/5'
2023-05-17 17:26:48.407 P00 DETAIL: create path '/var/lib/pgsql/15/data/global'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/log'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_commit_ts'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_csynchmem'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_logical'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_logical/mappings'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_logical/snapshots'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_multixact'
<<<<<< Output Truncated >>>>>>
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_multixact/members'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_multixact/offsets'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_notify'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_replslot'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_serial'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_snapshots'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_stat'
2023-05-17 17:26:48.408 P00 DETAIL: create path '/var/lib/pgsql/15/data/pg_stat_tmp'
2023-05-17 17:27:07.066 P00 INFO: restore global/pg_control (performed last to ensure aborted restores cannot be started)
2023-05-17 17:27:07.066 P00 DETAIL: sync path '/var/lib/pgsql/15/data/global'
2023-05-17 17:27:07.067 P00 INFO: restore size = 3.76B, file total = 2413
2023-05-17 17:27:07.067 P00 INFO: restore command end: completed successfully (10600ms)
```

**Figure 11.10:** PgBackRest: restore specific backup

Based on this execution output in the [Figure 11.10](#), it can be concluded that the restoration of the full backup **20230528-185132F** was performed successfully and skip the differential backup, ensuring database is restored to a specific point in time.

6. **Start PostgreSQL service:** Upon completing the restore process for all the database backups, start the PostgreSQL service, making the restored database accessible again.

```
systemctl stop postgresql-15.service
```

With the help of PgBackRest's efficient backup and restore capabilities, we have successfully recovered PostgreSQL database to the desired point in time, ensuring data integrity and minimizing downtime.

## Recipe 93: Restoring database with Barman

Restoring from Barman is a straightforward and reliable process for recovering a PostgreSQL database. Barman provides a user-friendly interface for selecting a specific backup and restoring the database to a desired point-in-time. By leveraging Barman's capabilities, database administrators can confidently restore the database to a previous state in the event of data corruption, hardware failure, or accidental deletion.

In this recipe, we will walk through the process of restoring a database using Barman in PostgreSQL.

**Prerequisite:** Before restoring a database using Barman, it is essential to have a backup of the database taken using Barman itself. When it comes to restoring a database, Barman utilizes the backups it has taken to perform the restore operation accurately:

- 1. Prepare the database backup:** Before delving into the restoration process, it is crucial to ensure the availability of a recent backup for the PostgreSQL database intended for restoration. Referencing the preceding chapter, [Chapter 10, Backup](#), and the recipe titled *Install and configure Barman*, we outlined the steps for creating backups tailored to the database using the Barman tool.

By referring this recipe, you can access a list of backups created for potential database restoration. An example of such a backup, performed in that chapter, is stored in the directory **/var/lib/barman/MY\_PG\_SERVER\_DEV/base/20230527T105132**.

Let us proceed with the steps to prepare the database backup for restoration.

- 2. Identify the backup:** Verify the list of available backups by executing the following command:

```
barman list-backup MY_PG_SERVER_DEV
```

Referring to [Figure 11.11](#), the above command execution lists the available backups for the server named **MY\_PG\_SERVER\_DEV** along with their respective details. Each backup entry includes the backup ID, timestamp, size of the backup data, and the size of the corresponding WAL files. Note down the backup ID or label of the backup that to be restore. Please refer to the following figure:

```
[barman@pgsrvdev base]$ barman list-backup MY_PG_SERVER_DEV
MY_PG_SERVER_DEV 20230527T194307 - Sat May 27 19:43:56 2023 - Size: 443.5 MiB - WAL Size: 112.0 MiB
MY_PG_SERVER_DEV 20230527T194211 - Sat May 27 19:42:14 2023 - Size: 440.1 MiB - WAL Size: 32.0 MiB
MY_PG_SERVER_DEV 20230527T193527 - Sat May 27 19:40:02 2023 - Size: 440.1 MiB - WAL Size: 32.0 MiB
MY_PG_SERVER_DEV 20230527T192954 - Sat May 27 19:34:13 2023 - Size: 405.5 MiB - WAL Size: 80.0 MiB
MY_PG_SERVER_DEV 20230527T191436 - Sat May 27 19:14:40 2023 - Size: 59.7 MiB - WAL Size: 640.0 MiB
MY_PG_SERVER_DEV 20230527T191341 - Sat May 27 19:13:42 2023 - Size: 59.6 MiB - WAL Size: 32.0 MiB
MY_PG_SERVER_DEV 20230527T105132 - Sat May 27 10:51:35 2023 - Size: 59.6 MiB - WAL Size: 32.0 MiB
[barman@pgsrvdev base]$
```

**Figure 11.11:** Barman: list backup

- 3. Stop PostgreSQL service:** Before initiating the restore process, stop the PostgreSQL service to prevent conflicts or data corruption during the restore operation. Execute the following command to stop the PostgreSQL service:

```
systemctl stop postgresql-15.service
```

- 4. Restore the database:** Execute the following command to restore the database from a specific backup:

```
barman recover --remote-ssh-command "ssh postgres@pgsrvdev"  
MY_PG_SERVER_DEV 20230527T194307 /var/lib/pgsql/15/data/
```

Refer to *Figure 11.12*, the above execution shows the result of running the barman recover command to initiate a remote restore for the server **MY\_PG\_SERVER\_DEV** using the backup labelled **20230527T194307**.

During the restore process, the base backup is copied, followed by the necessary WAL segments. Archive status files are generated to track the recovery progress. The execution concludes by confirming the completion of the recovery process, providing the start time and elapsed time of the operation. Please refer to the following figure:

```
[barman@pgsrvdev ~]$ barman recover --remote-ssh-command "ssh postgres@pgsrvdev" MY_PG_SERVER_DEV 20230527T194307 /var/lib/pgsql/15/data/  
Starting remote restore for server MY_PG_SERVER_DEV using backup 20230527T194307  
Destination directory: /var/lib/pgsql/15/data/  
Remote command: ssh postgres@pgsrvdev  
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.conf line 257: archive_command = false  
postgresql.auto.conf line 11: archive_command = false  
  
Recovery completed (start time: 2023-06-19 21:02:42.475212+08:00, elapsed time: 7 seconds)  
Your PostgreSQL server has been successfully prepared for recovery!  
[barman@pgsrvdev ~]$
```

*Figure 11.12: Barman: recover database*

**Note:** After performing a restore with Barman, it is important to note that `archive_command` settings in the `postgresql.conf` has been set to false. To ensure the proper functioning of the database and enable continuous WAL archiving for data protection, it is necessary to review and adjust the `archive_command` setting after the restore.

- 5. Start PostgreSQL service:** After the restore process completes, start the PostgreSQL service again to allow normal database operations. Execute the command to start PostgreSQL:

```
systemctl stop postgresql-15.service
```

- 6. Verify the database:** Connect to the restored database using a PostgreSQL client tool, such as **psql** and execute test queries to ensure that the data has been successfully restored.

## Comparing dropped versus damaged table recovery

Navigating the intricacies of dropped versus damaged table recovery in PostgreSQL unveils a key facet of database management and data restoration proficiency. Recognizing these distinctions is pivotal for anyone tasked with maintaining databases effectively. The detailed comparison table below provides a breakdown to guide you through the different recovery scenarios.

	Dropped table/database recovery	Damaged Table/database recovery
Scenario	Accidental deletion or dropping of tables or databases.	Data corruption or integrity issues in tables or databases.
Recovery approach	Restore the entire database from a recent backup and extract the dropped data.	Use backups to restore the affected tables or databases to a consistent state.
Goal	Recover accidentally deleted data.	Repair and recover data integrity issues.
Backup strategy	Important to have regular backups in place.	Crucial to have a comprehensive backup strategy.
Prevention measures	Regularly create backups and implement monitoring systems.	Implement monitoring and error detection mechanisms.
Key considerations	Availability of recent backups and proper restoration process.	Identifying and addressing the cause of data corruption.

**Table 11.1:** Dropped table versus damaged table

By referring to this table, readers can quickly grasp the differences between dropped table/database recovery and damaged table/database recovery in PostgreSQL, including the scenarios they

address, the recovery approaches involved, the tools and techniques used, and the importance of a robust backup strategy and prevention measures for each scenario.

## Recipe 94: Working with tables recovery

In PostgreSQL, working with tables recovery allows you to recover specific tables or their data in case of accidental deletion, data corruption, or other table-related issues. This recipe will guide through the process of recovering tables using the built-in tools and features in PostgreSQL.

Imagine a database administrator responsible for managing a PostgreSQL database in a production environment. One day, an application user accidentally drops the sales table. To rectify the situation and recover the lost data, the forthcoming steps will walk you through the process of table-level recovery, focusing specifically on the affected sales table in the public schema of the postgres database.

1. **Identify the affected table:** Confirm that the sales table in the public schema is the one that needs to be recovered.
2. **Enable PITR:** Refer to the recipe *PostgreSQL Continuous Archiving and Archive Log Examples* in [Chapter 5, Transaction Log](#), for instructions on how to modify the PostgreSQL configuration file. Enable archiving of transaction logs and configure an appropriate retention period to facilitate PITR in PostgreSQL.
3. **Take a database backup:** Perform a backup of the PostgreSQL database using the **pg\_dump** command:

```
pg_dump -U postgres -d postgres -F c -f Logical_Bkp_20230618.dump
```

Refer to the recipe *Working with Logical backup* in [Chapter 10, Backup](#), for detail instructions on how to perform logical backup using **pg\_dump**.

**Note:** When it comes to table recovery, the **pg\_basebackup** utility is typically used for full database recovery, where the backup is restored to a different database server rather than the one where the backup was taken. This process involves restoring the entire database.

However, if you only need to restore individual tables instead of the entire database, the **pg\_dump** utility provides a more suitable option. **pg\_dump** allows you to take a logical backup of specific tables or even specific data within tables. With the generated SQL file from **pg\_dump**, you can selectively restore the desired tables or data to the original or a different database.

Using **pg\_dump** for table recovery provides flexibility and granularity, allowing you to restore only the necessary tables or data, reducing the overall restore time and resource consumption compared to restoring the entire database with **pg\_basebackup**.

4. **Assuming sales table dropped by the user:** Before restoring the table, we need to drop the existing table (sales) from the existing database.

**Note: Please note that this step is for the illustration purpose of this recipe and the sales table will be dropped from the database at this point. Ensure that you have a backup of your data before proceeding.**

5. **Restore the table:** Use the **psql** command to restore the sales tables from the backup file created in step 3 of this recipe by executing the following command:

```
pg_restore -d postgres -t sales Logical_Bkp_20230618.dump
```

The above command restores a table named sales from a logical backup file named **Logical\_Bkp\_20230618.dump** into the **postgres** database.

6. **Verify the recovery:** Connect to the database using **psql** and verify that the recovered sales table is accessible and contains the expected data.

**Note: The restoration of a single table using the backup taken from pg\_dump has been successfully completed. However, it is important to note that this method does not support PITR. As a result, there might be data loss if you need to recover the table to a specific point in time. For comprehensive data recovery and PITR capabilities, it is recommended to use other backup and restore methods such as pg\_basebackup or pg\_receivexlog.**

**as continuous archiving or base backups using tools like pg\_basebackup and pgbackrest.**

## Recipe 95: Working with schema level restore

Schema-level restore in PostgreSQL is a valuable technique that allows database administrators to selectively restore specific schemas within a database. By utilizing a recent backup and the **pg\_restore** command, administrators can target a particular schema and restore it to a previous state, effectively recovering from schema-level corruption or accidental data modifications. This approach ensures that only the desired schema is restored, while the rest of the database remains unaffected.

Schema-level restore provides granular control over the recovery process, allowing administrators to efficiently address schema-related issues and minimize downtime. This recipe will guide through the process of performing a schema-level restore using PostgreSQL:

- 1. Prepare the backup files:** Ensure to have a recent backup of the PostgreSQL database that includes the schemas that we want to restore.

In the previous chapter called *Backup*, we discussed a recipe called *Working with Schema level backup*. This recipe provides instructions on how to create backups specifically for the database schema. By referring to that recipe, we can find a list of backups that were created for restoring the schema. One such backup we performed in that chapter was saved in a file called **/pg\_backup/schema\_bkp\_202305.dump**. This backup file contains all the important information about the **customers** and **orders** schema.

- 2. Restore the schema from schema level backup:** To recover a particular schema from the backup, simply run the following command. This will restore the specific schema that you want to retrieve from the backup:

```
pg_restore -U postgres -d postgres  
/pg_backup/schema_bkp_202305.dump
```

The above command connects to the PostgreSQL database server and restores the schema from the backup file **schema\_bkp\_202305.dump** into the **postgres** database.

**Note:** In the previous scenario, we restored the schema using a schema-level backup file, which contained specific schema information. However, in this scenario, we will restore only one schema from a backup of the entire database. This method lets us choose and restore just the schema we want, while still having the option to access the full database backup if necessary.

3. **Restore the schema from the backup dump of entire database:** Before proceeding with the restore process in this recipe, let us verify the schema that we will be working with. In this scenario, we will be restoring two schemas named customers and orders from a backup dump file of the entire database.

Within the customers schema, we have the following table:

Schema	Tables
Customers	<b>cust_info</b>
Customers	<b>cust_sale</b>

**Table 11.2:** Schema level restore-object in customers schema

And within the orders schema, we have the following table:

Schema	Tables
orders	<b>order_info</b>
orders	<b>order_sale</b>

**Table 11.3:** Schema level restore- Object in orders schema

These tables hold specific data and structure related to their respective schemas. We will be restoring these schemas to their previous state using the backup dump file, ensuring that the tables and their contents are recovered.

Open a terminal or command prompt and navigate to the directory where you want to store the backup dump file and take a backup of the entire database using the **pg\_dump** command:

```
pg_dump -Fc -f /pg_backup/PG_BKP.dump -d postgres
```

**Note:** In this scenario, it is assumed that both the customers and orders schemas were accidentally dropped from the database. As a result, all the tables and their data within these schemas were lost. To recover from this situation, we will be performing a restore process using a backup dump file. This backup dump file contains the necessary information to recreate the customers and orders schemas, along with their respective tables and data.

Execute the following command to generate a summarized TOC of the backup dump created with **pg\_dump**:

```
pg_restore --list "/pg_backup/PG_BKP.dump" --file="PG_BKP.list"
```

Referring to [Figure 11.13](#), the above command is used to create a list file: **PG\_BKP.list**, that contains the summary of the backup dump's contents. By examining the generated list file (**PG\_BKP.list**), we can locate the line or lines that pertain to the schema/objects we want to restore.

```
[postgres@pgsrvdev pg_backup]$ cat PG_BKP.list
;
; Archive created at 2023-06-19 13:06:33 +08
;     dbname: postgres
;     TOC Entries: 2003
;     Compression: -1
;     Dump Version: 1.14-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 15.2
;     Dumped by pg_dump version: 15.2
;
;
;
; Selected TOC Entries:
;
13; 2615 38124 SCHEMA - customers postgres
14; 2615 38125 SCHEMA - orders postgres
537; 1259 38141 TABLE customers cust_info postgres
534; 1259 38126 TABLE customers cust_sale postgres
536; 1259 38136 TABLE orders order_info postgres
535; 1259 38131 TABLE orders order_sale postgres
7136; 0 38141 TABLE DATA customers cust_info postgres
7133; 0 38126 TABLE DATA customers cust_sale postgres
7135; 0 38136 TABLE DATA orders order_info postgres
7134; 0 38131 TABLE DATA orders order_sale postgres
[postgres@pgsrvdev pg_backup]$
```

**Figure 11.13:** Barman- Full backup

Once relevant schema information identified from the list file (**PG\_BKP.list**), we can proceed with the following command to restore the specific schema from the backup dump:

```
pg_restore -v -L PG_BKP.list -d postgres PG_BKP.dump
```

Referring to [\*\*Figure 11.14\*\*](#), by running the above command, the **pg\_restore** utility will read the list file and identify the necessary information to restore the specific schema from the backup dump. The restore operation will be performed on the specified target database (postgres in this case), allowing to recover the desired schema and its associated objects. Please refer to the following figure:

```
[postgres@pgsrvdev pg_backup]$ pg_restore -v -L PG_BKP.list -d postgres PG_BKP.dump
pg_restore: connecting to database for restore
pg_restore: creating SCHEMA "customers"
pg_restore: creating SCHEMA "orders"
pg_restore: creating TABLE "customers.cust_info"
pg_restore: creating TABLE "customers.cust_sale"
pg_restore: creating TABLE "orders.order_info"
pg_restore: creating TABLE "orders.order_sale"
pg_restore: processing data for table "customers.cust_info"
pg_restore: processing data for table "customers.cust_sale"
pg_restore: processing data for table "orders.order_info"
pg_restore: processing data for table "orders.order_sale"
[postgres@pgsrvdev pg_backup]$
```

**Figure 11.14:** pg\_restore- Restore with list file

4. **Verify the restored schema:** Connect to the PostgreSQL database using a client tool, such as **psql**. Referring to [Figure 11.15](#), use the command **\dn** to list the schemas and verify that the restored schema is present.

List of schemas	
Name	Owner
<hr/>	
cust_info	postgres
customers	postgres
orders	postgres
payments	postgres
products	postgres
<b>public</b>	pg_database_owner
transactions	postgres
tst	postgres
(8 rows)	
postgres=#	

**Figure 11.15:** pg\_restore- Schema list

That is it! We have successfully restored the **customers** and **orders** schema from the backup dump of the entire database.

## Conclusion

In conclusion, the chapter has equipped readers with the knowledge and practical skills required to successfully recover a PostgreSQL database. By delving into topics of crash recovery, dropped table/database recovery, damaged table/database recovery, and PITR, readers have gained an understanding of the intricacies involved in database restoration.

The chapter also provided valuable recipes, including full and PITR recovery, restoring databases with Barman, incremental/differential restore, tablespace recovery, tables recovery, schema-level restore, monitoring restore operations, working with recovery modes, and recovery target options.

By mastering these concepts and techniques, readers are now well-prepared to handle database recovery scenarios and ensure data resilience and continuity in their PostgreSQL environments.

As we transition to the next chapter, readers will embark on a journey into the critical realm of monitoring and diagnosis in PostgreSQL 15. The chapter will also delve into monitoring with system views, exploring statistics, and uncovering monitoring scripts, providing you with a toolkit to ensure the seamless operation of your PostgreSQL environment.

## **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

# Monitoring and Diagnosis

## Introduction

This chapter explains the monitoring and diagnosis in PostgreSQL, providing a comprehensive guide to fortify your setup by implementing effective monitoring practices. Monitoring and diagnosis are indispensable for maintaining the optimal performance and reliability of a database system. This chapter offers a collection of recipes that address common monitoring and diagnosis actions within PostgreSQL, equipping you with the tools and knowledge needed to ensure the smooth operation of your database. Furthermore, you will explore various monitoring policies tailored to different components of the database.

## Structure

In this chapter, we will cover the following topics:

- Introducing monitoring and diagnosis
- System resource monitoring
- Database monitoring
- Prometheus and Grafana
- Monitoring with system view
- Vacuum and bloat consideration
- Explore statistics
- Discovering monitoring script
- Explore fsync

## **Objectives**

The primary objective of this chapter is to provide a comprehensive understanding of monitoring techniques in PostgreSQL. We focus on introducing foundational concepts and the significance of monitoring and diagnosis in the PostgreSQL environment, preparing you to effectively navigate the complexities of database maintenance and optimization. Additionally, we guide you through monitoring and diagnosing the PostgreSQL database, including its components and operations, with a focus on addressing potential issues. The integration of powerful tools like Prometheus and Grafana is explored for holistic monitoring and visualization of PostgreSQL metrics.

The chapter also delves into considerations related to vacuuming and bloat, offering insights into strategic decisions for optimizing database maintenance and performance. Further, we explore the utilization of statistical data for superior query optimization and performance tuning, providing practical knowledge to enhance overall PostgreSQL operations. The ultimate goal is to equip you with the necessary skills to adeptly employ monitoring techniques in the PostgreSQL context, covering foundational concepts and advanced tool integration for efficient database monitoring and optimization.

## **Introducing monitoring and diagnosis**

Monitoring and diagnosis are critical aspects of managing and maintaining PostgreSQL databases. In order to ensure the smooth operation and performance of a PostgreSQL system, it is essential to monitor key metrics and diagnose any potential issues. This includes monitoring database performance, disk usage, memory consumption, query execution, and system health indicators. By implementing effective monitoring strategies, administrators can identify bottlenecks, track trends, and proactively address potential problems before they escalate. Additionally, diagnosing and troubleshooting issues requires in-depth knowledge of PostgreSQL's logging and error reporting capabilities, as well as familiarity with diagnostic tools like `pg_stat_activity`, `pg_stat_statements`, and `pgBadger`.

## **System resource monitoring**

System resource monitoring in PostgreSQL allows administrators to manage and optimize the performance of their database environment effectively. With enhanced monitoring capabilities, administrators can

track and analyze key system resources such as CPU usage, memory utilization, disk I/O, and network activity. The monitoring system provides real-time insights into the database's resource consumption, allowing administrators to identify bottlenecks, troubleshoot performance issues, and make informed decisions regarding resource allocation and capacity planning.

Additionally, PostgreSQL offers integration with popular monitoring tools, enabling seamless integration with existing monitoring infrastructures and facilitating proactive monitoring and alerting for efficient database management. Overall, system resource monitoring in PostgreSQL empowers administrators with the tools they need to ensure the optimal performance and stability of their PostgreSQL databases.

## **Database monitoring**

PostgreSQL introduces enhanced monitoring capabilities that provide real-time visibility into various database metrics, including query execution times, lock waits, connection activity, and transaction throughput.

Administrators can leverage this information to detect and diagnose performance issues, identify resource bottlenecks, and optimize query execution plans. Moreover, PostgreSQL offers improved integration with monitoring tools, allowing seamless integration into existing monitoring infrastructures. With comprehensive database monitoring in PostgreSQL, administrators can proactively identify and resolve issues, ensuring the reliability, scalability, and efficiency of their database deployments.

## **Discovering monitoring script**

Custom monitoring scripts play a vital role in ensuring the health and performance of your PostgreSQL database. These scripts are tailored to your specific requirements and allow you to proactively identify and address issues. The provided recipe covers a diverse set of monitoring aspects beyond the specifically mentioned five objects. It includes scripts for identifying long-running queries, monitoring table sizes, tracking connection counts, and measuring replication lag. These scripts serve as examples and can be customized based on specific monitoring requirements for your PostgreSQL environment.

## **Recipe 96: Utilizing script-based monitoring**

Developing custom monitoring scripts is crucial for obtaining insights into specific aspects of your PostgreSQL database. These scripts can be tailored to monitor various parameters such as performance, resource utilization, and potential issues. Below are recipe examples of monitoring scripts for different database objects:

- a. **Long-running queries:** Create a script to identify and log long-running queries. Save the following script as **monitor\_long\_running\_queries.sh**:

```
#!/bin/bash

# Set the threshold for long-running queries in seconds
THRESHOLD=60
# Log queries exceeding the threshold
psql -U your_username -d your_database -c "SELECT * FROM pg_stat_activity
WHERE state = 'active' AND now() - pg_stat_activity.query_start >= interval
'$THRESHOLD seconds';" >> /path/to/long_running_queries.log
```

This script checks for queries that have been active for more than the defined threshold and logs them for further analysis.

- b. **Table size monitoring:** Create a script to monitor the sizes of tables in your database. Save the following script as **monitor\_table\_size.sh**:

```
#!/bin/bash
# Get table sizes in MB
psql -U your_username -d your_database -c "SELECT schemaname, tablename,
pg_size.pretty(pg_total_relation_size(schemaname || '.' || tablename)) AS size FROM
pg_tables ORDER BY pg_total_relation_size(schemaname || '.' || tablename) DESC;" >> /path/to/table_sizes.log
```

This script retrieves and logs the total sizes of all tables in your database, helping you identify potential storage issues.

- c. **Connection monitoring:** Create a script to monitor the number of database connections. Save the following script as **monitor\_connections.sh**:

```
#!/bin/bash
# Get the number of connections
psql -U your_username -d your_database -c "SELECT datname, numbackends
FROM pg_stat_database;" >> /path/to/connection_count.log
```

This script logs the current number of connections to your database, helping you keep track of resource utilization.

d. **Replication lag:** Create a script to monitor replication lag in a streaming replication setup. Save the following script as **monitor\_replication\_lag.sh**:

```
#!/bin/bash
# Get replication lag in bytes
psql -U your_username -d your_database -c "SELECT
pg_is_in_recovery() AS is_slave,
pg_last_wal_receive_lsn() AS receive_lsn,
pg_last_wal_replay_lsn() AS replay_lsn,
pg_last_wal_receive_lsn() = pg_last_wal_replay_lsn() AS is_synced,
EXTRACT(EPOCH FROM now()) - EXTRACT(EPOCH FROM
pg_last_xact_replay_timestamp()) AS replication_lag_seconds,
pg_wal_lsn_diff(pg_last_wal_receive_lsn(), pg_last_wal_replay_lsn()) AS
replication_lag_bytes
FROM
pg_stat_replication;» >> /path/to/replication_lag.log
```

This script logs the replication lag for each streaming replication client, allowing you to identify potential issues in your replication setup.

## Prometheus and Grafana

Prometheus and Grafana play pivotal roles in enhancing monitoring and visualization capabilities for PostgreSQL databases. Prometheus, an open-source monitoring solution, offers a flexible and scalable platform for collecting metrics, alerting, and time series data storage. With its efficient data model and query language, Prometheus empowers administrators and developers to gain valuable insights into their PostgreSQL database performance, resource utilization, and overall system health. Complementing Prometheus, Grafana serves as a feature-rich visualization tool, allowing users to create interactive and customizable dashboards for monitoring key metrics and generating informative graphs. Together, Prometheus and Grafana provide tools for monitoring, analyzing, and optimizing PostgreSQL databases in the ever-evolving world of data management.

## Recipe 97: Managing Prometheus and Grafana

In this recipe, we will guide you through the process of configuring and handling Prometheus, postgres\_exporter, and Grafana for monitoring a PostgreSQL database. Prometheus is an open-source monitoring and alerting toolkit, while Grafana is a powerful visualization tool that allows you to create interactive dashboards and postgres\_exporter acts as an intermediary between Prometheus and PostgreSQL. Together, they provide a comprehensive monitoring solution for your PostgreSQL database.

### Pre-requisites:

- A running instance of PostgreSQL.
- A server or virtual machine to install Prometheus, postgres\_exporter and Grafana.

#### 1. Install and configure Prometheus:

- a. Download the latest version of Prometheus from official GitHub repositories <https://github.com/prometheus/prometheus/releases> and extract the downloaded package. Find the desired release version (for example, **release/2.45.0**):

```
[root@controller Mon_binary]# wget https://github.com/prometheus/prometheus/releases/download/v2.45.0/prometheus-2.45.0.linux-amd64.tar.gz
2023-07-01 15:58:09 -> https://github.com/prometheus/prometheus/releases/download/v2.45.0/prometheus-2.45.0.linux-amd64.tar.gz
Resolving github.com (github.com)... 26.205.243.166
Connecting to github.com (github.com)|26.205.243.166|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objects.githubusercontent.com/gcr-production/release-asset/5026923/75750d2b-0c1e-4ac6-bdee-941c2edfaa7X-Anon-AlgorismnA654-HMAC-SHA256&X-Anon-C-e
2023-07-01 15:58:09 -> https://objects.githubusercontent.com/gcr-production/release-asset/2e059e5030321/75750d2b-0c1e-4ac6-bdee-941c2edfaa7X-Anon-AlgorismnA654-HMAC-S
Resolving objects.githubusercontent.com (objects.githubusercontent.com)... 285.199.108.133, 285.199.110.133, 185.199.111.133, ...
Connecting to objects.githubusercontent.com (objects.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 9118594 (8.7MB) [application/octet-stream]
Saving to: /prometheus-2.45.0.linux-amd64.tar.gz

prometheus-2.45.0.linux-amd64.tar.gz      100%[=====]  86.00M   8.37MB/s   in
2023-07-01 15:58:07 (7.93 MB/s) - 'prometheus-2.45.0.linux-amd64.tar.gz' saved [9118594/9118594]

[root@controller Mon_binary]# tar -xvf prometheus-2.45.0.linux-amd64.tar.gz
[root@controller Mon_binary]# ls -l prometheus-2.45.0.linux-amd64
total 2277312
drwxr-xr-x 2 1801 123      38 Jun 23 23:49 console_libraries
drwxr-xr-x 2 1801 123     173 Jun 23 23:49 config
drwxr-xr-x 1 1281 123    1157 Jun 23 23:49 metrics
drwxr-xr-x 1 1281 123     373 Jun 23 23:49 NOTICE
drwxr-xr-x 1 1801 123 119846216 Jun 23 23:12 prometheus
drwxr-xr-x 1 1281 123     234 Jun 23 23:42 prometheus.yml
drwxr-xr-x 1 1801 123 112826558 Jun 23 23:14 cronjob
[root@controller Mon_binary]#
```

**Figure 12.1: Prometheus- Install**

- b. Create a dedicated user for running prometheus using the following command:

```
sudo useradd --no-create-home --shell /bin/false prometheus
```

- c. Rename the **prometheus** extracted directory using the mv command.

```
sudo mv prometheus-2.45.0.linux-amd64 prometheus
```

- d. Assign ownership of the **prometheus** directory to the

**prometheus** user.

```
sudo chown -R prometheus:prometheus /Mon_binary/prometheus
```

- e. Open the Prometheus configuration file **prometheus.yml**, located in the extracted directory. Modify the configuration to include the following job definition for PostgreSQL and Save the changes to the configuration file:

```
# my global config
global:
  scrape_interval: 15s
  evaluation_interval: 15s
# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
      - targets:
          scrape_configs:
            - job_name: "prometheus"
              scrape_interval: 10s
            static_configs:
              - targets: ["192.168.187.133:9090"]
```

## 2. Start Prometheus:

- a. Open a terminal or command prompt and navigate to the directory where Prometheus is installed. Run the following command to start Prometheus:

```
sudo -u prometheus ./prometheus --config.file=prometheus.yml &
```

Referring to [Figure 12.2](#), this log output confirms that Prometheus has been successfully started with the provided configuration file, and it is now ready to collect and serve metrics:

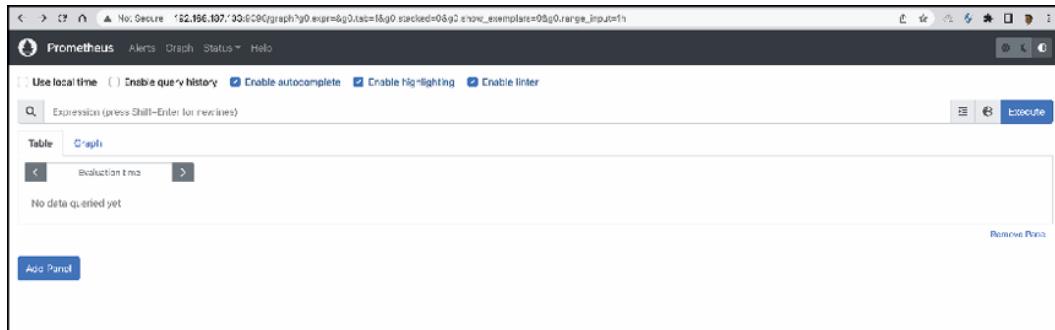
**Figure 12.2:** Prometheus- Start service

Prometheus will now start and begin collecting metrics from the PostgreSQL server.

### **3. Access Prometheus web interface:**

- a. Open a web browser and navigate to **http://localhost:9090** (or replace localhost with the appropriate hostname or IP address).
  - b. The Prometheus web interface will be accessible, allowing to explore and query collected metrics.

Upon successful configuration of Prometheus, you can now access the Prometheus web interface from your web browser using the address **<http://localhost:9090>**, as referenced in *Figure 12.3*. This access point becomes a pivotal gateway to monitor and manage your PostgreSQL environment effectively:



**Figure 12.3:** Prometheus- Web Interface

**Note: It is recommended to set up authentication and authorization for Prometheus when deploying it in production environments.**

#### **4. Install and configure postgres\_exporter:**

- a. Download the latest version of `postgres_exporter` from

prometheus official GitHub repositories

[https://github.com/prometheus-community/postgres\\_exporter/releases](https://github.com/prometheus-community/postgres_exporter/releases)

and extract the downloaded package. Find the desired release version (for example, **release/0.13.1**):

```
root@server:~# wget https://github.com/prometheus-community/postgres_exporter/releases/download/v0.13.1/postgres_exporter-v0.13.1.linux-amd64.tar.gz
2023-07-01 10:18:37  https://github.com/prometheus-community/postgres_exporter/releases/download/v0.13.1/postgres_exporter-v0.13.1.linux-amd64.tar.gz
Resolving github.com... (true.com)... 20.205.243.166
Connecting to github.com [true.com] (20.205.243.166):443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://objcts.githubusercontent.com/github-production-release-asset-2e65fe/41317079/0fa53915-e097-4f38-e78e-73a57309eb20?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=...
2023-07-01 10:18:37  https://objcts.githubusercontent.com/github-production-release-asset-2e65fe/41317079/0fa53915-e097-4f38-e78e-73a57309eb20?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=...
Resolving objcts.githubusercontent.com (objects.githubusercontent.com)... 185.109.120.133, 185.109.108.133, 185.109.111.133, ...
Connecting to objects.githubusercontent.com [objects.githubusercontent.com] (185.109.120.133):443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 8877943 (8.2M) [application/x-tar+gzip]
Saving to: “postgres_exporter-0.13.1.linux-amd64.tar.gz”

postgres_exporter-0.13.1.linux-amd64.tar.gz 100%[=] 8.18M 5.12MB/s   in
2023-07-01 10:18:40 (5.12 MB/s) - “postgres_exporter-0.13.1.linux-amd64.tar.gz” saved [8877943,8877943]
```

**Figure 12.4:** postgres\_exporter- Download and Install

- b. Create a dedicated user for running **postgres\_exporter** using the following command:

```
sudo useradd --no-create-home --shell /bin/false postgres_exporter
```

- c. Rename the **postgres\_exporter** extracted directory using the **mv** command.

```
sudo mv postgres_exporter-0.13.1.linux-amd64 postgres_exporter
```

- d. Assign ownership of the **postgres\_exporter** directory to the **postgres\_exporter** user.

```
sudo chown -R postgres_exporter:postgres_exporter
/Mon_binary/postgres_exporter
```

- e. Create a new file named **postgres\_exporter.env**. In the **postgres\_exporter.env** file, define the necessary environment variables. These variables provide configuration values for the **postgres\_exporter** process.

Here is an example of the content of the **postgres\_exporter.env** file:

```
DATA_SOURCE_NAME="postgresql://postgres:postgres@192.168.187.134:
5432/postgres?sslmode=disable"
```

Customize the environment variable values based on your PostgreSQL setup and monitoring requirements. The environment variable **DATA\_SOURCE\_NAME** (**DSN**) in the provided example

is used to define the PostgreSQL connection string DSN for the **postgres\_exporter** tool.

<b>Environment variable</b>	<b>Descriptions</b>
<b>DATA_SOURCE_NAME</b>	Name of the environment variable that will hold the PostgreSQL connection string.
postgresql://postgres:postgres@192.168.187.134:5432/postgres?sslmode=disable	PostgreSQL connection string that specifies the necessary information to establish a connection to the PostgreSQL database.

**Table 12.1:** *postgres\_exporter- Data source environment configuration*

## 5. Create the **systemd** service file and start **postgres\_exporter**:

- Open a terminal or text editor and create the **systemd** service file for **postgres\_exporter**.

