# **🚀 Zero-Downtime PostgreSQL 17 Table Optimization Using**pg\_repack

[](https://medium.com/@jramcloud1?source=post_page---byline--9e904d5d02b1---------------------------------------)

[Jeyaram Ayyalusamy](https://medium.com/@jramcloud1?source=post_page---byline--9e904d5d02b1---------------------------------------)

Following

14 min read

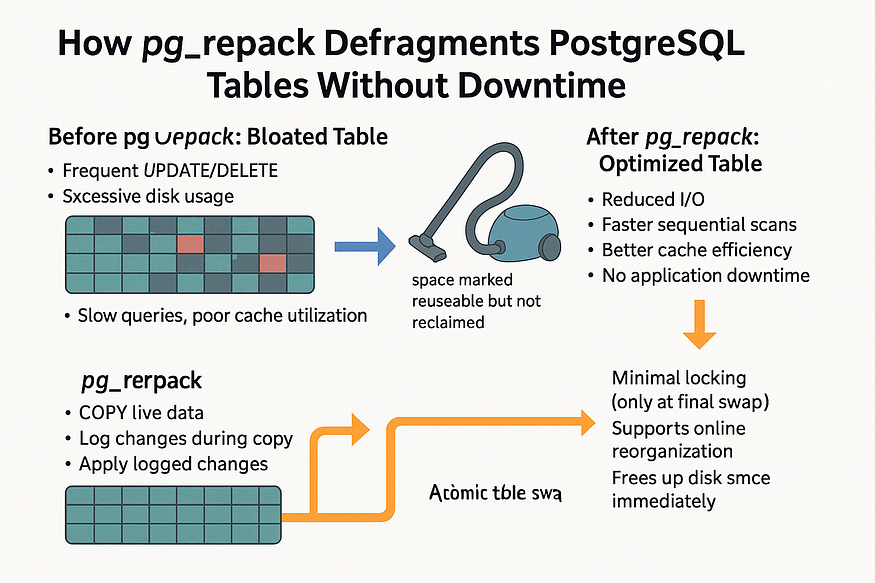
·

Aug 20, 2025

17

****Efficiently reclaim space and boost query performance without disrupting production workloads****

Press enter or click to view image in full size



PostgreSQL is a widely adopted open-source relational database known for its robustness, extensibility, and reliability in handling transactional workloads. However, when PostgreSQL is used in systems with ****high-frequency data modifications**** — such as frequent INSERT, UPDATE, and DELETE operations—it tends to accumulate ****table bloat**** over time.

In this article, we will explore:

* What causes table bloat in PostgreSQL
* Why VACUUM and autovacuum are insufficient for fully resolving it
* How pg\_repack works
* Step-by-step instructions for using pg\_repack on PostgreSQL 17
* Best practices for using pg\_repack in production environments

We will use a sample database named productiondb and a bloated table called customer\_orders to demonstrate the process.

## **What Is Table Bloat in PostgreSQL?**

PostgreSQL uses ****Multiversion Concurrency Control (MVCC)**** to manage concurrent transactions. This means that every time a row is updated or deleted, the old version of that row — referred to as a ****dead tuple**** — remains in the table until it is cleaned up.

Although dead tuples are invisible to active queries, they continue to ****occupy physical disk space****. As these dead tuples accumulate over time, they lead to ****table bloat****.

****Symptoms of table bloat include:****

* Increased disk space usage
* Slower query performance
* Degraded index efficiency
* Longer full table scans

## **Limitations of PostgreSQL’s VACUUM and autovacuum**

PostgreSQL provides two mechanisms to manage dead tuples:

1. ****VACUUM****: Marks dead tuples as reusable space. It allows PostgreSQL to reuse the space for new rows, but it does ****not release**** the space back to the operating system. The physical size of the table file remains the same.
2. ****VACUUM FULL****: Performs a full rewrite of the table to physically remove dead tuples and shrink the table on disk. However, this operation requires ****an exclusive lock**** on the table, making it unavailable during execution—a significant drawback for production databases.

