6TH EDITION

Mastering PostgreSQL 17

Elevate your database skills with advanced deployment, optimization, and security strategies

HANS-JÜRGEN SCHÖNIG

Mastering PostgreSQL 17

Elevate your database skills with advanced deployment, optimization, and security strategies

Hans-Jürgen Schönig

<packt>

Mastering PostgreSQL 17

Group Product Manager: Apeksha Shetty

Publishing Product Manager: Chayan Majumdar

Book Project Manager: Aparna Nair

Senior Editor: Rohit Singh

Technical Editor: Rahul Limbachiya

Copy Editor: Safis Editing

Proofreader: Rohit Singh

Indexer: Tejal Soni

Production Designer: Gokul Raj S.T

DevRel Marketing Executive: Nivedita Singh

First published: January 2018

Second edition: October 2018

Third edition: November 2019

Fourth edition: November 2020

Fifth edition: January 2023

Sixth edition: December 2024

Production reference: 1061224

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83620-597-5

`www.packtpub.com`

Contributors

About the author

Hans-Jürgen Schönig has 25 years of experience with PostgreSQL. He is the CEO of a PostgreSQL consulting and support company called CYBERTEC PostgreSQL International GmbH, which has successfully served countless customers around the globe. Before founding CYBERTEC PostgreSQL International GmbH in 2000, he worked as a database developer at a private research company that focused on the Austrian labor market, where he primarily worked on data mining and forecast models. He has also written several books about PostgreSQL.

About the reviewer

Rajneesh Verma is a seasoned technology leader with over 18 years of experience, currently heading IT and security at CYBERTEC PostgreSQL Services. Specializing in enterprise design and architecture, he is passionate about building reliable platforms, including database-as-a-service solutions and innovative setups for cloud and on-premises environments, IAAS, PAAS, and security. His technical stack includes PostgreSQL, SQL Server, Python, Flask, HTML, and CSS. Curious by nature, Rajneesh loves figuring out “how things work” and shares this enthusiasm through book reviews, exploring tech, leadership, and innovation with technical depth and approachable storytelling.

Table of Contents

Preface

Welcome to *Mastering PostgreSQL 17*, the ultimate guide to unlocking the full potential of one of the world's most popular open source relational databases – PostgreSQL. With decades of history and a community-driven development process, PostgreSQL has become the go-to choice for organizations seeking a robust, scalable, and reliable database solution. This has been true for many years and this will be the case for many years to come.

In this book, we'll take you on a comprehensive journey through the latest features and enhancements in PostgreSQL 17, the newest major release of the database system. Whether you're a seasoned DBA looking to expand your skillset or a developer seeking to improve your application's performance and scalability, this book is designed to help you master the art of working with PostgreSQL and it will hopefully be an enjoyable thing to read that helps you to understand things better, be more productive, and simply have a better time.

Mastering the art of handling data is an ever more important skill that is important to have. In a digital world, "data" is more or less the "new oil" – an important asset that drives the world and the importance of data is growing as we speak. Every sector of IT is data-driven. It does not matter whether you are at the forefront of machine learning or whether you are working on bookkeeping software – at the end of the day, IT is all about data.

PostgreSQL has become a hot technology in the area of open source, and it is an excellent technology to store and process data in the most efficient way possible. This book will teach you how to use PostgreSQL in the most professional way and explain how to operate, optimize, and monitor this core technology, which has become so popular over the years.

By the end of the book, you will be able to use PostgreSQL to its utmost capacity by applying advanced technology and cutting-edge features.

Who this book is for

This book is tailored for database administrators, PostgreSQL developers, and IT professionals aiming to implement advanced functionalities and tackle complex administrative tasks using PostgreSQL 17. A foundational understanding of PostgreSQL and core database concepts is essential, along with familiarity with SQL. Prior experience in database administration will enhance your ability to leverage the advanced techniques discussed throughout the book.

What this book covers

Chapter 1, *What is New in PostgreSQL 17*, guides you through the most important features that have made it into the new release of PostgreSQL and explains how those features can be used.

Chapter 2, *Understanding Transactions and Locking*, explains the fundamental concepts of transactions and locking. Both topics are key requirements to understand storage management in PostgreSQL.

Chapter 3, *Making Use of Indexes*, introduces the concept of indexes, which are the key ingredient when dealing with performance in general. You will learn about simple indexes as well as more sophisticated concepts.

Chapter 4, *Handling Advanced SQL*, unleashes the full power of SQL and outlines the most advanced functionality a query language has to offer. You will learn about windowing functions, ordered sets, hypothetical aggregates, and a lot more. All those techniques will open a totally new world of functionality.

Chapter 5, *Log Files and System Statistics*, explains how you can use runtime statistics collected by PostgreSQL to make operations easier and to debug the database. You will be guided through the internal information-gathering infrastructure.

Chapter 6, *Optimizing Queries for Good Performance*, is all about good query performance and outlines optimization techniques that are essential to bringing your database up to speed to handle even bigger workloads.

Chapter 7, *Writing Stored Procedures*, introduces you to the concept of server-side code such as functions, stored procedures, and a lot more. You will learn how to write triggers and dive into server-side logic.

Chapter 8, *Managing PostgreSQL Security*, helps you to make your database more secure and explains what can be done to ensure safety and data protection at all levels.

Chapter 9, *Handling Backup and Recovery*, helps you to make copies of your database to protect yourself against crashes and database failure.

Chapter 10, *Making Sense of Backups and Replication*, follows up on backups and recovery and explains additional techniques, such as streaming replication, redundancy, and a lot more. It covers the most advanced topics.

Chapter 11, *Deciding on Useful Extensions*, explores extensions and additional useful features that can be added to PostgreSQL.

Chapter 12, *Troubleshooting PostgreSQL*, completes the circle of topics and explains what can be done if things don't work as expected. You will learn how to find the most common issues and understand how problems can be fixed.