```
sudo vi /etc/systemd/system/postgres_exporter.service
```

- Save the file by adding the following content to the **postgres\_exporter.service** file:

```
[Unit]
Description=Prometheus PostgreSQL Exporter
After=network-online.target
Wants=network-online.target

[Service]
User=postgres_exporter
Group=postgres_exporter
WorkingDirectory=/Mon_binary/postgres_exporter
EnvironmentFile=/tmp/postgres_exporter.env
ExecStart=/Mon_binary/postgres_exporter/postgres_exporter --
web.listen-address=192.168.187.133:9187 --web.telemetry-path=/metrics
--log.level=debug

Restart=always

[Install]
WantedBy=multi-user.target
```

- c. Reload the **systemd** daemon to pick up the new service configuration:

```
sudo systemctl daemon-reload
```

- d. Enable the service to start automatically on boot and start the **postgres\_exporter** service:

*# Enable the service to start automatically on boot:*

```
sudo systemctl enable postgres_exporter.service
```

*# Start the postgres\_exporter service:*

```
sudo systemctl start postgres_exporter.service
```

- e. Verify that the service is running without errors:

*# verify the postgres\_exporter service:*

```
sudo systemctl status postgres_exporter.service
```

Referring to *Figure 12.5*, the status of the service indicates that it is active and running. With these steps, we have created a **systemd** service file to manage the start and stop of the **postgres\_exporter** service. The service will automatically start on system boot and can be managed using the **systemctl** command.

```
[root@monserver postgres_exporter]# systemctl status postgres_exporter.service
● postgres_exporter.service - PostgreSQL PostgreSQL Exporter
   Loaded: loaded (/etc/systemd/system/postgres_exporter.service; enabled; vendor preset: disabled)
   Active: active (running) since Sun 2023-07-02 11:31:03 -0800; 24 ago
     Main PID: 3044 (postgres_export)
        Tasks: 4 (limit: 37316)
       Memory: 2.6M
          CPU: 11ms
         CGroup: /system.slice/postgres_exporter.service
                 └─ 3044 /usr/bin/postgres_exporter --web.listen_address=192.168.107.133:5187 --web.retry_path=/metrics --log.level=debug

Jul 02 11:31:20 monserver postgres_exporter[3044]: ts=2023-07-02T03:31:20.886Z caller=namespace.go:193 level=debug msg="Querying namespace" namespace=pg_stat_database_conflict
Jul 02 11:31:20 monserver postgres_exporter[3044]: ts=2023-07-02T03:31:20.897Z caller=namespace.go:193 level=debug msg="Querying namespace" namespace=pg_locks
Jul 02 11:31:20 monserver postgres_exporter[3044]: ts=2023-07-02T03:31:20.900Z caller=namespace.go:193 level=debug msg="Querying namespace" namespace=pg_stat_replication
Jul 02 11:31:20 monserver postgres_exporter[3044]: ts=2023-07-02T03:31:20.902Z caller=namespace.go:193 level=debug msg="Querying namespace" namespace=pg_replication_slots
Jul 02 11:31:20 monserver postgres_exporter[3044]: ts=2023-07-02T03:31:20.904Z caller=namespace.go:194 level=debug msg="Collector succeeded" name=stat_table_over_tables duration=1
Jul 02 11:31:20 monserver postgres_exporter[3044]: ts=2023-07-02T03:31:20.905Z caller=namespace.go:193 level=debug msg="Querying namespace" namespace=pg_stat_statistic
Jul 02 11:31:20 monserver postgres_exporter[3044]: ts=2023-07-02T03:31:20.906Z caller=namespace.go:194 level=debug msg="Collector succeeded" name=stat_bgwriter duration=1
Jul 02 11:31:20 monserver postgres_exporter[3044]: ts=2023-07-02T03:31:20.907Z caller=namespace.go:193 level=debug msg="Querying namespace" namespace=pg_stat_activity
Jul 02 11:31:20 monserver postgres_exporter[3044]: ts=2023-07-02T03:31:20.910Z caller=namespace.go:194 level=debug msg="Collector succeeded" namespace=user_tables duration=1
Jul 02 11:31:20 monserver postgres_exporter[3044]: ts=2023-07-02T03:31:20.922Z caller=namespace.go:194 level=debug msg="Collector succeeded" name=database duration=0.0
[root@monserver postgres_exporter]
```

*Figure 12.5: postgres\_exporter- Service status*

## 6. Access postgres\_exporter metrics:

- a. Gain valuable insights into PostgreSQL metrics with ease by accessing the **postgres\_exporter** metrics. To do so, open a web browser and navigate to <http://192.168.107.133:5187/metrics>, as referred to in *Figure 12.6*. Ensure to replace <replace\_with\_your\_IP> with the IP address where the PostgreSQL exporter is configured. This straightforward process provides a comprehensive view

of the performance and health of your PostgreSQL instance, enhancing your monitoring capabilities. Please refer to the following figure:

```

# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0.000122751
go_gc_duration_seconds{quantile="0.25"} 0.00016027664
go_gc_duration_seconds{quantile="0.5"} 0.000269459
go_gc_duration_seconds{quantile="0.75"} 0.000284734
go_gc_duration_seconds{quantile="1"} 0.000284734
go_gc_duration_seconds_sum 0.000893589
go_gc_duration_seconds_sum_count 4
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 20
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.20.5"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 4.364808e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total gauge
go_memstats_alloc_bytes_total 1.161592e+07
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 4583
# HELP go_memstats_frees Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 145934
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 1.161592e+06
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 4.364808e+06
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 1.75956e+06
# HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use.
# TYPE go_memstats_heap_inuse_bytes gauge
go_memstats_heap_inuse_bytes 6.086556e+06
# HELP go_memstats_heap_objects Number of allocated objects.
# TYPE go_memstats_heap_objects gauge

```

**Figure 12.6:** *postgres\_exporter*- Web interface

Referring to [Figure 12.6](#), the **postgres\_exporter** metrics endpoint is accessible, providing the collected metrics in a Prometheus-compatible format.

At this stage, we have successfully set up Prometheus and **postgres\_exporter**. Referring to [Figure 12.7](#), we can see that the endpoint from **postgres\_exporter** is not included in the target scrape pool of Prometheus:

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://192.168.187.133:9090/metrics	up	instance="192.168.187.133:9090" job="prometheus"	8.918s ago	4.676ms	

**Figure 12.7:** Prometheus- target detail

By including the **postgres\_exporter** information in the target scrape pool of Prometheus, we enable Prometheus to collect important metrics from the PostgreSQL database. These metrics provide insights into the performance and health of the database, allowing us to monitor its behaviour and make informed decisions based on the collected data.

## 7. Integrate **postgres\_exporter** with Prometheus:

- a. Open the Prometheus configuration file (**prometheus.yml**) using a text editor and inside the **scrape\_configs** section, add a new job configuration for **postgres\_exporter** as follows. This configuration specifies the target to scrape metrics from the **postgres\_exporter** endpoint.

```
- job_name: 'postgres_exporter-PG-Monitor'  
  static_configs:  
    - targets: ['192.168.187.133:9187']
```

- b. Restart the Prometheus service to apply the configuration changes.

```
# verify the postgres_exporter service:
```

```
sudo systemctl restart prometheus.service
```

Prometheus will now start scraping metrics from **postgres\_exporter** based on the configured job. You can access the Prometheus web interface by visiting **http://localhost:9090** in your web browser.

- c. Verify the **postgres\_exporter** endpoint from the target scrape pool of Prometheus.

As Prometheus begins scraping metrics from **postgres\_exporter** based on the configured job, it is crucial to verify the **postgres\_exporter** endpoint within the target scrape pool of Prometheus. This step ensures the seamless integration of metrics retrieval. To do so, refer to *Figure 12.8*, it is evident that the **postgres\_exporter** endpoint has been successfully added to the target scrape pool of Prometheus. By including the **postgres\_exporter** endpoint in the target scrape pool, Prometheus will periodically fetch metrics from the **postgres\_exporter** and store them for analysis. This enables monitoring and visualization of PostgreSQL-specific metrics, such as database connections, query execution times, and other relevant performance indicators.

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
<a href="http://192.168.187.133:9187/metrics">http://192.168.187.133:9187/metrics</a>	UP	instance="192.168.187.133:9187", job="postgres_exporter-PG-Monitor"	13.563s ago	68.036ms	

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
<a href="http://192.168.187.133:9080/metrics">http://192.168.187.133:9080/metrics</a>	UP	instance="192.168.187.133:9080", job="prometheus"	5.862s ago	4.142ms	

**Figure 12.8:** Prometheus- postgres\_exporter endpoint

- d. To verify that metrics from **postgres\_exporter** are being scraped, we can use Prometheus's expression browser. Enter the name of a metric exposed by **postgres\_exporter**, such as **pg\_stat\_database\_numbackends**, and execute the query.

Referring to [Figure 12.9](#), we have four databases: template1, template0, pgs and postgres. The metric query provides the number of active connections for each database. According to the output, the postgres database has four active connections, pgs has 1 active connection, and the rest of the database have 0 active connections. Please refer to the following figure:

pg_stat_database_numbackends{datid='0', datname='unknown', instance='192.168.187.133:9187', job='postgres_exporter-PG-Monitor'}	0
pg_stat_database_numbackends{datid='1', datname='template1', instance='192.168.187.133:9187', job='postgres_exporter-PG-Monitor'}	0
pg_stat_database_numbackends{datid='2', datname='template0', instance='192.168.187.133:9187', job='postgres_exporter-PG-Monitor'}	0
pg_stat_database_numbackends{datid='4', datname='pgs', instance='192.168.187.133:9187', job='postgres_exporter-PG-Monitor'}	1
pg_stat_database_numbackends{datid='5', datname='postgres', instance='192.168.187.133:9187', job='postgres_exporter-PG-Monitor'}	4

**Figure 12.9:** Prometheus- Verify metrics

The next step involves setting up Grafana, which is a powerful tool that helps create customized visualizations to better understand database performance, resource usage, query execution, and other important metrics.

## 8. Install and configure Grafana:

- a. Download and install the latest version of Grafana from official

repositories

[https://dl.grafana.com/enterprise/release/grafana-enterprise-10.0.1-1.x86\\_64.rpm](https://dl.grafana.com/enterprise/release/grafana-enterprise-10.0.1-1.x86_64.rpm). Find the desired release version (for example, **release/10.0.1-1**):

```
sudo yum install -y https://dl.grafana.com/enterprise/release/grafana-enterprise-10.0.1-1.x86_64.rpm
```

```
[root@node1 prometheus]# sudo yum install -y https://dl.grafana.com/enterprise/release/grafana-enterprise-10.0.1-1.x86_64.rpm
Updating Subscription Management repositories.
Unable to read consumer identity

This system is not registered with an entitlement server. You can use subscription-manager to register.

Last metadata expiration check: 1:42:48 ago on Sun 02 Jul 2023 01:11:12 PM +08.
grafana-enterprise-10.0.1-1.x86_64.rpm
Dependencies resolved.

-----  

Package           Architecture      Version       Repository  

=====  

Installing:  

grafana-enterprise          x86_64        10.0.1-1    @commandline  

-----  

Transaction Summary  

-----  

Install 1 Package  

-----  

total size: 44 M  

Installed size: 305 M  

Downloading Packages:  

Running transaction check  

Transaction check succeeded.  

Running transaction test  

Transaction test succeeded.  

Running transaction  

Preparing :  

Installing : grafana-enterprise=10.0.1-1.x86_64  

Running scriptlet: grafana-enterprise=10.0.1-1.x86_64
```

**Figure 12.10:** Grafana- Download and install

## 9. Start Grafana:

- Open a terminal or command prompt and execute the following command to start Grafana:

```
# Start Grafana service
systemctl start grafana-server.service
# Verify Grafana service
systemctl status grafana-server.service
```

Now that Grafana is set up, you can access it through a web browser by typing in the address <http://192.168.187.133:3000>. This is the location where Grafana is installed on the server, and it will be running on port 3000. By entering this address in a web browser, we will be able to start using Grafana's features and functionalities.

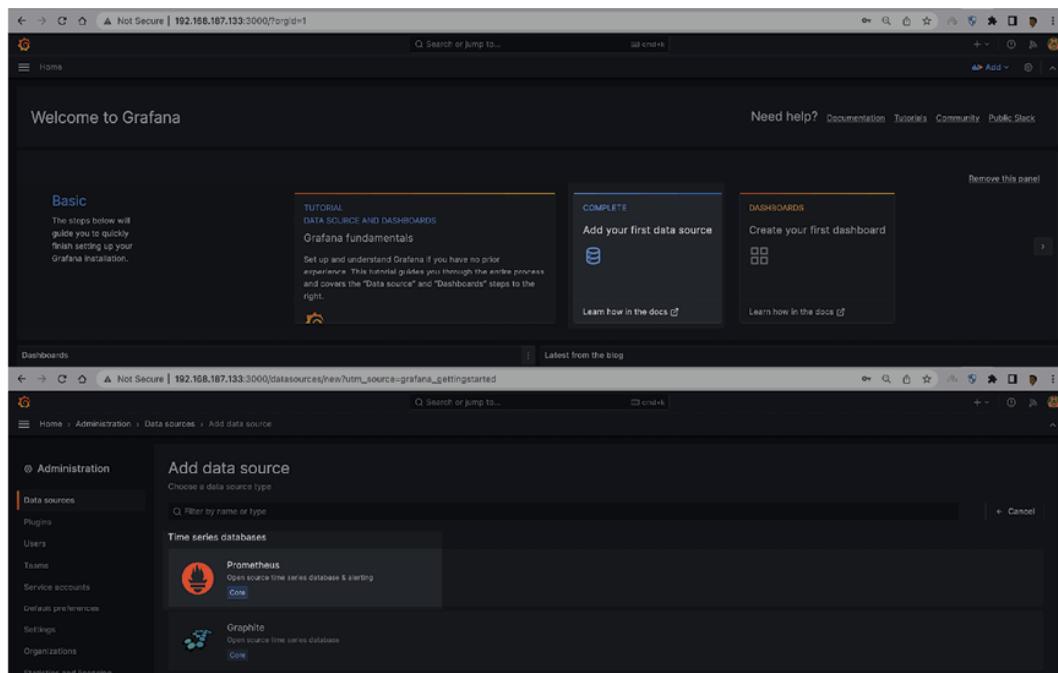
## 10. Configure Grafana dashboards:

- Open your web browser and navigate to <http://192.168.187.133:3000>. And log in to Grafana using the default credentials (**username: admin, password: admin**). Change the password for the admin user when prompted.

b. Click on the **Configuration** icon in the side menu and select **Data sources**.

c. Click on **Add data source** and select **Prometheus** from the **Grafana welcome screen**.

Referring to *Figure 12.11*, Enhance your Grafana configuration by seamlessly integrating Prometheus as a data source. click on **Add data source** and opt for **Prometheus** from the **Grafana welcome screen**:



*Figure 12.11: Grafana: Dashboard and data source*

d. Configure the data source with the following details and click **Save** and **Test**.

Name: Prometheus\_4\_PG

Prometheus server URL: <http://192.168.187.133:9090> #

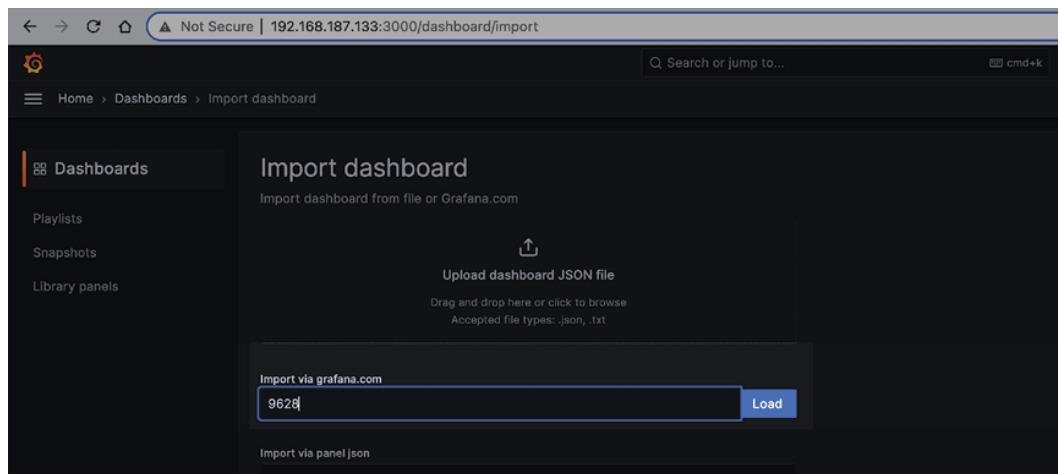
*Replace with the address and port where Prometheus is running*

e. Click on the **Dashboards** icon in the side menu.

f. Click on **Import** from the dropdown list under **New** menu and enter the ID of a PostgreSQL monitoring dashboard available from the Grafana community or create your own. In the **Import Dashboard** page, enter the dashboard ID

**9628** in the **Import via grafana.com** field and click on **Load** to proceed.

Referring to *Figure 12.12*, it serves as your visual guide, illustrating the process of importing a dashboard. Unlocking a wealth of monitoring insights and visualization options tailored to your PostgreSQL environment:

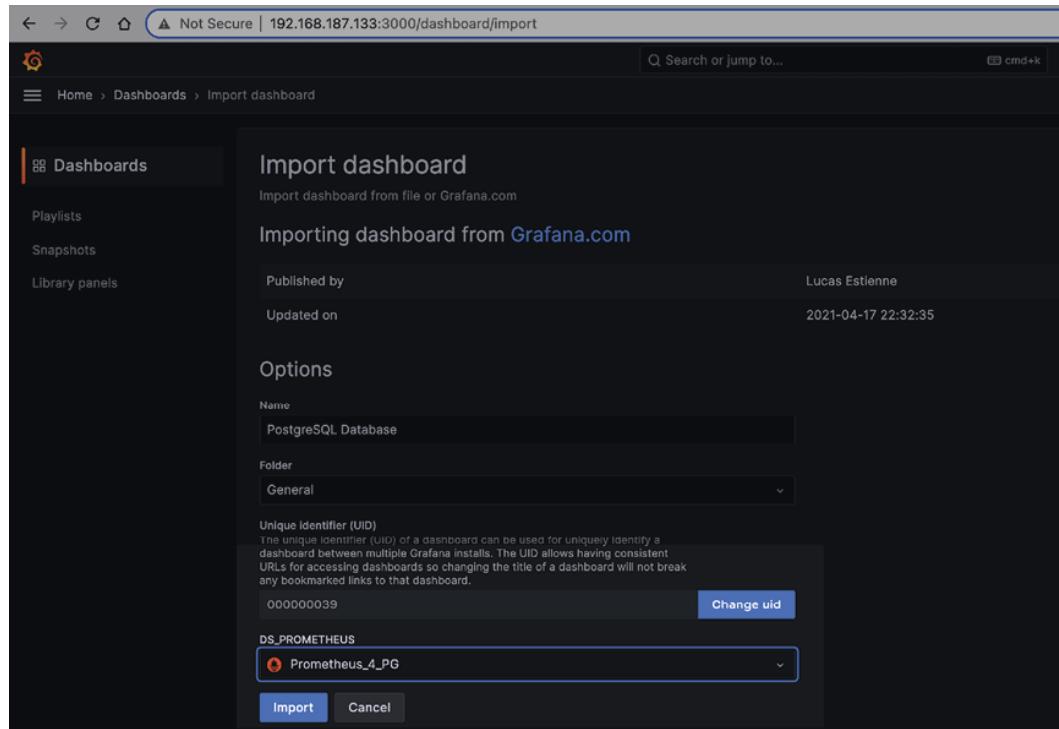


*Figure 12.12: Grafana- Import Dashboard*

**Note: Please note that for this recipe, we are using a preconfigured PostgreSQL dashboard (ID 9628) that is readily available in Grafana.**

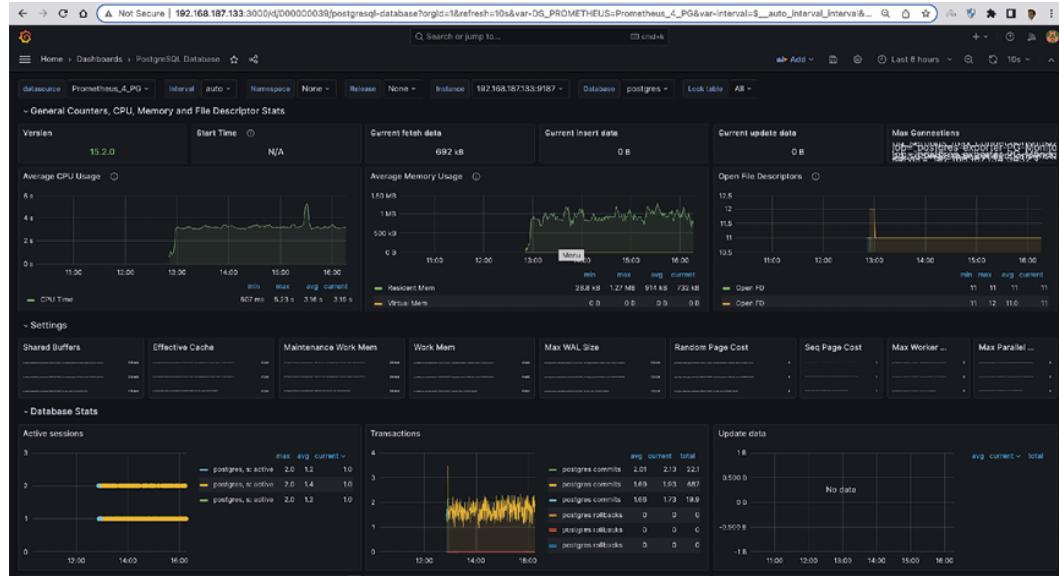
- g. From the import dashboard select **Prometheus\_4\_PG** in the **DS\_PROMETHEUS** field and click on **import** to proceed.

Referring to *Figure 12.13*, refine your Grafana dashboard by importing the designated **Prometheus\_4\_PG** configuration.



**Figure 12.13:** Grafana- Import Dashboard with Prometheus

Referring to the example shown in [Figure 12.14](#), we have successfully set up a monitoring dashboard specifically for PostgreSQL. With the combination of Prometheus, postgres\_exporter, and Grafana, you can now easily keep an eye on the performance and overall health of your PostgreSQL database. The Grafana dashboard provides an interactive and informative interface where you can visualize important metrics and analyze data to make informed decisions. Please refer to the following figure:



**Figure 12.14:** Grafana: PostgreSQL Monitoring Dashboard

## Recipe 98: Kill/terminate hung query

This recipe will guide you through the steps to kill or terminate a hung query in PostgreSQL. A hung query is a query that is running indefinitely or taking an unusually long time to execute. The forthcoming steps provide a comprehensive approach to identify and terminate such, ensuring your database operates seamlessly.

### 1. Connect to PostgreSQL:

- To begin, connect to PostgreSQL by opening the terminal or command prompt and utilizing the **psql** command-line tool:

```
psql -h <host> -p <port> -U <username> -W
```

- Execute the following query to identify the hung query:

```
SELECT pg_stat_activity.pid, pg_stat_activity.query_start,
pg_stat_activity.query FROM pg_stat_activity WHERE pg_stat_activity.state =
'active';
```

This query will display the **process ID (pid)**, the start time of the query, and the query itself for all active queries.

Referring to [Figure 12.15](#), we can see that three queries (with PIDs 60248, 61042 and 61157) were started on July 1, 2023. These queries have been active for a significant amount of time but have not yet been completed. This situation suggests that these queries might be stuck or not progressing as expected, which we refer to as hung queries. Please note that

the example mentioned in this recipe is based on an assumption and may not reflect real-world scenarios. Please refer to the following figure:

```
postgres# SELECT pg_stat_activity.pid, pg_stat_activity.query_start, pg_stat_activity.query FROM pg_stat_activity WHERE pg_stat_activity.state = 'active'
pid | query_start           | query
-----+---------------------+-----
60248 | 2023-07-03 13:12:05.49976+08 | SELECT transacts();
58312 | 2023-07-02 17:30:54.028555+08 | SELECT pg_stat_activity.pid, pg_stat_activity.query_start, pg_stat_activity.query FROM pg_stat_activity WHERE pg_stat_activity.state = 'a
;
61042 | 2023-07-01 21:19:02.248461+08 | select zfi_batch_cleanup();
61157 | 2023-07-01 21:19:21.464976+08 | select zfi_batch_update();
(4 rows)

pg_stat_activity:1 select now();
now
2023-07-02 17:20:47.513334+08
(1 row)

postgres#
```

**Figure 12.15: pg\_stat\_activity - List Hung Query**

## 2. Determine the query to terminate:

- Review the list of active queries obtained in *Step 1*.

pid	query_start	query
60248	2023-07-01 13:12:05.49976+08	SELECT transacts();
61042	2023-07-01 21:19:02.248461+08	select zfi_batch_cleanup();
61157	2023-07-01 21:19:21.464976+08	select zfi_batch_update();

## 3. Kill the Hung query:

- Execute the following command to terminate the identified hung query:

```
SELECT pg_cancel_backend(<PID>);
```

Replace **<pid>** with the **process ID (pid)** of the hung query obtained in *Step 2*. Referring to [Figure 12.16](#), This command sends a request to cancel the execution of the specified query, ensuring a prompt resolution to the hanging query issue:

```
postgres# SELECT pg_cancel_backend(60248);
pg_cancel_backend
t
(1 row)

postgres# SELECT pg_cancel_backend(61042);
pg_cancel_backend
t
(1 row)

postgres# SELECT pg_cancel_backend(61157);
pg_cancel_backend
t
(1 row)

postgres# SELECT pg_stat_activity.pid, pg_stat_activity.query_start, pg_stat_activity.query FROM pg_stat_activity WHERE pg_stat_activity.state = 'active'
pid      query_start           | query
-----+-----+-----
58312   2023-07-02 17:41:05.828638+08 | SELECT pg_stat_activity.pid, pg_stat_activity.query_start, pg_stat_activity.query FROM pg_stat_activity WHERE pg_stat_activity.state = 'active'
;
(1 row)

postgres#
```

**Figure 12.16:** pg\_stat\_activity – Terminate Hung query

If the **pg\_cancel\_backend** command does not terminate the query, we can escalate to a more forceful termination by executing the following command:

```
SELECT pg_terminate_backend(<PID>);
```

Again, replace **<pid>** with the process ID (pid) of the hung query. This command forcibly terminates the specified query.

#### 4. Verify query termination:

- a. Execute the query from *Step 1* again to check if the hung query is no longer present in the list of active queries.

## Explore statistics

Statistics provide valuable insights into the distribution and characteristics of data, enabling the query planner to make informed decisions on how to execute queries efficiently. PostgreSQL introduces significant improvements in statistics management, including enhancements to the auto vacuum process and improvements to the statistics collector.

## Recipe 99: Working with statistic collector and analyzer

The statistic collector collects various statistics about tables, indexes, and database activity, providing valuable insights into data distribution and access patterns. This information is then used by the statistic analyzer, which analyzes the collected statistics and updates them accordingly. By enabling these components and regularly analyzing the statistics, database administrators can ensure that the query planner makes accurate decisions based on the actual data distribution, resulting in more efficient query execution and improved overall performance of the PostgreSQL database.

In this recipe, we will explore how to work with the statistic collector and analyzer in PostgreSQL. We will cover the setup and configuration of these components, as well as a scenario that demonstrates their usage.

#### 1. Enabling the statistic collector:

- a. Open the PostgreSQL configuration file and locate the line that starts with **#track\_counts = off** and remove the leading **#** to uncomment the line.

Refer to [Figure 12.17](#) for visual guidance on enabling the statistic collector. The initial step, outlined in *Step 1*, directs you to the PostgreSQL configuration file, where you will locate and modify the line starting with **#track\_counts = off**. By removing the leading **#**, you uncomment the line, effectively enabling the statistic collector. Please refer to the following figure:

```
postgres@pgsrvdev ~]$ cat /var/lib/pgsql/15/data/postgresql.conf |grep -i track_counts
track_counts = on
postgres@pgsrvdev ~]$
postgres@pgsrvdev ~]$
```

**Figure 12.17:** statistic collector - Enable

- b. Optionally, adjust other relevant settings like **autovacuum**, **track\_activities**, and so on, based on your requirements and save the configuration file and restart the PostgreSQL service.

## 2. Querying statistics:

- a. Connect to the PostgreSQL database using a preferred client (for example, **psql**, **pgAdmin**). Execute the following SQL query to gather statistics about a specific table:

```
SELECT * FROM pg_stat_all_tables WHERE relname = 'testings';
```

Referring to [Figure 12.18](#), The executed query retrieves the statistics for a table named **testings** from the **pg\_stat\_all\_tables** view in the PostgreSQL database. You can observe various statistics on the table such as the number of sequential scans, the number of tuples read, the number of rows inserted, updated, and deleted, as well as information about the last vacuum and analyze operations performed. The **last\_autoanalyze** field in the output indicates the timestamp of the last **autoanalyze** operation, which occurred on July 2, 2023, at 16:45:46 UTC+8.

```

postgres=# 
SELECT * FROM pg_stat_all_tables WHERE relname = 'testings';
-[ RECORD 1 ]-----+
relid          | 21606
schemaname     | public
relname        | testings
seq_scan       | 6
seq_tup_read   | 114606000
idx_scan       |
idx_tup_fetch  |
n_tup_ins      | 2000000
n_tup_upd      | 0
n_tup_del      | 0
n_tup_hot_upd  | 0
n_live_tup    | 21101168
n_dead_tup    | 0
n_mod_since_analyze | 0
n_ins_since_vacuum | 2000000
last_vacuum    |
last_autovacuum|
last_analyze   |
last_autoanalyze | 2023-07-02 16:45:46.040626+08
vacuum_count   | 0
autovacuum_count | 0
analyze_count  | 0
autoanalyze_count | 1

postgres=#

```

**Figure 12.18:** statistic collector - Verify table statistics

### 3. Analyzing statistics:

- Connect to the PostgreSQL database. Execute the following SQL query to analyze the collected statistics:

```
ANALYZE <Table_Name>;
```

Referring to [Figure 12.19](#), outlines a fundamental step involving the execution of an SQL query to trigger the statistic analyzer for a specified table. This command will trigger the statistic analyzer to update the table's statistics based on the current data distribution:

```

postgres=# ANALYZE testings;
ANALYZE
postgres=# SELECT * FROM pg_stat_all_tables WHERE relname = 'testings';
-[ RECORD 1 ]-----+
relid          | 21606
schemaname     | public
relname        | testings
seq_scan        | 6
seq_tup_read   | 114606000
idx_scan        |
idx_tup_fetch  |
n_tup_ins      | 2000000
n_tup_upd      | 0
n_tup_del      | 0
n_tup_hot_upd  | 0
n_live_tup     | 21101168
n_dead_tup     | 0
n_mod_since_analyze | 0
n_ins_since_vacuum | 2000000
last_vacuum    |
last_autovacuum|
last_analyze   | 2023-07-02 18:25:25.309065+08
last_autoanalyze | 2023-07-02 16:45:46.040626+08
vacuum_count   | 0
autovacuum_count | 0
analyze_count  | 1
autoanalyze_count | 1

postgres=#

```

**Figure 12.19:** statistic collector - Verify table stats post analyze

**Note: We can also use the VACUUM ANALYZE <Table\_Name>; command to perform both vacuuming and analysis simultaneously.**

## Monitoring with system view

Monitoring with system views in PostgreSQL provides administrators with a powerful toolset to gain deep insights into the inner workings of their database. System views, such as **pg\_stat\_activity**, **pg\_stat\_bgwriter**, and **pg\_stat\_replication**, offer real-time information on various aspects of the database's performance, activity, and replication status. These views allow administrators to monitor active connections, identify long-running queries, track disk I/O, and analyze replication lag. By querying these system views, administrators can proactively detect and resolve issues, optimize resource utilization, and fine-tune database performance. With the robust monitoring capabilities offered by system views in PostgreSQL, administrators have

the necessary tools to ensure the smooth operation and optimal performance of their PostgreSQL databases.

## Recipe 100: Getting insight into query monitoring

By monitoring and analyzing the queries executed on your database, you can identify problematic queries, understand their execution patterns, and take proactive measures to optimize their performance. This recipe provides a step-by-step guide to enable query monitoring, collect query statistics, and analyze the data to gain valuable insights into the performance of your queries. By following these steps, you will be able to identify bottlenecks, optimize queries, and improve the overall efficiency and responsiveness of your PostgreSQL database. Let us get started with the recipe!

### 1. Enable query monitoring:

- Connect to your PostgreSQL database using a client tool or the command-line interface and create an extension **pg\_stat\_statements** to enable query monitoring.

```
# Connect to database
psql -h <host> -p <port> -U <username> -W
# Create pg_stat_statements extension
CREATE EXTENSION pg_stat_statements;
```

- Configure **shared\_preload\_library** for **pg\_stat\_statements**:

```
alter system set shared_preload_libraries='pg_stat_statements';
```

- Reload the configuration for the changes to take effect.

```
SELECT pg_reload_conf();
```

### 2. View query statistics:

- Connect to your PostgreSQL database. Run the following query to view the top 10 most frequently executed queries:

```
SELECT * FROM pg_stat_statements ORDER BY calls DESC LIMIT 10;
```

- Analyze the output to identify queries with a high number of calls.

### 3. Monitor query performance over time:

- Run the following query to view the query statistics aggregated over a period:

```
SELECT query, sum(calls) AS total_calls FROM pg_stat_statements
```

```
WHERE queryid IS NOT NULL GROUP BY query ORDER BY total_calls DESC;
```

- b. Use this information to identify queries that consistently have high call counts or total execution time, indicating areas that require further optimization.

That is it! We now have a recipe to get insight into query monitoring in PostgreSQL. By following these steps, you can identify problematic queries, and monitor query performance over time to ensure the smooth operation of the database.

## Recipe 101: Getting insight to monitor database lock

Monitoring database locks in PostgreSQL is essential for maintaining a high-performance and efficient database system. By monitoring database locks, administrators can gain valuable insights into the concurrency of transactions and identify any potential lock-related issues that may impact system performance. PostgreSQL provides built-in tools such as **pg\_stat\_activity** and **pg\_locks**, which allow administrators to view active sessions, identify lock types, and analyze lock contention.

This recipe provides step-by-step instructions on how to gain insights into database locks, identify potential issues, and optimize your system for better performance.

**Scenario:** An e-commerce website experiences occasional delays during the checkout process, leading to frustrated customers and potential revenue loss.

**Insight:** By monitoring database locks, administrators can identify if any long-held locks are causing the delays. Determine if a particular transaction is holding an exclusive lock on a critical table, preventing other transactions from proceeding. Let us delve into the recipe to get an insight on the database lock.

1. Identify active sessions with the long-held lock using **pg\_stat\_activity**:

```
SELECT pid,  
       datid,  
       username,  
       application_name,  
       wait_event_type,  
       wait_event,  
       pg_blocking_pids(pid) AS blocker,
```