## **Introducing**pg\_repack

To address the limitations of VACUUM FULL, PostgreSQL administrators can use ****pg\_repack****, a powerful extension designed to ****reorganize tables and indexes online****, without blocking concurrent access.

## **Key Features of**pg\_repack**:**

* Removes dead tuples by rewriting the table
* Operates ****without locking**** the table
* Rebuilds indexes to optimize storage and access
* Runs ****concurrently with active transactions****
* Suitable for production environments

pg\_repack works by creating a temporary copy of the table, copying live tuples, rebuilding indexes, and then replacing the original bloated table with the optimized version—****all without downtime****.

## **💡 What Is**pg\_repack**?**

pg\_repack is a PostgreSQL extension that enables the ****live reorganization of tables and indexes**** by removing dead tuples and physically shrinking storage files—****without requiring downtime****.

In a typical PostgreSQL environment, data updates and deletions create “dead tuples” which remain in the table until cleaned up. Although autovacuum can reclaim logical space internally, it does ****not reduce the physical size**** of the table or its indexes.

pg\_repack addresses this limitation through a background process that operates safely while the database remains fully online.

## **🔧 How**pg\_repack**Works**

Under the hood, pg\_repack executes the following steps:

1. ****Creates a shadow copy**** of the target table using an efficient parallel mechanism.
2. ****Streams live data**** into the new table structure while allowing ongoing reads and writes.
3. ****Rebuilds all associated indexes**** during the repack process to eliminate fragmentation.
4. ****Atomically swaps**** the original bloated table with the compacted one once processing is complete.

This entire process occurs with ****minimal locking****, enabling continuous application access and zero downtime.

✅ ****No exclusive locks. No blocked queries. No service interruptions.****

## **🎯 Why Use**pg\_repack**?**

Adopting pg\_repack in your PostgreSQL maintenance strategy delivers several practical benefits, particularly in large-scale transactional systems:

## **🧹 Reclaim Disk Space**

PostgreSQL’s MVCC model ensures data consistency but introduces ****dead rows**** from updates and deletes. These accumulate over time, leading to bloated tables that consume unnecessary disk space. pg\_repack physically removes these dead tuples, reducing table size and freeing up disk resources.

For example, a table originally 100 GB in size with 60% dead tuples may shrink to 40 GB after a successful pg\_repack run—resulting in substantial disk savings.

## **🚀 Improve Query Performance**

Bloated tables not only waste space but also degrade performance. Sequential scans take longer, and indexes become inefficient as they grow fragmented. By rebuilding tables and indexes, pg\_repack enhances:

* ****I/O efficiency****
* ****Query planning and execution times****
* ****Cache utilization****

The result: ****faster response times and reduced load**** on storage subsystems.

## **🕒 Perform Non-Disruptive Maintenance**

A key differentiator of pg\_repack is its ability to operate ****without locking the table****. Unlike VACUUM FULL, which halts access to the table during execution, pg\_repack runs ****safely during production hours****, even in multi-tenant or 24/7 environments.

This makes it ideal for businesses that cannot afford downtime or scheduled maintenance windows.

## **⚡ Rebuild and Optimize Indexes**

Indexes play a critical role in PostgreSQL performance, but they too suffer from bloat and fragmentation over time. pg\_repack reconstructs these indexes during the repack process, leading to:

* ****Faster lookups****
* ****Smaller index size****
* ****Improved query planning****

This is especially beneficial for OLTP applications with high-frequency reads and updates.

## **✅ Best Practices for Using**pg\_repack

To use pg\_repack effectively and safely in production, follow these best practices:

## **1. Target High-Churn Tables First**

Focus on tables that undergo frequent UPDATE and DELETE operations. These are typically more prone to bloat and yield the greatest benefits from repacking.

## **2. Monitor Disk Usage**

Before running pg\_repack, ensure there is enough available disk space. The tool temporarily creates a shadow copy of the table, which means the effective disk requirement could temporarily ****double****.

## **3. Use During Off-Peak Hours (When Possible)**

Although pg\_repack supports online execution, it still consumes I/O resources. Running it during low-traffic periods can reduce impact on performance-sensitive applications.