Chapter 13, *Migrating to PostgreSQL*, teaches you how to move your databases to PostgreSQL efficiently and quickly. It covers the most common database systems people will migrate from.

To get the most out of this book

This book has been written for a broad audience. However, some basic knowledge of SQL is necessary to follow along and make full use of the examples presented. In general, it is also a good idea to familiarize yourself with basic Unix commands as most of the book has been produced on Linux and macOS.

Software/hardware covered in the book	**Operating system requirements**
pgAdmin4	Windows, macOS, or Linux
PostgreSQL 17	
SQL Shell (psql)	

Note

Some parts of chapters, that is, 8, 9, 10, 11, 12, and 13 are mostly dedicated to Unix/Linux and macOS users, and the rest run fine on Windows.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The community has removed this feature and introduced a new variable called `transaction_timeout`, which can be set per session."

A block of code is set as follows:

```
CREATE OR REPLACE FUNCTION on_login_proc()
RETURNS event_trigger AS
$$
BEGIN
    INSERT INTO user_lo (w) VALUES (SESSION_USER);
    RAISE NOTICE 'You are welcome!';
END;
$$ LANGUAGE plpgsql;
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
test=# SHOW event_triggers;
 event_triggers
```

Any command-line input or output is written as follows:

```
test=# CREATE TABLE t_data (
    id    int,
    data  text
);
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "A new process called **summarizer** was added to PostgreSQL."

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

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 `copyright@packt.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 `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Mastering PostgreSQL 17 – Sixth Edition*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1 What is New in PostgreSQL 17

PostgreSQL has come a long way since its first release in 1986. Today, it's one of the most widely used open source databases in the world. In this chapter, you will be introduced to all the most important and shiny features of PostgreSQL 17. Of course, the list of new stuff is almost infinite; therefore, this chapter will focus on those things that are expected to be most relevant to the majority of users.

We will cover the following topics in this chapter:

- Understanding DBA and administration features
- Using SQL and developer features
- Making use of new replication and backup add-ons
- Considering breaking changes in PostgreSQL 17

By the end of this chapter, you will know all about these shiny new features and you will understand how to use the new functionality in more detail.

Understanding DBA and administration features

In every release, a comprehensive set of features is added to make the lives of **database administrators** (**DBAs**) easier and more effective. The same is true for PostgreSQL. So, let us take a look and dive into the new features of PostgreSQL 17.

Terminating long transactions

PostgreSQL supports various features to limit the duration of statements, the maximum time a query will wait for locks, and a lot more. However, there is one feature that has been requested by customers for some time and has finally made it into the next release of PostgreSQL – the ability to limit the duration of a transaction. In the past, PostgreSQL supported an instance-wide configuration variable called `old_snapshot_threshold`. The idea of this variable was to limit the maximum length of a transaction to avoid table bloat as well as some other issues that are dangerous for the server. However, the `old_snapshot_threshold` variable could only be set per instance and not in a more fine-grained way. Thus, the community has removed this feature and introduced a new variable called `transaction_timeout`, which can be set per session.

The default value of this new setting is "unlimited" (`0`):

```
test=# SHOW transaction_timeout ;
 transaction_timeout
---------------------
 0
(1 row)
```

However, if you want to limit the duration of your transaction, you can simply set a value inside your session. The following command sets the configuration variable to 5 seconds (`5000` milliseconds):

```
test=# SET transaction_timeout TO 5000;
SET
test=# BEGIN;
BEGIN
test=*# SELECT now();
             now
------------------------------
 2024-06-21 19:37:35.81715+00
(1 row)

test=*# SELECT now();
             now
------------------------------
 2024-06-21 19:37:35.81715+00
(1 row)

test=*# SELECT now();
FATAL:  terminating connection due to transaction timeout
server closed the connection unexpectedly
  This probably means the server terminated abnormally
  before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
test=#
```

After 5 seconds, our transaction will terminate and PostgreSQL closes the connection entirely. It is the easiest way to eliminate unnecessarily long transactions and prevent table bloat.

However, PostgreSQL 17 has more to offer.

Improved event triggers

Event triggers are an important feature and were introduced in PostgreSQL many years ago. What is the main idea behind an event trigger? Imagine somebody changes your data structure by creating a table, an index, or some other kind of object. An event trigger allows us to react to those events and execute code as needed.

In PostgreSQL, some functionality has been added. First of all, we now have a configuration variable that looks as follows:

```
test=# SHOW event_triggers;
 event_triggers
----------------
 on
(1 row)
```

When this one is enabled, event triggers will fire for all applicable statements. The important part here is that only superusers can change this value to a different setting.

In PostgreSQL, there is also the possibility to create an event trigger on `REINDEX`. However, this is not as critical as the next new feature that has to be discussed – the ability to trigger `LOGIN`. Now, what is a `LOGIN` trigger? It basically calls a function (or fires) when a new connection to the database is established. Needless to say, this is an incredibly powerful footgun and can cause serious issues because many things can go wrong.

But before we discuss running a trigger during `LOGIN`, it makes sense to take a look at a simple example and understand how things work in general. The most basic example is to write a trigger that is supposed to track login attempts in a table. To do that, we can create a simple table:

```
login_trigger=# CREATE TABLE user_logins (
    id serial,
    who text
);
CREATE TABLE
```

In PostgreSQL, a trigger will always launch a function. Therefore, the first step is to come up with the function we want to run:

```
CREATE FUNCTION on_login_proc()
RETURNS event_trigger AS
$$
BEGIN
    INSERT INTO user_logins (who)
    VALUES (SESSION_USER);
    RAISE NOTICE 'You are welcome!';
END;
$$ LANGUAGE plpgsql;
```

What we see here is that the return value of this function is `event_trigger` – it is a special data type specifically designed for this purpose. The rest is plain and simple PL/pgSQL code that does not require a return value.

Finally, we can define the trigger itself:

```
CREATE EVENT TRIGGER on_login_event
ON ddl_command_start
EXECUTE FUNCTION on_login_proc();
```

Note that the event we are interested in is `login`. The rest is like a normal event trigger that calls a function of our choice. In the next step, we can already enable the trigger:

```
login_trigger=# ALTER EVENT TRIGGER on_login_trigger ENABLE ALWAYS;
ALTER EVENT TRIGGER
```

Congratulations, the trigger has been deployed successfully. Let us log out and reconnect to the database. After we have reestablished the new connection, we can already see the content of our audit table:

```
login_trigger=# SELECT * FROM user_logins;
 id | who
----+-----
  1 | hs
(1 row)
```

This looks pretty successful but why did we just call this type of trigger a footgun? Let us modify the function and see what happens:

```
CREATE OR REPLACE FUNCTION on_login_proc()
RETURNS event_trigger AS
$$
BEGIN
    INSERT INTO user_lo (w) VALUES (SESSION_USER);
    RAISE NOTICE 'You are welcome!';
END;
$$ LANGUAGE plpgsql;
```

The function is essentially buggy. The consequences are nothing short of a total failure:

```
linux$ psql login_trigger
psql: error: connection to server
  on socket "/tmp/.s.PGSQL.5432"
  failed: FATAL:  relation "user_lo" does not exist
LINE 1: INSERT INTO user_lo (w) VALUES (SESSION_USER)

QUERY:  INSERT INTO user_lo (w) VALUES (SESSION_USER)
CONTEXT:  PL/pgSQL function on_login_proc() line 3 at SQL statement
```

The entire authentication process will fail. That is important because small mistakes can lead to large-scale outages. It is therefore highly recommended to think twice before deploying event triggers to handle login attempts. What is important to understand is that PostgreSQL does exactly what it has been designed to do – it just does so in a sensitive area that can cause a lot of trouble.

The problem is that when you have installed a broken event trigger and you want to get rid of it, you have a hard time ahead. First of all, you have to shut down the database and then you have to start it in single-user mode (`https://www.postgresql.org/docs/17/app-postgres.html`). In single-user mode, you can then drop the event trigger because, in single-user mode, event triggers are actually disabled – you can therefore log in without those functions being fired.

Inspecting wait events in PostgreSQL

PostgreSQL provides a shiny new system view called `pg_wait_events`. For many years, wait events have been an integral feature of PostgreSQL, and allowed us to monitor and inspect all kinds of performance problems. However, in real life, DBAs often had to switch between the database and the documentation to figure out which type of wait events actually do exist.

`pg_wait_events` puts an end to this type of problem and provides an easy way to understand which events are there and what they mean. Here is an example:

```
test=# \x
Expanded display is on.

test=# SELECT *
FROM pg_wait_events
WHERE name = 'DataFileFlush';
-[ RECORD1 ]-----------------------------------------
type        | IO
name        | DataFileFlush
description | Waiting for a relation data file to
              reach durable storage
```

What we can see here is that `DataFileFlush` means that we are waiting for the operating system to write the data to your physical storage device.

The beauty is that this new view provides a comprehensive overview that might surprise many people:

```
test=# SELECT  type, count(*)
FROM    pg_wait_events
GROUP BY ROLLUP (1)
ORDER BY 1;
   type    | count
-----------+-------
 Activity  |    16
 BufferPin |     1
 Client    |     9
 Extension |     1
 IO        |    77
 IPC       |    57
 LWLock    |    81
 Lock      |    12
 Timeout   |    10
           |   264
(10 rows)
```

Yes, this is true: PostgreSQL knows a grand total of 264 different types of events, which is a huge number.

Digging into checkpoints and background writing

The background writer and the checkpoint process have been around for many years. In the latest release of PostgreSQL, the system statistics related to those processes have been changed. First of all, a couple of fields have been removed from the `pg_stat_bgwriter` system view:

```
test=# \d pg_stat_bgwriter
     View "pg_catalog.pg_stat_bgwriter"
      Column      |           Type   ...
------------------+-------------------------
 buffers_clean    | bigint
 maxwritten_clean | bigint
 buffers_alloc    | bigint
 stats_reset      | timestamp with time zone
```

The view is way more compact now because a great deal of this information has been moved to a new system view that is defined as follows:

```
test=# \d pg_stat_checkpointer
                     View "pg_catalog.pg_stat_checkpointer"
       Column        |           Type ...
---------------------+---------------------------
```

```
  num_timed             | bigint
  num_requested         | bigint
  restartpoints_timed | bigint
  restartpoints_req     | bigint
  restartpoints_done  | bigint
  write_time            | double precision
  sync_time             | double precision
  buffers_written       | bigint
  stats_reset           | timestamp with time zone
```

The `pg_stat_checkpointer` view contains most of the information previously found in `pg_stat_bgwriter`. Therefore, it is necessary to adjust your monitoring queries to reflect those changes.

Improving pg_stat_statements

The `pg_stat_statements` module is an extension for PostgreSQL that is shipped as part of the `contrib` package. To me, it has always been the gold standard for performance analysis; therefore, I am happy to see even more changes made to improve the data provided by this view. There are various interesting ones worth mentioning:

- `CALL` statements now support parameters as placeholders
- Allows placeholders for savepoint and 2PC-related commands
- Tracks `DEALLOCATE` statements
- Adds support for local block I/O statistics
- Adds more details to JIT statistics
- Adds an optional argument to `pg_stat_statements_reset()`

Overall, these improvements will make `pg_stat_statements` more compact and thus make it easier to find relevant information.

Adding permissions for maintenance tasks

In the past, maintenance was often done by the superuser. The goal is to avoid this. Therefore, more and more permissions and roles have been added over the years to reduce the need for the superuser.

PostgreSQL 17 has added the `MAINTAIN` permission to a couple of commands, which allows us to execute various important tasks such as the following:

- `VACUUM` and `ANALYZE`
- `CLUSTER`

- `REINDEX`
- `REFRESH MATERIALIZED VIEW`
- `LOCK TABLE`

The feature works as follows:

```
test=# CREATE USER joe;
CREATE ROLE
test=# GRANT MAINTAIN ON …
```

The tab completion will reveal the full power of this new feature. The number of options you have are quite numerous:

- `ALL FUNCTIONS IN SCHEMA DATABASE`
- `TABLE`
- `ALL PROCEDURES IN SCHEMA`
- `DOMAIN`
- `LANGUAGE`
- `ROUTINE`
- `TABLESPACE`
- `ALL ROUTINES IN SCHEMA`
- `FOREIGN DATA WRAPPER`
- `LARGE OBJECT`
- `SCHEMA`
- `TYPE`
- `ALL SEQUENCES IN SCHEMA`
- `FOREIGN SERVER`
- `PARAMETER`
- `SEQUENCE`
- `ALL TABLES IN SCHEMA`
- `FUNCTION`
- `PROCEDURE`

After dealing with DBA-related functionalities, we can now turn our attention to some more developer-oriented functionalities.

Using SQL and developer features

In this section, we will discuss some of the most desired developer and SQL features that have made it into PostgreSQL 17.

Teaching COPY error handling

Let us start with my personal favorite: `COPY` is finally able to handle errors in a reasonably good way. Many people were frustrated by the error-handling behavior. Standard PostgreSQL will stop `COPY` when it hits an error. It is good to see that this vital functionality has made it into the official release of PostgreSQL.

The main question arising is: how can we make use of this feature? Here is a simple command to create a sample table:

```
test=# CREATE TABLE t_data (
    id    int,
    data  text
);
CREATE TABLE
```

The goal is to import the following dataset into the table and make sure those errors are handled properly:

```
1  hans
2  paul
abc  joe
4  jane
def  james
5  laura
```

What we see here is that the data is definitely wrong. The following listing proves this beyond doubt:

```
test=# COPY t_data FROM '/tmp/file.txt';
ERROR:  invalid input syntax for type integer: "abc"
CONTEXT:  COPY t_data, line 3, column id: "abc"
```

Fortunately, we can handle this kind of problem in PostgreSQL 17 and simply ignore the error:

```
test=# COPY t_data FROM '/tmp/file.txt'
WITH (ON_ERROR 'ignore');
NOTICE:  2 rows were skipped due to data type incompatibility
COPY 4
test=# SELECT * FROM t_data ;
 id | data
----+-------
```

```
   1 | hans
   2 | paul
   4 | jane
   5 | laura
(4 rows)
```

PostgreSQL will import the data and simply skip over invalid data. A `NOTICE` label will indicate how many rows have been skipped. As of version 17, two types of `ON_ERROR` settings are supported: `stop` and `ignore`. In the future, it is likely that more options will be available.

Splitting and merging partitions

In the previous years, there has not been a single version of PostgreSQL that has not provided relevant improvements to partitioning as a whole. The same holds true for the new release. This time, the development team has been working on splitting and merging partitions, which has been a frequent requirement over the years.

The following listing shows how a simple table including a partition can be created:

```
CREATE TABLE t_timeseries (
    id serial,
    d date,
    payload text
) PARTITION BY RANGE (d);

CREATE TABLE t_timeseries_2024
PARTITION OF t_timeseries
FOR VALUES FROM ('2024-01-01')
TO ('2025-01-01');
```

Here is a typical example one would encounter in real life. We have some kind of time series and we are using range partitions to split the data into smaller chunks for various reasons (faster cleanup, scalability, and so on). However, tables are often too large and we have to break them up into smaller chunks. This is when PostgreSQL can do some of the heavy lifting for us:

```
ALTER TABLE t_timeseries
   SPLIT PARTITION t_timeseries_2024
   INTO (
     PARTITION t_timeseries_2024_h1
    FOR VALUES FROM ('2024-01-01') TO ('2024-07-01'),
  PARTITION t_timeseries_2024_h2
    FOR VALUES FROM ('2024-07-01') TO ('2025-01-01')
);
```

What we do here is take our partition and split it into two new chunks that contain roughly half of the data. Note that this is a single command that takes care of this operation.

While splitting partitions into various pieces might be by far the most common new operation, it is also possible to reverse this decision and unify various partitions into a single entity. The way to do that is by using the `ALTER TABLE … MERGE PARTITIONS …` command, which is equally as easy to use as the `SPLIT` command that we have observed and tested before:

```
ALTER TABLE t_timeseries
MERGE PARTITIONS (
    t_timeseries_2024_h1,
    t_timeseries_2024_h2
)
INTO t_timeseries_2024;
```

All we have to do here is to tell PostgreSQL which partitions are supposed to form the new entity and let the database engine do its magic.

Tuning numbers into binary and octal values

One of the lesser-known features that made it into PostgreSQL is the ability to convert numbers to a binary and, respectively, octal representation. Two overloaded functions have been added – `to_bin` and `to_oct`:

```
test=# \df *to_bin*
   List of functions
   Schema   |  Name  | Result data type | Argument data types | Type
------------+--------+------------------+---------------------+------
 pg_catalog | to_bin | text             | bigint              | func
 pg_catalog | to_bin | text             | integer             | func
(2 rows)

test=# \df *to_oct*
   List of functions
   Schema   |  Name  | Result data type | Argument data types | Type
------------+--------+------------------+---------------------+------
 pg_catalog | to_oct | text             | bigint              | func
 pg_catalog | to_oct | text             | integer             | func
(2 rows)
```

Both functions can be called with 32- or 64-bit integer values. The following listing shows an example of those functions in action:

```
test=# SELECT to_bin(4711), to_oct(4711);
    to_bin     | to_oct
```

```
---------------+--------
 1001001100111 | 11147
(1 row)
```

Improving MERGE even more

MERGE has been around for various releases. In SQL, the MERGE command is used to merge data from two tables into one table. This command is useful when you need to update or insert rows based on a common column between the two tables.

The new release of PostgreSQL has also introduced another feature, namely, WHEN NOT MATCHED BY SOURCE THEN. This additional syntax allows us to define the behavior even better and adds some flexibility.

Here is how it works:

```
CREATE TABLE t_demo (
    a int PRIMARY KEY,
    b int
);

INSERT INTO t_demo
VALUES (1, 4711),
       (2, 5822),
       (3, 6933);

CREATE TABLE t_source (
    a int PRIMARY KEY,
    b int
);

INSERT INTO t_source
VALUES (2, 6822),
       (3, 6933),
       (4, 1252);

MERGE INTO t_demo AS t1
USING t_source AS t2
ON t1.a = t2.a
WHEN MATCHED THEN
    UPDATE SET b = t1.b * 100
WHEN NOT MATCHED THEN
```

```
    INSERT (a, b) VALUES (t2.a, t2.b)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE
RETURNING t1.*, t2.*;
```

This `MERGE` statement will return the following data:

```
 a |   b    | a |  b
---+--------+---+------
 1 |   4711 |   |
 2 | 582200 | 2 | 6822
 3 | 693300 | 3 | 6933
 4 |   1252 | 4 | 1252
(4 rows)
```

`RETURNING *` can be really useful to debug the statement as a whole. The same is true in my example: `a = 1` is available in the original table but not in the source table and the row is therefore deleted. In the case of `a = 2` and `a = 3`, we got a full match and, therefore, the `UPDATE` statement will execute. `a = 4` is only present in the `t_source` table and is therefore inserted into the `t_demo` table.

The following table shows what we can expect to find after the `MERGE` operation:

```
 a |   b
---+--------
 2 | 582200
 3 | 693300
 4 |   1252
(3 rows)
```

As you can see, all three cases defined in the `MERGE` statement have been executed successfully. The question is: which row was touched by which rule? We can modify the `RETURNING` clause a bit:

```
RETURNING merge_action(), t1.*, t2.*
```

In this case, PostgreSQL will provide us with even more information, as we can see in the following listing:

```
 merge_action | a |   b    | a |  b
--------------+---+--------+---+------
 DELETE       | 1 |   4711 |   |
 UPDATE       | 2 | 582200 | 2 | 6822
 UPDATE       | 3 | 693300 | 3 | 6933
 INSERT       | 4 |   1252 | 4 | 1252
(4 rows)
```

Additional JSON functionality

The number of JSON-related functionalities has skyrocketed over the years. The same is true for version 17, which provides many more features that will make JSON easier to use and more powerful overall. The first thing that caught my attention was the fact that more standard compliant functions have been added – namely, `JSON_EXISTS()`, `JSON_QUERY()`, and `JSON_VALUE()`.

What is also noteworthy is `JSON_TABLE`, which allows us to turn a JSON document into a tabular format in one go. This is pretty similar to what `XMLTABLE` does.

The syntax might look as follows:

```
SELECT jt.*
FROM customers,
     JSON_TABLE (
         js, '$.favorites[*]' COLUMNS (
             id FOR ORDINALITY,
             country text PATH '$.country',
             branch text PATH '$.industry[*].branch' WITH WRAPPER,
             ceo text PATH '$.company[*].ceo' WITH WRAPPER
         )
     ) AS jt;
```

What this does is to address various elements in the JSON document and return it in a format we can actually and safely read.

Creating BRIN indexes in parallel

Technically, this is not a developer feature, but given the fact that performance topics are often hard to categorize, I decided to include this here. BRIN indexes are often used in data warehouses to quickly filter data without carrying the overhead of full-blown B-tree indexes. Creating B-trees has long been possible using more than one CPU. However, in PostgreSQL 17, it is now possible to create BRIN indexes in parallel, which can greatly speed up the process

Making use of new replication and backup add-ons

As you've worked your way through some of the new developer-related features, you're now ready to address the new version's powerful set of advanced features related to database administration. In this section, we'll delve into the more complex world of database management, exploring topics that are new to PostgreSQL 17.

More powerful pg_dump, again

`pg_dump` is the single most well-known tool to run a basic backup in PostgreSQL. It is a command-line utility that comes with PostgreSQL, used for backing up a PostgreSQL database or extracting its schema and data in a format suitable for loading into another PostgreSQL database. The main question is: after 38 years of development, what might have been added to this tool that is not already there? Well, the answer is that you can now define a file that configures what you want to dump and what you want to ignore. By adding the `--filter` option, we can feed a file containing all our desired rules.

Handling incremental base backups

Talking about backups in general, `pg_basebackup` has also been extended. PostgreSQL 17 supports the idea of incremental base backups. Why is that important? Often, we might want to use a simple backup policy such as "Take a base backup every night and keep it for 7 days." The problem is that if your database is large (say, 50 TB) but static (virtually no changes), you will waste a lot of space just to store the backup, which can, of course, lead to serious cost considerations. Incremental base backup addresses this issue:

```
summarize_wal = on
wal_summary_keep_time = '7d'
```

A new process called **summarizer** was added to PostgreSQL. It will keep track of all those blocks that have been changed and help `pg_basebackup` to only copy those blocks that have indeed been touched, which reduces the amount of space needed for the backups to drop significantly.

Here is how it works:

```
pg_basebackup -h source_server.com \
  -D /data/full --checkpoint=fast
...
pg_basebackup -h source_server.com \
  --checkpoint=fast \
  --incremental=/data/full/backup_manifest \
      -D /backup/incremental
```

The secret to success is the backup manifest that is needed to run the incremental backup. It contains all the necessary information to tell the tooling what has to be done.

After running those two commands, we have a full backup as well as an incremental one. The question now is: how can we combine those things together and turn them into something usable? The following command shows how this works:

```
$ pg_combinebackup --help
pg_combinebackup reconstructs full backups from incrementals.
```

```
Usage:
  pg_combinebackup [OPTION]... DIRECTORY...