```

backend_start,
backend_type,
query
FROM pg_stat_activity
WHERE wait_event IS NOT NULL
ORDER BY backend_start DESC;

```

Referring to [Figure 12.20](#), The output shows information about the different tasks happening in a database. Focus on the blocker column. It shows the process ID(s) of any blocking backends that are causing the current backend to wait. In this case, the blocker array is populated with the process ID **508210** for the first two rows, indicating that the backends with process IDs **508499** and **508335** are being blocked by the backend with process ID **508210**.

	pid	datid	username	application_name	wait_event_type	wait_event_text	blocker	backend_start	backend_type	query
1	508499	6	postgres	psql	Lock	relation	(508210)	2023-06-26 15:47:26.262782+03	client backend	select * from orders where id = '1101';
2	508335	6	postgres	psql	Lock	relation	(508210)	2023-06-26 15:46:28.0100481+03	client backend	select * from customers;
3	508210	6	postgres	psql	Client	ClearRead	0	2023-06-26 15:46:48.351603+03	client backend	Select oid, id, name, application_name
4	301639	45187	postgres	pgAdmin 4 - 00-psqls	Client	ClearRead	0	2023-06-22 09:17:24.009244+03	client backend	Select roles_id, id, roles_name as
5	301639	5	postgres	pgAdmin 4 - 00-psqls	Client	ClearRead	0	2023-06-22 09:17:24.0500891+03	client backend	SELECT roles_id, id, roles_name as
6	226141	[n/a]	[n/a]		Activity	ArchiverMain	0	2023-06-20 15:22:03.510773+03	archiver	
7	226142	[n/a]	postgres		Activity	LogicalArchiverMain	0	2023-06-20 15:22:23.148652+03	logical replication launcher	
8	226140	[n/a]	[n/a]		Activity	AutoVacuumMain	0	2023-06-20 15:22:23.147564+03	autovacuum launcher	
9	226129	[n/a]	[n/a]		Activity	WALWriterMain	0	2023-06-20 15:22:23.147551+03	walwriter	
10	226137	[n/a]	[n/a]		Activity	BgWriterHibernate	0	2023-06-20 15:22:27.559591+03	background writer	
11	226128	[n/a]	[n/a]		Activity	UnReplicaMain	0	2023-06-20 15:22:27.559541+03	checkpointer	

[Figure 12.20: List Locked query](#)

2. Analyze specific lock details for the identified session using **pg\_locks**:

```

SELECT pid,
MODE,
locktype,
relation::regclass,
page,
tuple
FROM pg_locks
WHERE pid in ('508499',
'508335');

```

Referring to the preceding SQL execution output in the [Figure 12.21](#):

**virtualxid locks:** Both processes have an **ExclusiveLock** on a **virtualxid** locktype. This indicates that they are holding exclusive locks

related to transaction control and management.

**Relation locks:** Each process also has an **AccessShareLock** on a specific relation. The first process (**PID 508499**) has an AccessShareLock on the orders table, while the second process (**PID 508335**) has an AccessShareLock on the customers table. This lock type allows multiple processes to have shared read access to the specified table.

	pid integer	mode text	locktype text	relation regclass	page integer	tuple smallint
1	508499	ExclusiveLock	virtualxid	[null]	[null]	[null]
2	508335	ExclusiveLock	virtualxid	[null]	[null]	[null]
3	508499	AccessShareLock	relation	orders	[null]	[null]
4	508335	AccessShareLock	relation	customers	[null]	[null]

**Figure 12.21:** Verify Locked query detail

**Resolution:** After identifying the problematic locks and associated queries, administrators can optimize the batch job by optimizing the query execution plan, adjusting the batch job's configuration parameters, or utilizing parallel processing techniques.

## Recipe 102: Getting insight to monitor active session

In PostgreSQL, an active session refers to a connection established by a client application to the PostgreSQL database that is currently executing or waiting for an event. Each active session is represented by a backend process in the database system. Monitoring active sessions is crucial for database administrators to understand the workload, identify potential bottlenecks, and optimize performance.

This recipe is aimed at users who want to gain insights and monitor active sessions in PostgreSQL, administrators can take necessary actions to optimize query performance, allocate resources efficiently, and ensure the smooth operation of the PostgreSQL database.

**Scenario:** Identifying and analyzing active sessions.

**Insight:** Monitoring active sessions in PostgreSQL allows you to gain insights into the current activity and resource utilization within the database. The **pg\_stat\_activity** view provides detailed information about active sessions, including the session ID, username, query, start time, and more. Analyzing active sessions helps identify potential

issues such as long-running queries, excessive resource usage, or blocked processes. Let us begin the recipe to identify and analyze the active sessions by initiating the first step: Connecting to PostgreSQL.

## 1. Connect to PostgreSQL:

- Open a terminal or command prompt and connect to the PostgreSQL database using the **psql** command-line tool:

```
psql -h <host> -p <port> -U <username> -W
```

## 2. Query active sessions:

- Execute a query to retrieve information about the active sessions in the database. Use the **pg\_stat\_activity** system view to fetch details such as session ID, username, query being executed, start time, wait events, and state of the query.

```
SELECT pid,  
       datid,  
       username,  
       application_name,  
       backend_start,  
       backend_type,  
       state,  
       query  
FROM pg_stat_activity  
WHERE state = 'active';
```

Referring to the above SQL execution output in [Figure 12.21](#), the output shows four rows, each representing an active session with its corresponding information:

	pid	integer	datid	oid	username	name	application_name	text	backend_start	timestamp with time zone	backend_type	text	state	text	query	text
1	508499	5	postgres	pgsql					2023-06-26 15:47:26.292782+08	client backend	active			select * from orders where id = '1101';		
2	508335	5	postgres	pgsql					2023-06-26 15:46:28.910043+08	client backend	active			select * from customers;		
3	508297	5	postgres	pgsql	pgAdmin 4 - CONN:6171003				2023-06-26 15:45:46.996692+08	client backend	active			SELECT pid, datid, username, application_name, backend_start, backend_type, state, query FROM pg_stat_activity WHERE state = 'active';		
4	508210	5	postgres	pgsql					2023-06-26 15:43:33.023759+08	client backend	active			do \$\$ begin for r in 1..8000000 loop insert into customers(id) values(r); end loop; end\$\$;		

*Figure 12.22: List active sessions*

## 3. Take appropriate actions based on the insights gained:

- Optimize long-running queries by examining their execution plans, identifying missing indexes, or rewriting the queries.
- Address high resource utilization by optimizing queries, tuning database configuration parameters, or upgrading hardware

resources.

- c. Resolve wait events by investigating and addressing underlying issues causing the wait conditions.

## Recipe 103: Working with query plan

In this recipe, we will explore the concepts of query plans and query string parsing in PostgreSQL. Understanding query plans is crucial for database performance optimization, as it allows us to analyze how the database executes queries. Additionally, parsing query strings helps us understand the structure and components of a query. By following this recipe, you will learn how to obtain query plans and parse query strings effectively.

Let us consider a scenario where you are a database administrator responsible for optimizing the performance of a PostgreSQL database. You have received a complaint from a user about slow query execution. To identify the issue, you decide to analyze the query plan and parse the query string. Follow the steps outlined in this recipe to navigate through the query plan, empowering you to analyze and enhance the performance of your PostgreSQL database.

### 1. Analyzing the query plan:

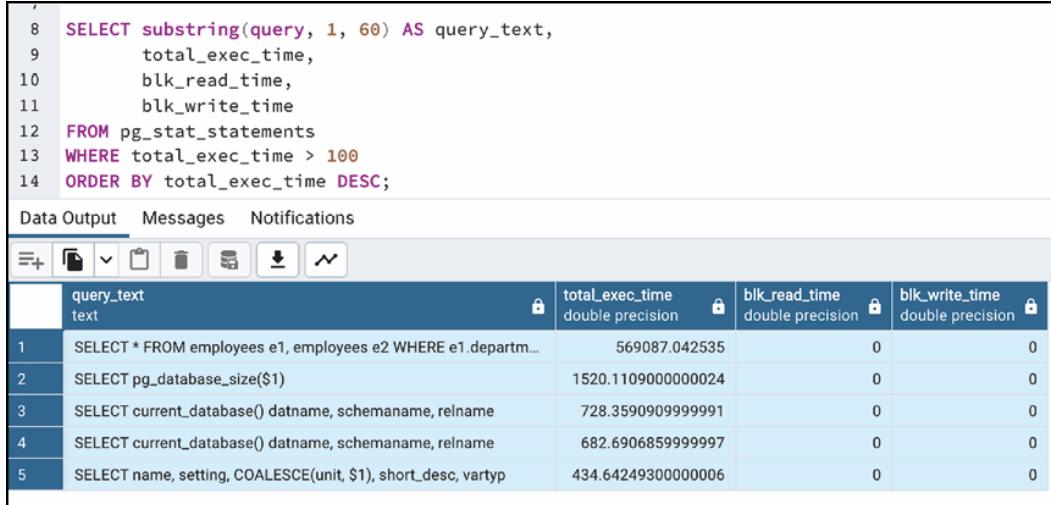
- a. Open a terminal or command prompt and connect to the PostgreSQL database using the **psql** command-line tool and identify the slow query that needs analysis by executing the following query:

```
SELECT substring(query, 1, 60) AS query_text,  
       total_exec_time,  
       blk_read_time,  
       blk_write_time  
  FROM pg_stat_statements  
 WHERE total_exec_time > 100  
 ORDER BY total_exec_time DESC;
```

Referring to [Figure 12.23](#), executing the above query will retrieve the slow queries based on the specified threshold. The query applies a filter condition **total\_exec\_time > 100** to only include queries with a total execution time greater than 100 milliseconds.

Query (**SELECT \* FROM employees e1, employees e2 WHERE e1.department ...**) has a **total\_exec\_time** of

**569087.042535** milliseconds, indicating that it is the slowest query among the captured queries. Please refer to the following figure:



The screenshot shows a PostgreSQL pg\_stat\_statements interface. At the top, there is a SQL query block:

```

8  SELECT substring(query, 1, 60) AS query_text,
9    total_exec_time,
10   blk_read_time,
11   blk_write_time
12  FROM pg_stat_statements
13 WHERE total_exec_time > 100
14 ORDER BY total_exec_time DESC;

```

Below the query block is a table titled "Data Output". The table has columns: query\_text, total\_exec\_time, blk\_read\_time, and blk\_write\_time. The table contains five rows of data:

	query_text	total_exec_time	blk_read_time	blk_write_time
1	SELECT * FROM employees e1, employees e2 WHERE e1.department...	569087.042535	0	0
2	SELECT pg_database_size(\$1)	1520.1109000000024	0	0
3	SELECT current_database() datname, schemaname, relname	728.3590909999991	0	0
4	SELECT current_database() datname, schemaname, relname	682.6906859999997	0	0
5	SELECT name, setting, COALESCE(unit, \$1), short_desc, vartyp	434.64249300000006	0	0

**Figure 12.23:** pg\_stat\_statements - Verify query

- b. Enable the **EXPLAIN** command for the slow query to obtain its query plan.

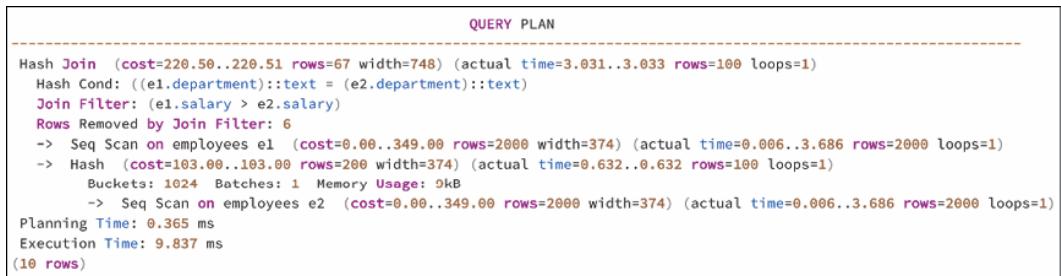
```

EXPLAIN ANALYZE SELECT * FROM employees e1, employees e2
WHERE e1.department = e2.department AND e1.salary > e2.salary;

```

- c. Study the output of the **EXPLAIN ANALYZE** command, which provides information about the query plan, such as the order of operations, the number of rows processed, and execution time.

Referring to [Figure 12.24](#), guides you through the process of enabling the **EXPLAIN** command for a specific slow query and subsequently studying its output:



The screenshot shows the output of the EXPLAIN ANALYZE command. The output is titled "QUERY PLAN" and includes the following details:

```

Hash Join  (cost=220.50..220.51 rows=67 width=748) (actual time=3.031..3.033 rows=100 loops=1)
  Hash Cond: ((e1.department)::text = (e2.department)::text)
  Join Filter: (e1.salary > e2.salary)
  Rows Removed by Join Filter: 6
    -> Seq Scan on employees e1  (cost=0.00..349.00 rows=2000 width=374) (actual time=0.006..3.686 rows=2000 loops=1)
        -> Hash  (cost=103.00..103.00 rows=200 width=374) (actual time=0.632..0.632 rows=100 loops=1)
          Buckets: 1024  Batches: 1  Memory Usage: 9kB
            -> Seq Scan on employees e2  (cost=0.00..349.00 rows=2000 width=374) (actual time=0.006..3.686 rows=2000 loops=1)
Planning Time: 0.365 ms
Execution Time: 9.837 ms
(10 rows)

```

**Figure 12.24:** Explain Analyze - Query plan

## Vacuum and bloat consideration

**Vacuuming** and **bloat considerations** are crucial aspects of database maintenance. Vacuuming is the process of reclaiming storage space by removing dead tuples, which are no longer needed for query execution. It prevents database bloat, where tables and indexes consume more space than necessary, leading to performance degradation. PostgreSQL provides an automatic vacuuming mechanism called autovacuum, which ensures timely clean-up and prevents excessive bloat. However, manual vacuuming may still be necessary in certain scenarios. Understanding and managing bloat is essential for optimizing database performance. Techniques such as reindexing, cluster operations, and table partitioning can help alleviate bloat and improve query response times. By addressing vacuuming and bloat considerations, PostgreSQL users can maintain a healthy database environment, ensuring optimal performance and storage efficiency.

## Recipe 104: Getting insight into vacuum and bloat

Vacuum and bloat management in PostgreSQL 15 has been significantly improved, offering better performance and efficiency. Vacuum is a critical process that removes dead tuples from the database, allowing the space to be reclaimed and reused. In PostgreSQL 15, the autovacuum process has been enhanced to optimize its operation and reduce the impact on system resources.

Additionally, the new version introduces improvements in bloat management, which refers to the excessive space occupied by tables or indexes due to inefficient data storage. PostgreSQL 15's enhanced bloat detection and mitigation mechanisms help prevent excessive storage consumption and maintain optimal database performance.

In this recipe, we will explore how to gain insights into vacuuming and managing bloat in PostgreSQL 15. We will cover the necessary steps to monitor and optimize the database using built-in commands and tools. This recipe aims to provide a comprehensive understanding of vacuuming and bloat management for PostgreSQL users.

**Scenario:** Let us imagine that a database administrator is responsible for maintaining a PostgreSQL database for a rapidly growing e-commerce platform. Database administrator have noticed some performance degradation lately, and suspects that vacuuming and bloat might be contributing factors. To proactively tackle these issues, this recipe guides you through gaining insights into the vacuuming process and effectively managing bloat. Now, let us delve into the

initial step of analyzing bloat, providing you with the tools to monitor and optimize your database effectively.

### 1. Analyzing bloat:

- a. To gain insights into bloat, we first need to analyze the tables and indexes in our database. Open the command-line interface and connect to the PostgreSQL server using the **psql** command. Once connected, run the following command to view the bloat information:

```
# Connect to the database
psql -U postgres -d postgres
      # Install pgstattuple extension
CREATE EXTENSION pgstattuple;
      # Identify dead tuples
select * from pgstattuple('testings');
```

Referring to [Figure 12.25](#), we can observe that the '**testings**' table has a significant amount of free space (55.5%) but also a relatively high percentage of dead tuples (20.32%). This indicates that there is potential bloat in the table, and performing a vacuum operation can help reclaim space and improve database performance:

```
postgres=# select * from pgstattuple('testings');
-[ RECORD 1 ]-----+-----
table_len          | 475193344
tuple_count        | 1897400
tuple_len          | 58597191
tuple_percent      | 12.33
dead_tuple_count   | 3021540
dead_tuple_len     | 96565463
dead_tuple_percent | 20.32
free_space         | 263725468
free_percent       | 55.5
```

```
postgres=#
```

**Figure 12.25:** Table dead tuples stats

Referring to [Figure 12.26](#), we can conclude that the **testings** table has a significant number of dead tuples (11,212,000) that are occupying space in the table. The presence of dead tuples can lead to bloat, which affects the performance and efficiency of the database.

```
postgres=# SELECT * FROM pg_stat_all_tables WHERE relname = 'testings';
-[ RECORD 1 ]-----+
relid          | 21606
schemaname     | public
relname        | testings
seq_scan       | 1601
seq_tup_read   | 15532183886
idx_scan
idx_tup_fetch
n_tup_ins      | 49081377
n_tup_upd      | 0
n_tup_del      | 11212000
n_tup_hot_upd  | 0
n_live_tup     | 1897400
n_dead_tup     | 11212000
n_mod_since_analyze | 44321400
n_ins_since_vacuum | 13109400
last_vacuum
last_autovacuum
last_analyze
last_autoanalyze
vacuum_count   | 0
autovacuum_count | 0
analyze_count  | 0
autoanalyze_count | 0

postgres=#

```

**Figure 12.26:** Table stats before Vacuuming

## 2. Vacuuming:

- After identifying the bloated tables, it is time to perform a vacuum operation. Vacuuming removes dead tuples and reclaims space. To perform a manual vacuum on a specific table, use the following command:

```
VACUUM testings;
```

Referring to [Figure 12.27](#), after performing a vacuum operation on the **testings** table, the following changes can be observed based on the provided query results:

**n\_dead\_tup (0)**: The number of dead tuples in the table is now 0. This means that the vacuum operation has successfully removed all the dead tuples, reclaiming the space they were occupying.

**n\_ins\_since\_vacuum (0)**: The count of tuples inserted since the last vacuum operation is now 0. This indicates that no new tuples have been added to the table after the vacuum operation.

**last\_vacuum (2023-07-02 20:45:06.64065+08)**: The timestamp of the last vacuum operation is now recorded as 2023-07-02 20:45:06.64065+08, indicating when the vacuum operation was last performed.

**n\_dead\_tup, dead\_tuple\_len, and dead\_tuple\_percent**: These values are all 0, confirming that there are no dead tuples remaining in the table after the vacuum operation.

**table\_len (475193344)**: The total size of the table remains the same at 475,193,344 bytes.

**free\_space (405063748)**: The amount of free space in the table has increased to 405,063,748 bytes (approximately 405 MB). This indicates that the vacuum operation has successfully freed up space by removing the dead tuples.

**free\_percent (85.24)**: The percentage of free space in relation to the total table size is now 85.24%. This indicates that a significant portion of the table is now available for future data insertion.