## **4. Automate with Scheduled Jobs**

Use job schedulers (e.g., cron, pg\_cron, or orchestration tools) to periodically run pg\_repack on known high-bloat tables. This keeps performance stable without requiring manual intervention.

Example:

pg\_repack -d dvdrental -t address--jobs=4

## **5. Use Index-Only Repacking If Needed**

If the goal is only to defragment indexes, you can selectively run:

pg\_repack -d dvdrental -t address -i

This is faster and less resource-intensive than repacking the full table.

## **6. Test in a Staging Environment First**

Always validate the pg\_repack process in a non-production environment. Confirm table size reduction, execution time, and system load before scheduling it in live systems.

## **🧪 Sample Scenario: Repacking a Bloated Orders Table**

Let’s assume your customer\_orders table in the productiondb database has accumulated bloat from months of transactional activity.

## **Installing**pg\_repack**on EC2 (RHEL 10)**

pg\_repack is a PostgreSQL extension and utility that allows you to rebuild tables and indexes online, without significant downtime. Since it isn’t included by default in PostgreSQL installations, you’ll need to install it manually.

On your EC2 instance, run the following command:

sudo dnf install -y pg\_repack\_17

## **📋 Example Output**

When executed on ****RHEL 10**** (with PostgreSQL repositories enabled), the system will download and install the required packages. You’ll see an output similar to this:

After installation, the terminal confirms:

[root@ip-172-31-25-61 ~]# sudo dnf install -y pg\_repack\_17  
Updating Subscription Management repositories.  
Unable to read consumer identity  
  
This system is not registered with an entitlement server. You can use "rhc" or "subscription-manager" to register.  
  
Last metadata expiration check: 0:27:44 ago on Wed Aug 20 20:17:39 2025.  
Dependencies resolved.  
===================================================================================================================================================================================================================================================  
 Package Architecture Version Repository Size  
===================================================================================================================================================================================================================================================  
Installing:  
 pg\_repack\_17 x86\_64 1.5.2-4PGDG.rhel10 pgdg17 67 k  
  
Transaction Summary  
===================================================================================================================================================================================================================================================  
Install 1 Package  
  
Total download size: 67 k  
Installed size: 160 k  
Downloading Packages:  
pg\_repack\_17-1.5.2-4PGDG.rhel10.x86\_64.rpm 3.2 MB/s | 67 kB 00:00  
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  
Total 2.9 MB/s | 67 kB 00:00  
Running transaction check  
Transaction check succeeded.  
Running transaction test  
Transaction test succeeded.  
Running transaction  
 Preparing : 1/1  
 Installing : pg\_repack\_17-1.5.2-4PGDG.rhel10.x86\_64 1/1  
 Running scriptlet: pg\_repack\_17-1.5.2-4PGDG.rhel10.x86\_64 1/1  
Installed products updated.  
  
Installed:  
 pg\_repack\_17-1.5.2-4PGDG.rhel10.x86\_64  
  
Complete!  
[root@ip-172-31-25-61 ~]#

## **✅ Verification**

To confirm the installation, check the version:

/usr/pgsql-17/bin/pg\_repack --version

Expected output:

[root@ip-172-31-25-61 ~]# /usr/pgsql-17/bin/pg\_repack --version  
pg\_repack 1.5.2  
[root@ip-172-31-25-61 ~]#

🔍 ****Note:****

* On RHEL 10, installing pg\_repack may also pull in additional PostgreSQL packages (e.g., PostgreSQL 13 components) as dependencies.
* Ensure you match the pg\_repack version with the PostgreSQL major version you’re using in your environment.

## **Enabling the**pg\_repack**Extension in PostgreSQL 17**

Once the pg\_repack package is installed on your EC2 instance, you must also enable it ****inside the target database**** before you can use it. This step creates the necessary objects (functions and catalog entries) that pg\_repack relies on.

Connect to your database (dvdrental in this case) as the postgres user:

sudo -iu postgres psql -d dvdrental

[root@ip-172-31-25-61 ~]# sudo -iu postgres psql -d dvdrental  
psql (17.6)  
Type "help" for help.  
  
dvdrental=#  
dvdrental=#

Inside the psql shell, run:

CREATE EXTENSION pg\_repack;

You should see the following output:

dvdrental=# CREATE EXTENSION pg\_repack;  
CREATE EXTENSION  
dvdrental=#

✅ At this point, the extension is active, and you can begin running pg\_repack commands on your tables or even across the entire database to rebuild them online with minimal locking.