Options:
  -d, --debug               generate lots of debugging output
  -n, --dry-run             do not actually do anything
  -N, --no-sync             do not wait for changes to be written
                            safely to disk
  -o, --output              output directory
  -T, --tablespace-mapping=OLDDIR=NEWDIR
                            relocate tablespace in OLDDIR to NEWDIR
      --clone               clone (reflink) instead of copying files
      --copy-file-range     copy using copy_file_range() syscall
      --manifest-checksums=SHA{224,256,384,512}|CRC32C|NONE
                            use algorithm for manifest checksums
      --no-manifest         suppress generation of backup manifest
      --sync-method=METHOD  set method for syncing files to disk
  -V, --version             output version information, then exit
  -?, --help                show this help, then exit
```

`pg_combinebackup` does exactly what we want. It creates the desired set of files that are then needed for recovery. Given our example, we could use the following instruction to combine our full backup with our incremental backup:

```
pg_combinebackup -o /data/combined \
  /data/full \
  /backup/incremental
```

What is noteworthy here is that this process works for one base backup and exactly one incremental backup. However, in real life, we might have to apply a set of incremental backups to reach the desired state. In this case, we can simply list all those incremental ones one after the other, as shown in the next listing:

```
pg_combinebackup -o /data/combined \
  /data/full \
  /backup/incremental \
  /backup/incremental2 \
  /backup/incremental3
```

Simply list all the incremental backups to produce the desired state.

Logical replication upgraded

In PostgreSQL, there are two types of replication: physical (binary) and logical (text) replication. While binary replication is ideal for all kinds of backup, logical replication has become more and more widespread in heterogeneous environments to achieve cross-cloud portability.

The trouble is that publications and subscriptions (the backbone of logical replication) were lost during `pg_upgrades` prior to PostgreSQL 17. This has now changed and has significantly eased the burden.

Adding pg_createsubscriber

In the new release, we can all enjoy a new command-line tool called `pg_createsubscriber`. What is the purpose of this new tool? When people decide to use logical replication, the initial sync phase can take quite a while – especially when the database instance is large. `pg_createsubscriber` has been designed to help solve this problem. It converts a physical standby (binary replication) and turns it into a logical standby by wiring all the publications, subscriptions, and so on for you. For each database, a replication set will be created and automatically configured. The command has to be executed on the target system.

Considering breaking changes in PostgreSQL 17

PostgreSQL tries to keep the user interface as constant as possible. However, once in a while, breaking changes are necessary. This is, of course, also true for the current release.

Let us take a look at some of those changes. The first thing that has happened is the fact that support for AIX has dropped. This somehow makes sense because nobody here at CYBERTEC nor any other fellow PostgreSQL consultant I know has seen deployment on AIX in years.

The next thing is that `--disable-thread-safety` and MSVC builds have been dropped. All those things won't hurt users at all.

What is more important is that some toolings have been removed from the `contrib` section. The one module I am referring to is `adminpack`, which has not been widely used anyway. The same is true for `snapshot too old`, which has been replaced with a way better implementation (`transaction_timeout`).

Finally, `search_path` is now fully secured during maintenance operations, which means that maintenance scripts should use fully qualified object names.

Summary

The new release has countless new features and it is close to impossible to mention them all. During the development cycle, well over 2,000 commits have happened and thousands of things have been improved.

Some of the key features, such as fault-tolerant `COPY` and improved partitioning, have long been awaited and finally made it into the core. Things such as incremental base backups will significantly reduce the cost of large-scale PostgreSQL deployments. And other features simply lead to a way better user experience.

Therefore, relax, lean back, and enjoy the brand-new release of PostgreSQL, which will be covered in this book in great detail.

In *Chapter 2*, *Understanding Transactions and Locking*, we will discuss important concepts such as transactions and locking, which form a core component of every relational database system.

2 Understanding Transactions and Locking

Now that we've been introduced to PostgreSQL 17 and all the new shiny features it brings to the table, we want to focus our attention on the next important topic. **Locking** is a vital concept for any kind of database. It is not enough to understand how locking works just to write proper or better applications – it is also essential from a performance point of view and, therefore, proper locking behavior will directly translate to an excellent user experience. Without handling locks properly, your applications might not only be slow – they might also behave in very unexpected ways (for example, timeouts, unpredictable results, and a lot more). In my opinion, locking is the key to performance. Why is that the case? There is no slower form of execution than waiting on something. Even more CPUs will not speed up waiting. Therefore, understanding locking and transactions is important for administrators and developers alike.

In this chapter, you will learn about the following topics:

- Working with PostgreSQL transactions
- Understanding basic locking
- Making use of `FOR SHARE` and `FOR UPDATE`
- Understanding transaction isolation levels
- Observing deadlocks and similar issues
- Utilizing advisory locks
- Optimizing storage and managing cleanup

By the end of this chapter, you will be able to understand and utilize PostgreSQL transactions in the most efficient way possible. You will see that many applications can benefit from improved performance.

Working with PostgreSQL transactions

PostgreSQL provides you with highly advanced transaction machinery that offers countless features to developers and administrators alike. In this section, we will look at the basic concept of transactions. The first important thing to know is that, in PostgreSQL, everything is a transaction. If you send a simple query to the server, it is already a transaction. Here is an example:

```
test=# SELECT now(), now();
              now              |              now
-------------------------------+-------------------------------
 2024-05-24 12:59:33.594603+02 | 2024-05-24 12:59:33.594603+02
(1 row)
```

In this case, the SELECT statement will be a separate transaction. If the same command is executed again, different timestamps will be returned.

Tip

Keep in mind that the now() function will return the transaction time. The SELECT statement will, therefore, always return two identical timestamps. If you want the *real time*, consider using clock_timestamp() instead of now().

If more than one statement has to be a part of the same transaction, the BEGIN statement must be used, as follows:

```
test=# \h BEGIN
Command: BEGIN
Description: start a transaction block
Syntax:
 BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
where transaction_mode is one of:
  ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED |
READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
URL: https://www.postgresql.org/docs/17/sql-begin.html
```

The BEGIN statement will ensure that more than one command is packed into a transaction. Here is how it works:

```
test=# BEGIN;
BEGIN
test=*# SELECT now();
              now
-------------------------------
```

```
 2024-05-24 13:00:39.864604+02
(1 row)

test=*# SELECT now();
              now
-------------------------------
 2024-05-24 13:00:39.864604+02
(1 row)

test=*# COMMIT;
COMMIT
```

The important point here is that both timestamps will be identical. As we mentioned earlier, we are talking about transaction time.

To end the transaction, `COMMIT` can be used:

```
test=# \h COMMIT
Command: COMMIT
Description: commit the current transaction
Syntax:
 COMMIT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
 URL: https://www.postgresql.org/docs/17/sql-commit.html
```

There are a few syntax elements here. You can just use `COMMIT`, `COMMIT WORK`, or `COMMIT TRANSACTION`. All three commands have the same meaning. If this is not enough, there's more – the `END` command is identical to `COMMIT` and can be used interchangeably:

```
test=# \h END
Command: END
Description: commit the current transaction
Syntax:
 END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
 URL: https://www.postgresql.org/docs/17/sql-end.html
```

As you can see, the `END` clause is the same as the `COMMIT` clause from a feature point of view.

`ROLLBACK` is the counterpart of `COMMIT`. Instead of successfully ending a transaction, it will simply stop the transaction without ever making things visible to other transactions, as shown in the following code:

```
test=# \h ROLLBACK
Command: ROLLBACK
Description: abort the current transaction
```

```
Syntax:
 ROLLBACK [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
 URL: https://www.postgresql.org/docs/17/sql-rollback.html
```

Some applications use `ABORT` instead of `ROLLBACK` (those two commands are interchangeable in PostgreSQL). The meaning is the same. What is really useful in PostgreSQL is the idea of transaction chains. `COMMIT AND CHAIN` will help you to achieve exactly that:

```
test=# SHOW transaction_read_only;
 transaction_read_only
-----------------------
 Off
 (1 row)
test=# BEGIN TRANSACTION READ ONLY ;
 BEGIN
test=*# SELECT 1;
 ?column?
----------
        1
 (1 row)
test=*# COMMIT AND CHAIN;
 COMMIT
test=*# SHOW transaction_read_only;
 transaction_read_only
-----------------------
 On
 (1 row)
test=*# SELECT 1;
 ?column?
----------
        1
 (1 row)
test=*# COMMIT AND NO CHAIN;
 COMMIT
test=# SHOW transaction_read_only;
 transaction_read_only
-----------------------
 Off
(1 row)
test=# COMMIT;
 WARNING: there is no transaction in progress
COMMIT
```

Let's go through this example step by step:

1. Display the content of the `transaction_read_only` setting. It is `Off` because, by default, we are in read/write mode.
2. Start a read-only transaction using `BEGIN`. This will automatically adjust the `transaction_read_only` variable.
3. Commit the transaction using `AND CHAIN`, and then PostgreSQL will automatically start a new transaction featuring the same properties as the previous transaction.

In our example, we will also be in read-only mode, just like the transaction before. There is no need to explicitly open a new transaction and set whatever values again, which can dramatically reduce the number of round trips between the application and the server. In the case of high latency systems, saving on commands can really make a difference. If a transaction is committed normally (`= NO CHAIN`), the read-only attribute of the transaction will be gone.

Handling errors inside a transaction

In this section, we will dig deeper into error handling and learn how to handle problems inside a database transaction. It is not always the case that transactions are correct from beginning to end. Things might just go wrong for whatever reason. However, in PostgreSQL, only error-free transactions can be committed. The following listing shows a failing transaction, which errors out due to a `division by zero` error:

```
test=# BEGIN;
 BEGIN
test=*# SELECT 1;
 ?column?
----------
        1
 (1 row)
test=*# SELECT 1 / 0;
 ERROR: division by zero
test=!# SELECT 1;
 ERROR: current transaction is aborted, commands ignored until end of
transaction block
test=!# SELECT 1;
 ERROR: current transaction is aborted, commands ignored until end of
transaction block
test=!# COMMIT;
 ROLLBACK
```

Note that `division by zero` did not work out.

Note

In any proper database, an instruction similar to this will instantly error out and make the statement fail.

It is important to point out that PostgreSQL will error out. After an error has occurred, no more instructions will be accepted, even if those instructions are semantically and syntactically correct. It is still possible to issue `COMMIT`. However, PostgreSQL will roll back the transaction because it is the only correct thing to be done at that point.

Making use of SAVEPOINT

In professional applications, it can be pretty hard to write reasonably long transactions without ever encountering a single error. To solve this problem, users can utilize something called `SAVEPOINT`, as follows:

```
test=# \h SAVEPOINT
Command:     SAVEPOINT
Description: define a new savepoint within the current transaction
Syntax:
SAVEPOINT savepoint_name

URL: https://www.postgresql.org/docs/17/sql-savepoint.html
```

As the name indicates, a savepoint is a safe place inside a transaction that the application can return to if things go terribly wrong. Here is an example:

```
test=# BEGIN;
BEGIN
test=*# SELECT 1;
 ?column?
----------
        1
 (1 row)
test=*# SAVEPOINT a;
SAVEPOINT
test=*# SELECT 2 / 0;
ERROR: division by zero
test=!# SELECT 2;
ERROR: current transaction is aborted, commands ignored until end of
transaction block
test=!# ROLLBACK TO SAVEPOINT a;
ROLLBACK
```