```
postgres=# SELECT * FROM pg_stat_all_tables WHERE relname = 'testings';
-[ RECORD 1 ]-----
relid          | 21606
schemaname     | public
relname        | testings
seq_scan       | 1603
seq_tup_read   | 15535978686
idx_scan       |
idx_tup_fetch  |
n_tup_ins     | 49081377
n_tup_upd     | 0
n_tup_del     | 11212000
n_tup_hot_upd | 0
n_live_tup    | 1897400
n_dead_tup    | 0
n_mod_since_analyze | 44321400
n_ins_since_vacuum | 0
last_vacuum   | 2023-07-02 20:45:06.64065+08
last_autovacuum| 
last_analyze   |
last_autoanalyze| 
vacuum_count   | 3
autovacuum_count | 0
analyze_count  | 0
autoanalyze_count | 0

postgres=# select * from pgstattuple('testings');
      table_len      |      tuple_count      |      tuple_len      |      tuple_percent      |      dead_tuple_count      |      dead_tuple_len      |      dead_tuple_percent      |      free_space      |      free_percent
-----+-----+-----+-----+-----+-----+-----+-----+-----+
475193344 | 1897400 | 58597191 | 12.33 | 0 | 0 | 0 | 405063748 | 85.24
(1 row)
```

**Figure 12.27: Table stats post vacuuming**

To perform a vacuum on the entire database, use the following command:

```
VACUUM full;
```

Alternatively, we can enable the autovacuum feature to automate the process. Run the following command to enable autovacuum:

```
ALTER TABLE testings SET (autovacuum_enabled = true);
```

### 3. Monitoring autovacuum:

- To monitor the **autovacuum** process, PostgreSQL provides several built-in views. Run the following command to view the **autovacuum** information:

```
SELECT * FROM pg_stat_progress_vacuum;
```

This command displays information about the current autovacuum operations, such as table name, current phase, and progress.

### 4. Adjusting autovacuum settings:

- To optimize **autovacuum** performance, we may need to adjust the configuration parameters. Open the PostgreSQL configuration file (**postgresql.conf**) and modify the following parameters according to your requirements:

```
autovacuum_vacuum_scale_factor = 0.1
```

```
autovacuum_analyze_scale_factor = 0.05
```

```
autovacuum_vacuum_threshold = 50
```

```
autovacuum_analyze_threshold = 50
```

These parameters control when **autovacuum** processes are triggered and how much work they perform. Adjusting these values can help optimize the vacuuming process.

By following these steps, one can gain insight into vacuuming and bloat in PostgreSQL. Analyzing bloat, performing vacuums, monitoring autovacuum progress, and adjusting settings will help you maintain a well-performing and efficient database system. Regular maintenance is essential for preventing excessive bloat and optimizing database performance.

## Explore fsync

In PostgreSQL, **fsync** is a crucial configuration parameter that determines whether the system enforces immediate synchronization to disk of all changes made to the database. When **fsync** is enabled (which is the default setting), PostgreSQL ensures that committed

transactions are durably written to disk, providing a high level of data integrity and compliance with ACID properties. This feature is essential for maintaining the reliability of the database, especially in production environments where data loss is unacceptable.

Here is a table summarizing the advantages and disadvantages of enabling **fsync** in PostgreSQL, along with situations where it should be enabled and when it can be considered for disabling:

Aspect	Enabling fsync	Disabling fsync
Advantage	<p>Durability: Ensures that committed transactions are safely written to disk, preventing data loss in case of a crash.</p> <p>Reliability: Guarantees the integrity of the database by maintaining a consistent state even after unexpected shutdowns.</p> <p>Atomicity: Helps maintain the atomicity of transactions by ensuring that either all or none of the changes are applied.</p> <p>Compliance: Necessary for compliance with ACID properties of database transactions.</p>	<p>Performance: Disabling <b>fsync</b> can lead to improved write performance, as it allows PostgreSQL to buffer changes in memory without immediately synchronizing with disk.</p> <p>Reduced I/O: Without <b>fsync</b>, there is less disk I/O, which can be beneficial for systems where write performance is crucial and data loss risk is acceptable.</p> <p>Testing/development: In non-production environments, disabling <b>fsync</b> might be acceptable to achieve better performance for testing or development purposes.</p>

Aspect	Enabling <b>fsync</b>	Disabling <b>fsync</b>
Disadvantage	<p>Data loss risk: Enabling <b>fsync</b> reduces the risk of data loss in the event of a system crash or power failure, as committed transactions are immediately written to disk. However, this immediate synchronization can impact write performance.</p> <p>ACID compliance: Enabling <b>fsync</b> is essential for maintaining ACID compliance by ensuring that committed transactions are durably stored on disk. This guarantees the reliability and consistency of the database, but it comes at the cost of potential performance implications.</p> <p>Safety: While enabling <b>fsync</b> enhances the safety and durability of the database, it may introduce performance overhead, and the system needs to carefully balance the trade-off between safety and performance.</p>	<p>Data loss risk: Disabling <b>fsync</b> increases the risk of data loss in the event of a system crash or power failure, as committed transactions may not be written to disk immediately.</p> <p>ACID compliance: Disabling <b>fsync</b> compromises ACID compliance, which is essential for maintaining the reliability and consistency of the database.</p> <p>Safety: May lead to data corruption or inconsistent states if the system crashes before changes are flushed to disk.</p>
When to use	<p>Production environments: Enable <b>fsync</b> in production environments to ensure data durability and maintain the integrity of the database.</p> <p>Critical systems: For databases handling critical data where data loss is unacceptable, <b>fsync</b> should be enabled.</p> <p>Compliance requirements: Enable <b>fsync</b> when compliance with ACID properties is mandatory.</p>	<p>Non-critical environments: In non-critical environments like testing or development, where performance is a higher priority than data durability.</p> <p>Performance testing: For performance testing scenarios where data loss is acceptable for short-term testing purposes.</p> <p>Caching scenarios: In scenarios where data can be regenerated or is not critical, and faster write operations are more important.</p>

**Table 12.2:** PostgreSQL- Summarized **fsync**

## Recipe 105: Getting insight into **fsync**

Understanding and fine-tuning the file synchronization process, commonly known as **fsync**, is crucial for ensuring data integrity and durability. This recipe guides you through the steps to gain insights into **fsync**, empowering you to optimize its configuration for your PostgreSQL database. Let us begin the recipe, guiding you through the steps to glean insights into **fsync**, enabling you to fine-tune its configuration for your PostgreSQL database.

## 1. Edit the PostgreSQL configuration file:

- Determine the location of the PostgreSQL configuration file (**postgresql.conf**) by executing the following query within a PostgreSQL session:

```
# Locate the configuration file  
SHOW config_file;
```

The output will provide the full path to the **postgresql.conf** file.

**Note: This above step is necessary if you plan to update the configuration directly in the postgresql.conf file.**

**If you intend to update the configuration using the ALTER SYSTEM statement instead of directly editing the postgresql.conf file, follow the alternative steps provided in the subsequent instructions.**

- Alter the **fsync** configuration to ensure it is set to **on** for activating the PostgreSQL **fsync**. However, before making any changes to the configuration, retrieve and display the current value of the **fsync** parameter.

```
# Display previous fsync configuration
```

```
SHOW log_rotation_age;
```

*# Execution output of above command:*

```
fsync
```

```
-----
```

```
on
```

```
(1 row)
```

```
# Enable fsync
```

```
ALTER SYSTEM SET fsync = 'on';
```

Understanding **fsync** values. Gain insight into the various **fsync** settings with the following tables:

Value	Description
Off	Disables <b>fsync</b> . Use with caution, suitable for non-production environments, but poses a risk of data loss during system crashes.
On	Enables <b>fsync</b> , the recommended setting for most production environments, ensuring data durability.

**Table 12.3:** PostgreSQL-fsync configuration value options

Understanding **fsync** values and their dependency on **wal\_sync\_method**. Delve into the two available settings, **On** and **Off**, while recognizing the critical relationship with the **wal\_sync\_method** parameter through the following steps:

- a. The **fsync** parameter is associated with the **wal\_sync\_method** parameter, both of which play a crucial role in determining how committed transactions are synchronized to disk.
- b. The **wal\_sync\_method** parameter, which specifies the method used for synchronizing the **write-ahead logging (WAL)** to disk, is closely related. There are different options for **wal\_sync\_method**, including **fsync**, **open\_datasync**, **open\_sync**, and **fdatasync**. Each option affects how PostgreSQL ensures the durability of the write-ahead log, which is critical for maintaining transaction consistency.

Here is a more detailed breakdown:

Scenario	fsync	wal_sync_method
Critical systems: Data loss is unacceptable	Enable	Typically set to <b>fsync</b>
Compliance requirements: ACID compliance is crucial	Enable	Typically set to <b>fsync</b>
Non-critical environments: Performance is a priority	Enable for data integrity, may experiment for performance improvements	Options may include <b>fdatasync</b> , <b>open_datasync</b> , <b>fsync</b> , <b>fsync_writethrough</b> , <b>open_sync</b>
Performance testing: Temporary acceptance of data loss for testing purposes	Disable for performance	Irrelevant; <b>wal_sync_method</b> is inconsequential without <b>fsync</b>

**Table 12.4:** PostgreSQL-fsync dependency with wal\_sync\_method

- c. Begin on the experimentation phase by following these steps to modify **fsync** Settings based on your specific requirements. Afterward, initiate a reload of PostgreSQL config to ensure the changes take effect using the following steps:

```
# Enable fsync
```

```
ALTER SYSTEM SET fsync = 'on';
```

```
# Enable wal_sync_method setting to fsync
```

```
ALTER SYSTEM SET wal_sync_method = 'fsync';
```

```
# Save the changes and apply
```

```
SELECT pg_reload_conf();
```

**Note:** By default, **fsync** is typically set to 'on' in PostgreSQL. Therefore, altering it to 'on' in your statement might not necessarily change it from its default setting, but it ensures that it remains explicitly set to 'on'.

Let us initiate a practical test scenario to demonstrate the impact of different **fsync** settings on PostgreSQL performance. Using the provided insert query, we will assess the results for various **fsync** configurations, specifically focusing on **fdatasync**, **open\_datasync**, **fsync**, **fsync\_writethrough**, and **open\_sync**.

```
# Example scenario insert query
```

```
DO $$
```

```
BEGIN
```

```
    FOR r IN 1..1000 LOOP
```

```
        INSERT INTO pgbook (id) VALUES (r);
```

```
    END LOOP;
```

```
END;
```

```
$$;
```

### Test scenario and results:

- **Standard fsync() (Default Setting):**
  - **Result:** Roughly 6000 inserts per second.
- **fsync Disabled:**
  - **Result:** Approximately 12000-13000 inserts per second.
- **Re-enable fsync with open\_sync:**
  - **Result:** Achieving about 8000-9000 inserts per second.

This test scenario illustrates the significant impact of **fsync** settings on PostgreSQL performance. Disabling **fsync** boosts insertion speed, but at the expense of data durability. Re-enabling **fsync** with the **open\_sync** option strikes a balance, providing a compromise between performance and data safety. It is crucial to consider these results when configuring **fsync** based on the specific requirements and priorities of your application.

Keep in mind that the decision to enable or disable **fsync** should be made carefully based on the specific requirements and priorities of the database system in question. In most production scenarios, maintaining data integrity and durability is crucial, so **fsync** is typically enabled. Performance considerations should be balanced against the risk of data loss and the importance of ACID compliance.

## PostgreSQL other monitoring use cases

Monitoring PostgreSQL is crucial for ensuring the performance, stability, and security of your database system. In addition to traditional performance monitoring, PostgreSQL offers several features and tools for addressing specific use cases. Implementing these practices will help maintain a healthy PostgreSQL database and ensure optimal performance. Adjust parameters and configurations based on specific workload characteristics and business requirements. Below, we will cover detailed steps for various monitoring use cases in PostgreSQL.

### Recipe 106: Monitoring cache hit ratio

Cache hit ratio in PostgreSQL represents the efficiency of the database's cache utilization, particularly the **shared\_buffers** setting. The **shared\_buffers** parameter defines the amount of memory allocated to PostgreSQL for caching data pages. The cache hit ratio is calculated by comparing the number of blocks retrieved from the cache (buffer cache hits) to the total number of blocks read (including those fetched from disk). A higher cache hit ratio indicates that a significant portion of database requests are being satisfied directly from the in-memory cache, minimizing the need for disk I/O operations.

PostgreSQL provides module such as the **pg\_stat\_statements** to gather statistics about SQL statements, enabling database administrators to calculate the cache hit ratio. This recipe outlines the steps to monitor and optimize the cache hit ratio in PostgreSQL, helping you ensure efficient use of memory and minimize disk I/O for improved database performance:

```
# Install pg_stat_statements extension
```

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;
```

```
# Monitor Cache Hit Ratio
```

```
SELECT
```

```
    pg_stat_statements.queryid,
```

```

pg_stat_statements.query,
pg_stat_statements.calls,
pg_stat_statements.blk_read_time,
pg_stat_statements.blk_write_time,
pg_stat_statements.shared_blk_hit,
pg_stat_statements.shared_blk_reads,
pg_stat_statements.shared_blk_written,
(pg_stat_statements.shared_blk_hit * 100) /
NULLIF(pg_stat_statements.shared_blk_reads + pg_stat_statements.shared_blk_written, 0) AS cache_hit_ratio
FROM
pg_stat_statements;

```

The above query is using the **pg\_stat\_statements** extension, which provides detailed statistics about SQL statements executed in a PostgreSQL database. It is attempting to calculate the cache hit ratio based on shared block hits and reads.

## Recipe 107: Monitor long-running query

Identifying and managing long-running queries is crucial for maintaining optimal database performance. Monitoring tools and techniques are essential to pinpoint resource-intensive queries that may impact overall system responsiveness. This recipe provides a step-by-step guide on how to effectively monitor and identify long-running queries in PostgreSQL, leveraging built-in features and extensions to gain insights into query execution times, resource utilization, and overall database health.

Run the following SQL query to identify and monitor long-running queries:

```

# Monitor Long running queries
WITH statements AS (
SELECT
    pss.calls,
    pss.mean_exec_time / 1000 AS mean_exec_time_seconds,
    pss.query,
    pr.rolname,
    EXTRACT(EPOCH FROM now()) - EXTRACT(EPOCH FROM
    pg_stat_activity.query_start) AS duration_seconds

```

```

    FROM
        pg_stat_statements pss
    JOIN pg_roles pr ON pss.userid = pr.oid
        CROSS JOIN pg_stat_activity
    WHERE
        pr rolname = current_user
    AND pg_stat_activity.state = 'active'
        AND EXTRACT(EPOCH FROM now()) - EXTRACT(EPOCH FROM
            pg_stat_activity.query_start) > 300 -- 300 seconds (5 minutes)
    )
    SELECT
        calls,
        mean_exec_time_seconds,
        query,
        duration_seconds
    FROM
        statements
    WHERE
        calls > 500
    AND duration_seconds > 0
    ORDER BY
        mean_exec_time_seconds DESC
    LIMIT 10;

```

Above query is designed to monitor long-running queries in PostgreSQL. Here is a breakdown of the key components:

### **Common table expression (CTE) statements:**

- It selects information about SQL statements from **pg\_stat\_statements** and **pg\_stat\_activity**, focusing on the current user's queries that are actively running (state = 'active').
- The **EXTRACT(EPOCH FROM now()) - EXTRACT(EPOCH FROM pg\_stat\_activity.query\_start)** calculates the duration of each query in seconds.

### **Final SELECT:**

It filters the results based on certain conditions:

- **calls > 500:** Select queries that have been called more than 500 times.
- **duration\_seconds > 0:** Exclude queries with a duration of 0 seconds.
- The result columns include the number of calls, mean execution time in seconds, the actual query, and the duration of the query in seconds.
- The results are ordered by mean execution time in descending order.
- The **LIMIT 10** ensures that only the top 10 results are returned.

## Conclusion

In conclusion, monitoring and diagnosis play a crucial role in ensuring the reliability and optimal performance of a PostgreSQL database. This chapter has provided a valuable resource of recipes, covering a wide range of monitoring and diagnosis actions that can be applied within your database environment.

By following these recipes, you will be able to build a robust monitoring setup, gaining insights into system resources, database operations, query performance, and more. Additionally, you have been introduced to the power of Prometheus and Grafana for comprehensive monitoring and visualization, as well as explored the significance of statistics and vacuuming in maintaining a healthy database. With the knowledge gained from this chapter, you are well-equipped to proactively monitor, diagnose, and optimize your PostgreSQL database, ensuring its stability and responsiveness for your applications.

In the next chapter, we navigate through the diverse landscapes of database management, you will uncover practical solutions to address a spectrum of use cases. From transaction logs and checkpoints to benchmarking and data checksums, this chapter is designed to empower administrators with actionable insights and best practices.

## 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

# Troubleshooting

## Introduction

The troubleshooting of a PostgreSQL database is a critical aspect of ensuring its optimal performance and reliability. In this chapter, we will delve into the various use cases that can arise while working with PostgreSQL and explore the convenient solutions available to database administrators. By understanding these scenarios and implementing best practices, administrators can effectively manage their databases and overcome common challenges. This chapter aims to provide a comprehensive overview of troubleshooting techniques in PostgreSQL, covering topics such as transaction logs, benchmarking, and monitoring best practices.

## Structure

In this chapter, we will cover the following topics:

- Transaction log and checkpoint
- Benchmarking
- Data checksums

## Objectives

This chapter is crafted with the following objectives in mind, steering administrators towards adept troubleshooting in PostgreSQL. First and foremost, it aims to impart a detailed understanding of the significance of transaction logs and checkpoints. By delving into topics such as log file rotation adaptation and tuning transaction logs and checkpoints, administrators can elevate database performance.

Furthermore, the chapter delves into the process of benchmarking PostgreSQL databases, leveraging the potent capabilities of pgBench. Through practical insights gained from benchmarking, administrators can measure performance, identify improvement areas, and refine their databases for optimal efficiency. The chapter also provides a comprehensive exploration of monitoring best practices for PostgreSQL databases, encompassing pg\_checksums monitoring and key parameter practices, ensuring administrators can seamlessly detect and address issues.