## **Step 1: Identify Table Size**

SELECT  
 relname AS table,  
 pg\_size\_pretty(pg\_total\_relation\_size(relid)) AS size  
FROM  
 pg\_catalog.pg\_statio\_user\_tables  
WHERE  
 relname = 'actor';

dvdrental=#  
dvdrental=# SELECT  
 relname AS table,  
 pg\_size\_pretty(pg\_total\_relation\_size(relid)) AS size  
FROM  
 pg\_catalog.pg\_statio\_user\_tables  
WHERE  
 relname = 'actor';  
 table | size  
-------+-------  
 actor | 72 kB  
(1 row)  
  
dvdrental=#

## **Step 2: Run pg\_repack**

/usr/pgsql-17/bin/pg\_repack -d dvdrental -t actor

Within minutes, the bloated table is replaced with a compact, optimized version — without taking your application offline.

[postgres@ip-172-31-25-61 ~]$ /usr/pgsql-17/bin/pg\_repack -d dvdrental -t actor  
INFO: repacking table "public.actor"  
[postgres@ip-172-31-25-61 ~]$

## **Conclusion**

PostgreSQL’s MVCC model ensures excellent concurrency and consistency, but it also leads to table and index bloat in high-write systems. Traditional tools like VACUUM and VACUUM FULL offer partial or disruptive solutions.

With ****pg\_repack****, you gain a powerful alternative: ****online, non-blocking compaction**** of tables and indexes that enhances performance, reduces storage consumption, and requires ****no downtime****.

Whether you’re managing critical OLTP systems, customer-facing platforms, or internal analytics databases, integrating pg\_repack into your PostgreSQL maintenance toolkit is a proven best practice.

Start small, monitor improvements, and automate intelligently — ****your PostgreSQL performance depends on it.****

## **Creating Databases Before Enabling**pg\_repack

Before you can enable the pg\_repack extension, you may need to create the target databases where it will be used. In this example, we create three new databases — productiondb, reportsdb, and analyticsdb.

Connect to PostgreSQL as the postgres superuser and run:

postgres=# CREATE DATABASE productiondb;  
CREATE DATABASE  
  
postgres=# CREATE DATABASE reportsdb;  
CREATE DATABASE  
  
postgres=# CREATE DATABASE analyticsdb;  
CREATE DATABASE

✅ At this stage, the databases are created successfully. You can now connect to each of them individually and run:

CREATE EXTENSION pg\_repack;

This ensures that pg\_repack is installed and available in all the databases where you want to use it.

## **Create the Extension in Each Target Database**

psql -d productiondb -c "CREATE EXTENSION pg\_repack;"  
psql -d reportsdb -c "CREATE EXTENSION pg\_repack;"  
psql -d analyticsdb -c "CREATE EXTENSION pg\_repack;"

*The CREATE EXTENSION command must be executed within each database where pg\_repack operations are planned. It creates the required SQL functions and metadata to enable repack functionality.*

[postgres@ip-172-31-25-61 ~]$ psql -d productiondb -c "CREATE EXTENSION pg\_repack;"  
CREATE EXTENSION  
[postgres@ip-172-31-25-61 ~]$  
[postgres@ip-172-31-25-61 ~]$ psql -d reportsdb -c "CREATE EXTENSION pg\_repack;"  
CREATE EXTENSION  
[postgres@ip-172-31-25-61 ~]$  
[postgres@ip-172-31-25-61 ~]$ psql -d analyticsdb -c "CREATE EXTENSION pg\_repack;"  
CREATE EXTENSION  
[postgres@ip-172-31-25-61 ~]$

*If this step is skipped, pg\_repack will report: ERROR: extension "pg\_repack" is not installed in database.*

✅ Once these four steps are complete, pg\_repack is ready to be used on your PostgreSQL 17 instance.