```
test=*# SELECT 3;
 ?column?
----------
        3
 (1 row)
test=*# COMMIT;
COMMIT
```

After the first `SELECT` clause, I decided to create a savepoint to make sure that the application can always return to this point inside the transaction. As you can see, the savepoint has a name, which is referred to later.

After returning to the savepoint called `a`, the transaction can proceed normally. The code has jumped back to before the error, so everything is fine.

The number of savepoints inside a transaction is practically unlimited. We have seen customers with over 250,000 savepoints in a single operation. PostgreSQL can easily handle this.

If you want to remove a savepoint from inside a transaction, there's the `RELEASE SAVEPOINT` command:

```
test=# \h RELEASE
Command: RELEASE SAVEPOINT
Description: destroy a previously defined savepoint
Syntax:
 RELEASE [ SAVEPOINT ] savepoint_name
URL: https://www.postgresql.org/docs/17/sql-release-savepoint.html
```

Many people ask what will happen if you try to reach a savepoint after a transaction has ended. The answer is that the life of a savepoint ends as soon as the transaction ends. In other words, there is no way to return to a certain point in time after the transactions have been completed.

Transactional DDLs

PostgreSQL has a very nice feature that is unfortunately not present in many commercial database systems. In PostgreSQL, it is possible to run DDLs (commands that change the data structure) inside a transaction block. In a typical commercial system, a DDL will implicitly commit the current transaction by default. This does not occur in PostgreSQL.

Apart from some minor exceptions (`DROP DATABASE`, `CREATE TABLESPACE`, `DROP TABLESPACE`, and so on), all DDLs in PostgreSQL are transactional, which is a huge advantage and a real benefit to end users.

Here is an example:

```
test=# BEGIN;
BEGIN
```

```
test=*# CREATE TABLE t_test (id int);
CREATE TABLE
test=*# ALTER TABLE t_test ALTER COLUMN id TYPE int8;
ALTER TABLE
test=*# \d t_test
              Table "public.t_test"
 Column |  Type  | Collation | Nullable | Default
--------+--------+-----------+----------+---------
 id     | bigint |           |          |
test=*# ROLLBACK;
ROLLBACK
test=# \d t_test
Did not find any relation named "t_test".
```

In this example, a table has been created and modified, and the entire transaction has been aborted. As you can see, there is no implicit `COMMIT` command or any other strange behavior. PostgreSQL simply acts as expected.

Transactional DDLs are especially important if you want to deploy software. Just imagine running a **content management system** (**CMS**). If a new version is released, you'll want to upgrade. Running the old version would still be okay; running the new version would also be okay, but you really don't want a mixture of old and new. Therefore, deploying an upgrade in a single transaction is highly beneficial, as it upgrades an atomic operation.

Note

To facilitate good software practices, we can include several separately coded modules from our source control system into a single deployment transaction.

After dealing with transaction and error handling in general, it is important to focus our attention more on locking and concurrency.

Understanding basic locking

In this section, you will learn about basic locking mechanisms. The goal is to understand how locking works in general and how to get simple applications right.

To show you how things work, we will create a simple table. For demonstrative purposes, I will add one row to the table using a simple `INSERT` command:

```
test=# CREATE TABLE  t_test (id int);
CREATE TABLE
```

```
test=# INSERT INTO t_test VALUES (0);
INSERT 0 1
```

The first important thing is that tables can be read concurrently. Many users reading the same data at the same time won't block each other. This allows PostgreSQL to handle thousands of users without any problems.

The question now is what happens if reads and writes occur at the same time? Here is an example. Let's assume that the table contains one row and `id = 0`:

Transaction 1	**Transaction 2**
`BEGIN;`	`BEGIN;`
`UPDATE t_test SET id = id + 1 RETURNING *;`	
User will see `1`	`SELECT * FROM t_test`
	User will see `0`
	`COMMIT;`
`COMMIT;`	

Table 2.1 – Transaction isolation

Two transactions are opened. The first one will change a row. However, this is not a problem, as the second transaction can proceed. It will return to the old row as it was before `UPDATE`. This behavior is called **multi-version concurrency control** (**MVCC**).

Note

A transaction will only see data if it has been committed by the write transaction before the initiation of the read transaction. One transaction cannot inspect the changes that have been made by another active connection. A transaction can see only those changes that have already been committed.

There is also a second important aspect – many commercial or open source databases are still unable to handle concurrent reads and writes. In PostgreSQL, this is absolutely not a problem – reads and writes can coexist.

Note

Write transactions won't block read transactions.

What will happen when two people try to change the same row at the same time? Can we actually end up with data loss or does PostgreSQL handle the operation properly? Let us give it a try:

Transaction 1	**Transaction 2**
`BEGIN;`	`BEGIN;`
`UPDATE t_test SET id = id + 1 RETURNING *;`	
It will return `2`	`UPDATE t_test SET id = id + 1 RETURNING *;`
	It will wait for transaction 1
`COMMIT;`	It will wait for transaction 1
	It will reread the row, find `2`, and return `3`
	`COMMIT;`

Table 2.2 – Handling concurrent updates

Suppose you want to count the number of hits on a website. If you run the preceding code, no hits will be lost because PostgreSQL guarantees that one `UPDATE` statement is performed after another.

Note

PostgreSQL will only lock rows affected by `UPDATE`. So, if you have 1,000 rows, you can theoretically run 1,000 concurrent changes on the same table.

It is also worth noting that you can always run concurrent reads. Our two writes will not block reads.

Avoiding typical mistakes and explicit locking

In my life as a professional PostgreSQL consultant (`https://www.cybertec-postgresql.com`), I have seen a couple of mistakes that are repeated frequently. If there are constants in life, these typical mistakes are definitely among them. The following listing shows one of the most common problems:

Transaction 1	**Transaction 2**
`BEGIN;`	`BEGIN;`
`SELECT max(id) FROM product;`	`SELECT max(id) FROM product;`
User will see `17`	User will see `17`
User decides to set to `18`	User decides to set to `18`