## **Transaction log and checkpoint**

In PostgreSQL, the transaction log and checkpoint play important roles in ensuring data integrity and recovery in the event of a system failure. Both serve distinct purposes and have different roles in the database management system. Here are the key differences between the two:

**Function:** The transaction log, also known as the **write-ahead log (WAL)**, records all changes made to the database. Its primary purpose is to provide durability and ensure data consistency in the event of a system failure. On the other hand, a checkpoint is a mechanism that optimizes database performance. It marks a specific point in the transaction log where all modified data pages have been written to disk.

**Data capture:** The transaction log captures every transactional modification before it is written to the actual data files. It serves as a safety net for recovery purposes. Conversely, a checkpoint does not capture data changes but rather indicates a point in the transaction log where all modified data pages have been flushed to disk.

**Recovery versus performance:** The transaction log is crucial for database recovery. It allows PostgreSQL to restore the database to a consistent state after a crash or unexpected shutdown. It provides the ability to replay transactions and bring the system back to the point of failure. On the other hand, a checkpoint is primarily aimed at optimizing performance. It ensures that data modifications are regularly written to disk, reducing the amount of replay needed during recovery.

**Frequency:** The transaction log is continuously updated with each transaction, capturing all changes in real-time. It maintains a sequential record of all modifications made to the database. In contrast, a checkpoint occurs at specific intervals or under certain conditions, marking a point in the transaction log where data pages are synchronized with the disk.

**Role in database operations:** The transaction log is crucial for crash recovery, replication, and high-availability solutions. It provides a reliable

source of data modifications that can be replayed to restore the database. Checkpoints, on the other hand, primarily enhance database performance by minimizing the amount of work required during recovery.

## Recipe 108: Adapt log file rotation

Log file rotation is an essential aspect of managing PostgreSQL databases. It helps to keep log files organized, prevents them from becoming too large, and ensures that important information is not lost. Proper log file rotation is crucial for maintaining optimal database performance and troubleshooting issues effectively. In this recipe, we will learn how to adapt log file rotation for PostgreSQL. This recipe assumes that you already have PostgreSQL installed and have administrative privileges, let us delve into the practical steps of adapting log file rotation for PostgreSQL.

### 1. Edit the PostgreSQL configuration file:

- a. Determine the location of the PostgreSQL configuration file (**postgresql.conf**) by executing the following query within a PostgreSQL session:

```
# Locate the configuration file  
SHOW config_file;
```

The output will provide the full path to the **postgresql.conf** file.

**Note: This above step is necessary if you plan to update the configuration directly in the postgresql.conf file.**

**If you intend to update the configuration using the ALTER SYSTEM statement instead of directly editing the postgresql.conf file, follow the alternative steps provided in the subsequent instructions.**

- b. Alter the logging configuration (**logging\_collector**) to ensure it is set to **on** for activating the PostgreSQL logging collector process. However, before making any changes to the configuration, retrieve and display the current value of the **logging\_collector** parameter.

```
# Display previous logging configurations  
show logging_collector;
```

*# Execution output of above command:*

```
logging_collector
```

```
-----  
off
```

```
(1 row)
```

```
# Enable logging collector
```

```
ALTER SYSTEM SET logging_collector = 'on';
```

- c. Alter the **log\_rotation\_size** configuration by adjusting it to specify the maximum size of a log file before rotation occurs. Set the value according to your requirements. Properly configuring this parameter is crucial to prevent log files from becoming too large, which can negatively impact database performance. However, before making any changes to the configuration, retrieve and display the current value of the **log\_rotation\_size** parameter.

```
# Display previous log_rotation_size configuration  
SHOW log_rotation_size;  
# Execution output of above command:  
log_rotation_size
```

```
-----  
10 MB
```

```
(1 row)
```

```
# Set log rotation size to 100MB
```

```
ALTER SYSTEM SET log_rotation_size = '100 MB';
```

- d. Alter the **log\_rotation\_age** configuration parameter to specify the maximum age of a log file before rotation occurs. The value is specified in time units (microsecond, milliseconds, seconds, minutes, hours, or days). Adjust it based on your needs. Regular log rotation ensures that log files do not accumulate excessively, which helps maintain optimal performance. However, before making any changes to the configuration, retrieve and display the current value of the **log\_rotation\_age** parameter.

```
# Display previous log_rotation_age configurations  
SHOW log_rotation_age;  
# Execution output of above command:  
log_rotation_age
```

```
-----  
0
```

```
(1 row)
```

```
# Set log rotation age to 1 day
```

```
ALTER SYSTEM SET log_rotation_age = '1d';
```

## 2. Apply configuration changes:

- a. Save the changes to the configuration file and apply by executing **pg\_reload\_conf()** file.

```
# Save the changes and apply  
SELECT pg_reload_conf();
```

### 3. Verify log rotation:

- a. Open a terminal and connect to the PostgreSQL database.  

```
psql -U <username> -d <database_name>
```
- b. Execute a few queries or perform database operations to generate some log data.
- c. Locate the PostgreSQL log directory. The location of the log files may vary based on your system configuration.
- d. Check if log rotation is working by examining the log files. Observe that log files are being rotated based on the size and age parameters specified in the configuration file.

Remember to monitor log files periodically to ensure they are being rotated as expected and to retain an appropriate amount of log history for analysis and troubleshooting purposes.

## Recipe 109: Tune transaction log and checkpoints

Tuning the transaction log and checkpoints in PostgreSQL is crucial for optimizing database performance and ensuring data durability. This recipe will guide you through the necessary steps to configure these parameters effectively. By following this recipe, you can maximize the performance and reliability of your PostgreSQL database:

### 1. Understand the basics:

Before tuning the transaction log and checkpoints, it is important to understand their purpose and how they work in PostgreSQL:

- **Transaction log:**

- The transaction log (or WAL) stores a record of changes made to the database.
- It ensures durability by allowing crash recovery and replication.
- The transaction log is crucial for performance since it determines how often checkpoints occur.

- **Checkpoints:**

- Checkpoints are points in the transaction log where all data is

- They are essential for crash recovery and prevent the transaction log from growing infinitely.
- Checkpoints also affect database performance, as they introduce some I/O overhead.

## 2. Analyze the current configuration:

Before making any changes, it is essential to analyze the current configuration of the transaction log and checkpoints:

### a. Checkpoint configuration:

Run the following command to retrieve the current checkpoint-related configuration parameters:

```
SHOW checkpoint_timeout;  
SHOW checkpoint_warning;  
SHOW checkpoint_completion_target;  
SHOW checkpoint_flush_after;
```

### b. Transaction log configuration:

Use the following command to check the current transaction log-related configuration:

```
SHOW wal_level;  
SHOW max_wal_size;  
SHOW min_wal_size;
```

## 3. Adjust checkpoint configuration:

Optimizing the checkpoint configuration is crucial for balancing performance and recovery time.

### a. Adjusting `checkpoint_timeout`:

- Analyze the average time taken by a checkpoint to complete.
- Set the `checkpoint_timeout` value accordingly (for example, 10 minutes):

```
ALTER SYSTEM SET checkpoint_timeout TO '10min';
```

### b. Apply the changes:

- Reload the configuration to apply the changes:

```
SELECT pg_reload_conf();
```

## 4. Configure transaction log:

Optimizing the transaction log configuration can significantly impact database performance and crash recovery.

a. **Adjusting wal\_level:**

- i. Set the **wal\_level** parameter based on your requirements (for example, **replica** for replication or **logical** for logical decoding):

```
ALTER SYSTEM SET wal_level TO 'replica';
```

b. **Adjusting max\_wal\_size and min\_wal\_size:**

- i. Analyze the average rate of transaction log generation.
- ii. Set **max\_wal\_size** and **min\_wal\_size** based on the duration you want the transaction log to retain (for example, 16GB and 4GB, respectively):

```
ALTER SYSTEM SET max_wal_size TO '16GB';
```

```
ALTER SYSTEM SET min_wal_size TO '4GB';
```

c. **Apply the changes:**

- i. Reload the configuration to apply the changes:

```
SELECT pg_reload_conf();
```

## 5. Monitor and fine-tune:

After applying the configuration changes, closely monitor the database performance and make further adjustments if required.

a. **Monitoring:**

- Monitor database performance metrics such as transaction throughput, checkpoint frequency, and disk I/O.
- Use tools like **pg\_stat\_bgwriter** and **pg\_stat\_progress\_vacuum** to analyze the checkpoint activity and vacuuming process.

b. **Fine-tuning:**

- If necessary, revisit the configuration parameters and adjust them based on the observed performance and recovery times.
- Regularly monitor and fine-tune the parameters to maintain optimal performance.

## Benchmarking

Benchmarking in PostgreSQL involves evaluating the performance and scalability of the database system under various workloads and configurations. It helps users assess the effectiveness of their hardware, configuration settings, and application design. PostgreSQL offers improved features and optimizations for benchmarking, allowing users to obtain more accurate results.

Several tools are available to conduct benchmarking in PostgreSQL 15 that assist in measuring various aspects of the database's performance. Some of the commonly used tools for benchmarking PostgreSQL include pgbench, HammerDB, Apache JMeter, pgBadger, pg\_stat\_statements, and pgreplay. These tools provide different functionalities, such as generating synthetic workloads, stress testing, monitoring query performance, and analyzing query statistics, enabling users to benchmark and optimize their PostgreSQL deployments effectively.

### Recipe 110: Benchmark performance with pgbench

In this recipe, we will explore how to benchmark the performance of a PostgreSQL 15 database using the pgbench tool. Benchmarking is essential for evaluating the performance and capacity of your database, allowing you to identify potential bottlenecks and optimize your system accordingly.

#### 1. Install and configure pgbench for benchmarking:

- To install the **pgbench** utility for PostgreSQL 15, you can use the **yum** package manager. Open a terminal or command prompt and run the following command:

```
sudo yum install -y postgresql15-contrib
```

This command will install the PostgreSQL 15 **contrib** package, which includes **pgbench** along with other useful extensions and tools for PostgreSQL. Once the installation is complete, we can proceed with configuring and using pgbench to benchmark the performance of your PostgreSQL 15 database.

- Verify **pgbench** installed version by execution the following command:

```
# command to verify the pgbench version
```

```
./pgbench --version
```

```
# pgbench version output with above execution
```

```
pgbench (PostgreSQL) 15.2
```

The above execution and output display the version information of pgbench. In this specific case, the version output of the **pgbench** utility is **pgbench (PostgreSQL) 15.2.**

## 2. Initialize the database and run a benchmark test with the default configuration:

- Open a terminal or command prompt and execute the following command to initialize the database with the default parameter.

```
pgbench -i postgres
```

Referring to *Figure 13.1*, The above command was executed to initialize benchmark data in the **postgres** database. The output indicates the different stages of the data initialization process. Firstly, old tables were dropped to start fresh. Then, the required benchmark tables were created. The data generation process took 0.07 seconds, resulting in 100,000 tuples being generated. After that, vacuuming was performed to optimize storage layout. Finally, primary keys were created for data integrity. Please refer to the following figure:

```
[postgres@pgsrdev ~]$ pgbench -i postgres
dropping old tables...
creating tables...
generating data (client-side)...
100000 of 100000 tuples (100%) done (elapsed 0.07 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done in 0.17 s (drop tables 0.01 s, create tables 0.00 s, client-side generate 0.09 s, vacuum 0.03 s, primary keys 0.04 s).
[postgres@pgsrdev ~]$
```

**Figure 13.1:** pgbench initialize database

Overall, the entire process took 0.17 seconds, with individual times for each step: dropping tables (**0.01 s**), creating tables (**0.00 s**), client-side data generation (**0.09 s**), vacuuming (**0.03 s**), and creating primary keys (**0.04 s**). This prepares the database for running benchmark tests and evaluating PostgreSQL's performance.

## 3. Run the Benchmark:

- Open a terminal or command prompt and execute the following command to run the benchmark:

```
pgbench -c 200 -j 10 -T 60 -U postgres -h 192.168.187.134 -p 5432 postgres
```

Referring to *Figure 13.1*, The command **pgbench -c 200 -j 10 -T 60 -U postgres -h 192.168.187.134 -p 5432 postgres** was executed to perform a benchmark test. The output provides specific details of the benchmark results. It indicates that the benchmark used the TPC-B transaction type, with a scaling factor of 1. The test was run in simple query mode with 200 clients and 10 threads. The

benchmark was executed for a duration of 60 seconds. During the test, a total of 65,462 transactions were processed, and there were no failed transactions (0.000%). The average latency was 179.817 milliseconds, and the initial connection time was 1431.498 milliseconds. The transactions per second (**tps**) achieved during the test, excluding the initial connection time, was 1112.242349. These results provide valuable performance metrics for evaluating the PostgreSQL database under the specified workload:

```
[postgres@pgsrvdev ~]$ pgbench -c 200 -j 10 -T 60 -U postgres -h 192.168.187.134 -p 5432 postgres
Password:
pgbench (15.2)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 200
number of threads: 10
maximum number of tries: 1
duration: 60 s
number of transactions actually processed: 65462
number of failed transactions: 0 (0.000%)
latency average = 179.817 ms
initial connection time = 1431.498 ms
tps = 1112.242349 (without initial connection time)
[postgres@pgsrvdev ~]$
```

**Figure 13.2: pgbench - Execute benchmark**

#### 4. Analyze the results:

- a. Once the benchmark is complete, pgbench will output the results in the terminal or command prompt.
- b. Use the gathered data to analyze and evaluate the performance of PostgreSQL database and identify areas for improvement.

By following this recipe, we have learned how to benchmark the performance of a PostgreSQL 15 database using the pgbench tool. Benchmarking allows to evaluate database's performance, identify potential bottlenecks, and optimize the system accordingly. Use the results to make informed decisions about improving PostgreSQL database's performance and scalability.

### Recipe 111: Practice with key parameters in PostgreSQL

In this recipe, we will explore some of the key parameters and learn how to practice with them to optimize your PostgreSQL database. Follow the steps to get started:

#### 1. Locate the configuration file:

- a. Determine the location of the PostgreSQL configuration file

(**postgresql.conf**) by executing the following query within a PostgreSQL session:

```
# Locate the configuration file  
SHOW config_file;
```

The output will provide the full path to the **postgresql.conf** file.

## 2. Backup the configuration file:

Before making any changes, it is essential to create a backup of the original **postgresql.conf** file. This will allow to revert to the previous configuration if needed. You can use commands like cp on Linux or copy on Windows:

```
# Backup Config file on Linux  
cp /path/to/original/postgresql.conf  
/path/to/backup/postgresql.conf.bak
```

```
# Backup Config file on (Command Prompt)  
copy "C:\Path\To\Original\postgresql.conf"  
"C:\Path\To\Backup\postgresql.conf.bak"
```

## 3. Practice with key parameters:

**Workload-related parameters:** Control and optimize the database's memory usage, concurrent connections, and internal operations, influencing performance and resource utilization for various workloads as detailed in the [Table 13.1](#):

Parameters	Descriptions
<b>max_connections</b>	Adjust the maximum number of concurrent connections allowed to the database.
<b>shared_buffers</b>	Configure the amount of memory allocated for shared buffers.
<b>work_mem</b>	Set the amount of memory used for internal sort operations and hash tables in a query.
<b>effective_cache_size</b>	Indicate the amount of memory available for caching data.
<b>maintenance_work_mem</b>	Set the amount of memory used for maintenance operations.

**Table 13.1:** Workload-related parameters

**Write-ahead logging -related parameters:** Enable fine-tuning the write-ahead logging process, controlling the amount of information written to the WAL, checkpoint frequency, and replication settings,

ensuring data durability and efficient recovery as detailed in the [Table 13.2](#):

Parameters	Descriptions
<b>wal_level</b>	Set the level of information written to the write-ahead log.
<b>max_wal_size/ min_wal_size</b>	Define the size of WAL segments.
<b>checkpoint_timeout/checkpoint_completion_target</b>	Adjust the frequency and time allowed for automatic checkpoints.

**Table 13.2:** WAL-related parameters

**Autovacuum-related parameters:** Enable to manage and adjust the behaviour of the autovacuum process, which automatically performs maintenance tasks like a vacuum and analyzes to reclaim storage and optimize query performance as detailed in the [Table 13.3](#):

Parameters	Descriptions
<b>autovacuum</b>	Enable or disable the autovacuum process.
<b>Autovacuum_vacuum_scale_factor/autovacuum_analyze_scale_factor</b>	Set threshold for automatic vacuum and analyze processes.

**Table 13.3:** Autovacuum-related parameters

**Replication-related parameters:** This allows you to configure and control synchronous replication settings, ensuring data consistency and high availability in a replicated database environment as detailed in the [Table 13.4](#):

Parameters	Descriptions
<b>synchronous_commit</b>	Enable or disable synchronous replication commit.
<b>synchronous_standby_names</b>	Specify the synchronous replication standby servers.

**Table 13.4:** Replication-related parameters

#### **4. Save and apply changes:**

- a. After making the desired changes to the **postgresql.conf** file, save the file and close the text editor.
- b. Restart PostgreSQL to apply the changes.

#### **5. Observe the impact:**

- a. Monitor the database performance and observe how the changes to key parameters affect the behaviour of your PostgreSQL database.

## **Data checksum**

Data checksums are a mechanism to ensure the integrity of data stored in PostgreSQL. When enabled, PostgreSQL calculates and stores checksums for each block of data in the database. During reads, the checksums are recalculated, and any discrepancy indicates possible data corruption. This helps in early detection of issues, providing a higher level of data reliability. Enabling data checksums is especially valuable in critical production environments where data integrity is paramount, offering an additional layer of protection against silent data corruption.

### **Recipe 112: Working with pg\_checksums**

The tool allows you to enable or disable data checksums on your PostgreSQL cluster. Data checksums help detect and prevent data corruption, ensuring the reliability of your database. Here is a step-by-step guide on working with **pg\_checksums**.