## **📊 Section 2: Generating Test Data for Benchmarking**

## **Step 5: Create a Large Table for Load Testing**

psql -d productiondb \  
 -c "CREATE TABLE customer\_orders AS SELECT \* FROM generate\_series(1,1000000) AS id;"  
  
ALTER TABLE customer\_orders ADD CONSTRAINT customer\_orders\_pkey PRIMARY KEY (id);

*This uses PostgreSQL’s built-in generate\_series() function to create 1 million rows in a new table. This synthetic dataset allows you to:*

* Validate pg\_repack speed and behavior
* Simulate production-like loads
* Perform regression testing on repack scripts

*You can vary row count or add more columns to simulate different data models.*

[postgres@ip-172-31-25-61 ~]$ psql -d productiondb \  
 -c "CREATE TABLE customer\_orders AS SELECT \* FROM generate\_series(1,1000000) AS id;"  
SELECT 1000000  
[postgres@ip-172-31-25-61 ~]$

[postgres@ip-172-31-25-61 ~]$ psql -d reportsdb -c "CREATE TABLE customer\_orders AS SELECT \* FROM generate\_series(1,1000000) AS id;"  
SELECT 1000000  
[postgres@ip-172-31-25-61 ~]$  
[postgres@ip-172-31-25-61 ~]$ psql -d analyticsdb -c "CREATE TABLE customer\_orders AS SELECT \* FROM generate\_series(1,1000000) AS id;"  
SELECT 1000000  
[postgres@ip-172-31-25-61 ~]$

## **🧪 Section 3: Executing**pg\_repack**Safely and Effectively**

## **Step 6: Run a Dry Run to Preview the Repack Actions**

/usr/pgsql-17/bin/pg\_repack --dry-run -d productiondb  
/usr/pgsql-17/bin/pg\_repack --dry-run -d reportsdb  
/usr/pgsql-17/bin/pg\_repack --dry-run -d analyticsdb

*The --dry-run option allows you to simulate the repack process without modifying data. This step is strongly recommended before executing repack jobs in production. It ensures the configuration is correct, checks disk space, and lists the objects that would be affected.*

*This is especially useful for compliance and auditing before any live operation.*

[postgres@ip-172-31-25-61 ~]$ /usr/pgsql-17/bin/pg\_repack --dry-run -d productiondb  
INFO: Dry run enabled, not executing repack  
[postgres@ip-172-31-25-61 ~]$  
[postgres@ip-172-31-25-61 ~]$ /usr/pgsql-17/bin/pg\_repack --dry-run -d reportsdb  
INFO: Dry run enabled, not executing repack  
[postgres@ip-172-31-25-61 ~]$  
[postgres@ip-172-31-25-61 ~]$ /usr/pgsql-17/bin/pg\_repack --dry-run -d analyticsdb  
INFO: Dry run enabled, not executing repack  
[postgres@ip-172-31-25-61 ~]$

## **Step 7: Run the Repack Process on the Entire Database**

Here’s a polished section you can add to your Medium blog to explain this step and the warning:

## **Running**pg\_repack**on a Table Without a Primary Key**

After enabling the pg\_repack extension, you can begin repacking individual tables to reclaim space and improve performance. For example, on the productiondb database, we attempt to repack the customer\_orders table:

/usr/pgsql-17/bin/pg\_repack -d productiondb -t customer\_orders

The command returns the following warning:

WARNING: relation "public.customer\_orders" must have a primary key or not-null unique keys

## **🔍 Why This Warning Appears**

pg\_repack requires each table it processes to have either:

* A ****primary key****, or
* A ****NOT NULL unique index****

This is because pg\_repack works by creating a shadow copy of the table and tracking changes using the key during the rebuild process. Without a unique identifier, it cannot guarantee row-level consistency.