1. **Execute checksum:** Before making any changes, shut down the PostgreSQL cluster and check the current checksum status of your PostgreSQL cluster using the following command:

```
# Stop PostgreSQL service
```

```
systemctl stop postgresql-15
```

```
# Execute the checksum
```

```
pg_checksums --check -D /path/to/data_directory
```

When executing the command, if checksums are not enabled in the database, the following error is encountered.

```
pg_checksums: error: data checksums are not enabled in cluster
```

This error signifies that data checksums are not presently active in your PostgreSQL cluster. The **--check** option is utilized to verify the

integrity of checksums on data pages. However, for this operation to succeed, checksums must be enabled in the cluster beforehand.

An alternative approach to confirm the status of **pg\_checksums** in PostgreSQL involves querying the **pg\_settings** system view. Specifically, you will want to check the value of the **data\_checksums** configuration parameter. Here is the SQL command:

```
# Stop PostgreSQL service
SHOW data_checksums;
##### Output #####
data_checksums
-----
off
(1 row)
```

### Resolution steps:

2. **Enable data checksums:** To resolve this issue, enable data checksums in your PostgreSQL cluster using the following commands:

```
# Stop PostgreSQL service
systemctl stop postgresql-15
# Execute the checksum
pg_checksums --enable -D /var/lib/pgsql/15/data/
# Start PostgreSQL service
systemctl start postgresql-15
```

The above sequence of commands stops PostgreSQL, enables checksums, and then restarts PostgreSQL to apply the changes.

3. **Verify checksums:** After enabling checksums, confirm that the checksums have been successfully applied, the execution of below command should now execute without errors, indicating that data checksums are enabled and operational:

```
# Execute checksums
pg_checksums --check -D /var/lib/pgsql/15/data/
##### Output #####
Checksum operation completed
Files scanned: 2739
Blocks scanned: 81291
Bad checksums: 0
Data checksum version: 1
```

The above output of the **pg\_checksums --check** command provides information about the checksum verification operation on a PostgreSQL

database cluster. Here is a breakdown of the output:

- The command scanned a total of 2739 files within the PostgreSQL data directory. These files include various components of the database, such as data files, indexes, and other system-related files.
- This line shows the total number of blocks (database storage units) that were scanned during the checksum verification. Each block corresponds to a certain amount of data within the files.
- The absence of any **Bad checksums** (with a value of **0**) indicates that all the checksums for the scanned files and blocks matched the expected values. A non-zero value here would suggest that there were discrepancies, indicating potential data corruption.

Enabling data checksums is a crucial step in maintaining data integrity within PostgreSQL. It involves temporarily stopping the PostgreSQL cluster, enabling checksums using **pg\_checksums --enable**, and then restarting the cluster. Once checksums are enabled, subsequent calls to **pg\_checksums --check** can successfully verify the integrity of data pages.

This proactive approach ensures that PostgreSQL can detect and prevent data corruption, enhancing the reliability of the database.

## Conclusion

In conclusion, this chapter has provided an in-depth exploration of troubleshooting techniques in PostgreSQL. By examining various use cases and implementing the recommended solutions, database administrators can effectively manage their PostgreSQL databases. The topics covered, including transaction logs, benchmarking, connection pooling, and monitoring best practices, equip administrators with the knowledge and tools necessary to optimize database performance, enhance scalability, and ensure the reliability of their PostgreSQL systems. By applying these techniques, administrators can address common challenges and maintain a high-performing PostgreSQL environment.

## 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

AccessShareLock [418](#)  
active session [419](#)  
    monitoring [419, 420](#)  
advanced object-relational table concept [29](#)  
concurrent index, working with [35, 36](#)  
indexing technique, with  
    PostgreSQL database [32-34](#)  
    tables inheritance [30-32](#)  
Amazon Web Services (AWS) [54](#)  
architecture-based replication [210](#)  
    master-master replication [210](#)  
    master-slave replication [210](#)  
archive\_cleanup\_command [165](#)  
archive logs [161](#)  
    deleting, with proactive solution [165-168](#)  
audit log  
    using, with PostgreSQL trigger [314-317](#)  
automatic partitioning [187](#)  
autovacuum [423](#)  
    monitoring [427](#)  
autovacuum-related parameters [448](#)  
AWS database migration service  
    (AWS DMS)  
    utilizing [100-118](#)  
AWS EC2  
    PostgreSQL instance, managing with [56-64](#)  
AWS EC2 instance  
    native backup or restore with [72-79](#)  
AWS RDS  
    PostgreSQL instance, managing with [65-72](#)  
AWS RDS instance  
    backup or restore with [80-88](#)

## B

backup  
    monitoring [363](#)

backup performance  
improving 364, 365  
backups  
automating 332-334  
backup tools  
Bacula 335  
Barman 335  
pgBackRest 335  
pg\_dump 335  
pg\_probackup 335  
WAL-E 335  
backward compatibility  
exploring 17, 18  
upgrade on major release 19-22  
upgrade on minor release 18, 19  
Bacula 335  
Barman 335, 349  
advantages 349  
configuring 351-356  
installing 350  
benchmarking 443, 444  
performance, with pgbench 444-446  
pgbench, configuring for 444  
pgbench, installing for 444  
bloat management 423, 424  
BLOB 287  
data types, importing 287-289

## C

CAP theorem 209, 210  
checkpoint processing  
optimizing 154, 155  
versus, WAL writer process 154  
checkpoints 439  
tuning 441-443  
Citus 202  
installing 202  
sharding, configuring with 202-204  
cloud deployment option 55  
cloud service models  
IaaS 54  
PaaS 54  
SaaS 54  
cloud solution for PostgreSQL 54  
cloud deployment option 55

- cloud service models 54
- cluster 4
  - common table expression (CTE) 36
    - calculations, adding with 39
    - joining, with main query 40
    - utilized, for improving complex queries 36
    - utilized, for SELECT queries 37
    - WITH queries 38
  - common table expressions (CTE) 24
- composite partitioning 173
- concurrent index 35
- continuous archiving 152
  - administering 152, 153
  - implementing 161
- crash recovery 371, 372

## D

- data access 250
  - exploring 250
- database access management 24
- database admin-specific start-up logs
  - parsing 48, 49
- database backup 325, 326
- database consistency and integrity 279
- database index 26
- database instance and object hierarchy 24-26
- database locks
  - monitoring 417-419
- database migration roadmap 99
- database privileges 293
  - database objects, granting 296
  - default privileges, using 297
  - fetch object-level privileges 294, 295
  - fetch schema-level privileges 294
  - fetch specific routine privileges 295
  - fetch specific table privileges 295
  - list database-level privileges 293
  - revoking 296
  - role authorization 297
- database recovery 368
- database server and client environment 9, 10
- database storage layout 41, 42
  - database admin specific start-up logs, parsing 48, 49
  - database log directory adaptation 44, 45
  - database storage adaptation, for PostgreSQL database 42-44

PostgreSQL TOAST, working with 45-47  
data checksums 448  
data control language (DCL) 263, 264  
data definition language (DDL) 261  
    ALTER statement 262  
    CREATE statement 261  
    DROP statement 262  
data integrity 254  
data manipulation language (DML) 260  
    DELETE statement 261  
    INSERT statement 261  
    SELECT statement 260  
    UPDATE statement 261  
data restoring  
    with Barman 383-385  
dataset export/import  
    working with 266-268  
datasets  
    loading, from spreadsheet/flat files 269, 270  
data synchronization-based replication 210  
    asynchronous replication 210  
    synchronous replication 210  
db2topg 125  
    using, for migration to PostgreSQL 125-127  
DDL adaptation 264-266  
delay standby 222  
    for performing PITR recovery 225, 226  
    setting up 222-225  
diagnosis 394  
differential backup 356-359  
dropped table recovery  
    versus, damaged table recovery 385

## E

Elastic Compute Cloud (EC2) instance 54  
    exploring 55  
ExclusiveLock 418

## F

failover 235  
    with repmgr 237, 238  
foreign data wrapper (FDW) 204, 205  
    setting up, for PostgreSQL migration 127-135  
foreign key constraints 280-282  
fsync 427-432

## G

[global] section 345  
Grafana 397  
    configuring 406-409  
    installing 406  
    managing 397  
graphical user interface  
    (GUI) tools 10

## H

HAProxy 244  
    using, for HA solution 244-247  
hash-based sharding 201  
hash partitioning 173  
heterogeneous database system migration 97  
high availability 209  
homogeneous database system migration 97  
horizontal partitioning 183-186  
horizontal sharding 201  
hung query 410  
    terminating 410-412

## I

incremental backup 356, 360  
incremental/differential restore process 376, 377  
    with Barman 377-380  
    with pgBackRest 380-383  
Infrastructure as a Service (IaaS) 54  
installation, PostgreSQL  
    database configuration file 8  
    working with, from binaries 5, 6  
    working with, from source code 6, 7

## J

JavaScript Object Notation (JSON) 274  
join  
    exploring 272-274  
JSON data  
    querying 274-277

## K

key parameters  
    practicing with 446-448

## L

LDAP authentication [322](#)

enabling [323](#), [324](#)

list-based sharding [201](#)

list partitioning [172](#)

load balancing [239](#)

log file rotation [439](#)

adapting [439-441](#)

logical backup

working [328](#), [329](#)

logical replication [208](#)

log\_statement [317](#)

working with [317-322](#)

## M

maintenance\_work\_mem [50](#)

managed option [55](#)

master-master replication [210](#)

master-slave replication [210](#)

max\_connection [50](#)

migration [95](#), [96](#)

migration methodology [96](#), [97](#)

monitoring [394](#)

database monitoring [395](#)

script-based monitoring [395](#)

system resource monitoring [394](#), [395](#)

monitoring script [395](#)

monitoring use cases [432](#)

cache hit ratio monitoring [432](#), [433](#)

long-running query monitoring [433-435](#)

multi-master replication [214](#)

Postgres, scaling with [214-222](#)

## P

partitioning [172](#)

automatic partitioning [187](#)

composite partitioning [173](#)

exploiting [176-179](#)

hash partitioning [173](#)

horizontal partitioning [183-186](#)

list partitioning [172](#)

range partitioning [172](#)

setting up [173-175](#)

versus, sharding [200](#)

- vertical partitioning [186](#)
- partition management
  - using custom scripts, with scheduler [194-196](#)
  - with pg\_partman extension [188-193](#)
  - with PostgreSQL 15 [179-182](#)
- partition pruning [197](#)
  - working [197-200](#)
- pgAdmin [10](#)
  - configuring [11, 12](#)
  - installing [10, 11](#)
- pgaudit [310](#)
  - configuring [311-314](#)
  - installing [310, 311](#)
- pgBackRest [335, 343](#)
  - advantages [343](#)
  - configuring [344-349](#)
  - installing [343, 344](#)
  - working with [336](#)
- pgbackrest check command [347](#)
- pgbench
  - performance, benchmarking with [444-446](#)
- pg\_checksums
  - working with [449-451](#)
- pgcrypto
  - encryption, configuring [306-310](#)
- pg\_dump [335](#)
- pgloader [118](#)
  - MariaDB migration, to PostgreSQL
    - on EC2 instance [121-124](#)
  - migrating database mode [119](#)
  - setting up [119, 120](#)
- pg\_partman [188](#)
- pg\_probackup [335](#)
  - configuring [337-342](#)
  - installing [336](#)
  - working with [336](#)
- pg\_restore utility [99](#)
- pg tables [26](#)
- physical backup [329](#)
  - working with [329-332](#)
- point-in-time recovery (PITR) [162, 372-376](#)
  - performing, with delay standby [225, 226](#)
  - recovering with [372](#)
- PostgreSQL [1, 2](#)
  - architecture [3](#)
  - database structure [4, 5](#)

- history [3](#)
- installation [5](#)
  - internal components [24](#)
  - scaling, with multi-master replication [214-222](#)
  - scaling, with primary-standby replication [211-213](#)
- PostgreSQL 15 [1](#)
  - auditing [310](#)
  - continuous archiving [161-164](#)
  - features [2](#)
- PostgreSQL Automatic Failover [239](#)
  - deploying, for HA solution [239-244](#)
- PostgreSQL autovacuum problem
  - debugging [160](#)
- PostgreSQL CAST operator [277](#)
  - working [277, 278](#)
- PostgreSQL client application [9](#)
- PostgreSQL data
  - querying [255-257](#)
- PostgreSQL database structural object
  - discovering [15-17](#)
- PostgreSQL database system migration types
  - database migration roadmap [99, 100](#)
  - EC2 to RDS instance on AWS [100-118](#)
  - heterogeneous database system migration [97](#)
  - homogeneous database system migration [97](#)
  - on-premise to AWS EC2 instance migration [98, 99](#)
- PostgreSQL GUI tool [10](#)
- PostgreSQL index type [26](#)
- PostgreSQL instance
  - managing, with AWS EC2 [56-65](#)
  - managing, with AWS RDS [65-72](#)
- PostgreSQL memory configuration [50](#)
  - local memory area [50, 51](#)
  - shared memory area [51](#)
- PostgreSQL migration
  - db2topg, using for [125-127](#)
  - foreign data wrapper, setting up for [127-135](#)
- PostgreSQL remote access
  - working with [14, 15](#)
- PostgreSQL replication [88](#)
  - working with, on AWS [88-94](#)
- PostgreSQL role management and authorization [298-302](#)
- PostgreSQL schema
  - adaptation with [26-28](#)
- PostgreSQL server access solution
  - working [12-14](#)

PostgreSQL trigger  
audit log, using with 314-317  
primary-standby replication  
Postgres, scaling with 211  
setting up 211-213  
proactive solution 165  
Prometheus 397  
configuring 398, 399  
installing 398  
managing 397  
postgres\_exporter, configuring 400-403  
postgres\_exporter, installing 400  
postgres\_exporter, integrating 404, 405  
proxy server 239  
psql 270  
Python and Java connection 282-286

## **Q**

query  
access, structuring with psql 270, 271  
executing, with shell script 258-260  
query monitoring 416  
enabling 416  
query plan 421  
working with 421, 422

## **R**

range-based sharding 201  
range partitioning 172  
rebalancer 204  
recovery planning 368, 369  
recovery from physical backup 370, 371  
recovery from logical backup 369, 370  
recovery.signal file 164  
recovery\_target parameter 163  
Recovery Time Objectives (RTO) 369  
Relational Database Service (RDS) instance 54  
exploring 55  
replication 208-210  
logical replication 208  
monitoring 248  
streaming replication 208  
synchronous replication 208  
replication classification 210  
architecture-based replication 210

- data synchronization-based replication [210](#)
- replication slot [213](#)
  - using [213, 214](#)
- repmgr [228](#)
  - configuring [229-234](#)
  - installing [229](#)
  - prerequisites [228](#)
- RHEL system [152](#)

## S

- SCHEMA [4](#)
- schema level backup [360](#)
  - working with [360-363](#)
- schema level restore
  - working with [387-390](#)
- schemas [25](#)
- script-based monitoring
  - connection monitoring [396](#)
  - replication lag [396, 397](#)
  - table size monitoring [396](#)
  - utilizing [395](#)
- Secure Sockets Layer (SSL) [303](#)
- self-managed options [55](#)
- server control and auditing [292](#)
- sharding [172](#)
  - configuring, with Citus data [202-204](#)
  - versus, partitioning [200](#)
- sharding strategies [201](#)
  - hash-based sharding [201](#)
  - horizontal sharding [201](#)
  - list-based sharding [201](#)
  - range-based sharding [201](#)
  - vertical sharding [201](#)
- shared\_buffers [51](#)
- Software as a Service (SaaS) [54](#)
- SQL dump
  - exploring [327](#)
  - file system level backup [327](#)
- SQL statement transaction
  - sizing [150](#)
- SSL authentication
  - setting up, in PostgreSQL [303-305](#)
- SSL/TLS authentication [302, 303](#)
- standby server [227](#)
  - promoting, as primary server [227, 228](#)

- statistic analyzer
  - working with 412
- statistic collector
  - working with 412
- statistic controller
  - enabling 413
- statistics 412
  - analyzing 414
  - querying 413
- streaming replication 208
- subqueries
  - exploring 272-274
- switchover 235
  - with repmgr 235-237
- synchronous replication 208
- system resource monitoring 394, 395
- system view
  - monitoring with 415

## T

- table operations 251, 254
  - data insertion 252
  - data modification 253
  - data retrieval 252, 253
  - table creation 251, 252
- tables 25, 251
  - column 25
  - row 25
- table size monitoring 396
- tables recovery
  - working with 386, 387
- TDE encryption
  - exploring 305, 306
- temp\_buffer 51
- The Oversized-Attribute Storage
  - Technique (TOAST) mechanism 45
  - working with 46, 47
- transaction control language (TCL) 262
  - COMMIT statement 262
  - ROLLBACK statement 263
  - SAVEPOINT statement 263
- transaction log 137, 138, 438
  - tuning 441-443
- Transparent Data Encryption (TDE) 305
- troubleshooting 437

## **U**

unique constraints [279](#), [280](#)

## **V**

vacuuming [422](#), [425](#)  
    considerations [423-426](#)  
vacuum process  
    automating [155](#), [156](#)  
    in PostgreSQL database [157-160](#)  
vertical partitioning [186](#)  
vertical sharding [201](#)  
virtualxid locktype [418](#)

## **W**

WAL buffers [51](#)  
WAL-E [335](#)  
WAL file  
    configuring [141](#), [142](#)  
WAL performance parameter  
    configuring [151](#)  
    managing [151](#), [152](#)  
WAL-related parameters [447](#)  
WAL segment  
    checksum [140](#)  
    control information [140](#)  
    header [140](#)  
    internal layout [139](#)  
    page header [140](#)  
    records [140](#)  
WAL writer process [153](#)  
    managing [153](#)  
    versus checkpoint processing [154](#)  
workload-related parameters [447](#)  
work\_mem parameter [50](#)  
Write-Ahead Logging (WAL) [139](#), [208](#)  
    archive mode, enabling/disabling [143-146](#)  
    benefits [142](#), [143](#), [164](#), [165](#)  
    performance, enhancing [164](#)  
    remote WAL Archive options,  
        working with [146](#), [147](#)  
WAL compression option, for  
    space management [148-150](#)

## **X**

XLOG record [140](#)

internal layout [140](#), [141](#)

## Z

Zstandard (ZSTD) compression [148](#)