## **✅ How to Resolve**

To successfully use pg\_repack on such a table, you need to add a key:

1. ****Add a Primary Key**** (recommended for transactional tables):

ALTER TABLE customer\_orders  
ADD CONSTRAINT customer\_orders\_pkey PRIMARY KEY (order\_id);

productiondb=# ALTER TABLE customer\_orders ADD CONSTRAINT customer\_orders\_pkey PRIMARY KEY (id);  
ALTER TABLE  
productiondb=#

2. ****Or Add a Unique, NOT NULL Index**** (if no natural primary key exists):

CREATE UNIQUE INDEX customer\_orders\_unique\_idx  
ON customer\_orders (order\_number)  
WHERE order\_number IS NOT NULL;

Once the key or index is in place, re-run the command:

/usr/pgsql-17/bin/pg\_repack -d productiondb -t customer\_orders

✅ With the required key defined, pg\_repack can now rebuild the customer\_orders table online, minimizing downtime while reclaiming space and optimizing performance.

/usr/pgsql-17/bin/pg\_repack -d productiondb

*This command repacks all tables and indexes in the database. pg\_repack works online, meaning:*

* No exclusive locks are taken on the tables.
* Applications can continue reading and writing data.
* Tables are compacted by copying live rows to a shadow table, swapping, and then dropping the bloated copy.

[postgres@ip-172-31-25-61 ~]$ /usr/pgsql-17/bin/pg\_repack -d productiondb  
INFO: repacking table "public.customer\_orders"  
[postgres@ip-172-31-25-61 ~]$  
[postgres@ip-172-31-25-61 ~]$

*The repack process is ideal for periodic maintenance in OLTP systems where downtime is unacceptable.*

## **Step 8: Use Parallel Jobs to Reduce Runtime on Large Tables**

/usr/pgsql-17/bin/pg\_repack -d productiondb -j 2

*The -j flag enables parallel execution using multiple worker threads. This improves performance when dealing with:*

* Large partitioned tables
* Multiple bloated indexes
* High-volume transactional environments

*Adjust the number of jobs based on your system’s CPU and I/O bandwidth.*

[postgres@ip-172-31-25-61 ~]$ /usr/pgsql-17/bin/pg\_repack -d productiondb -j 2  
NOTICE: Setting up workers.conns  
INFO: repacking table "public.customer\_orders"  
[postgres@ip-172-31-25-61 ~]$

*Example: -j 4 uses four concurrent threads, reducing total execution time on large datasets.*

## **🎯 Section 4: Targeted Optimization of Tables and Indexes**

## **Step 9: Repack a Specific Table Only**

/usr/pgsql-17/bin/pg\_repack -t customer\_orders -d productiondb

*Use the -t flag when you want to repack only one table rather than the entire database. This minimizes I/O and runtime when only certain tables are impacted by bloat.*

[postgres@ip-172-31-25-61 ~]$ /usr/pgsql-17/bin/pg\_repack -t customer\_orders -d productiondb  
INFO: repacking table "public.customer\_orders"  
[postgres@ip-172-31-25-61 ~]$

*Ideal use cases:*

* Tables frequently updated or deleted (e.g., order history, temp records)
* Ad-hoc fixes before full maintenance cycles

## **Step 10: Rebuild a Specific Index Only**

To find the index, use the command below:

productiondb=# \di+  
 List of relations  
 Schema | Name | Type | Owner | Table | Persistence | Access method | Size | Description  
--------+----------------------+-------+----------+-----------------+-------------+---------------+-------+-------------  
 public | customer\_orders\_pkey | index | postgres | customer\_orders | permanent | btree | 21 MB |  
(1 row)  
  
productiondb=#

/usr/pgsql-17/bin/pg\_repack -i customer\_orders\_pkey -d productiondb

*The -i flag allows you to target individual indexes. Over time, indexes can become fragmented, impacting performance.*

[postgres@ip-172-31-25-61 ~]$ /usr/pgsql-17/bin/pg\_repack -i customer\_orders\_pkey -d productiondb  
INFO: repacking index "public.customer\_orders\_pkey"  
[postgres@ip-172-31-25-61 ~]$

*Use this option when:*

* A specific index shows poor performance
* Index size grows disproportionately to table size
* EXPLAIN plans reveal inefficient index scans

*This option saves resources when you don’t need to repack the entire table.*

## **🧹 Section 5: Removing the Extension When No Longer Needed**

## **Step 11: Uninstall**pg\_repack**from a Database**

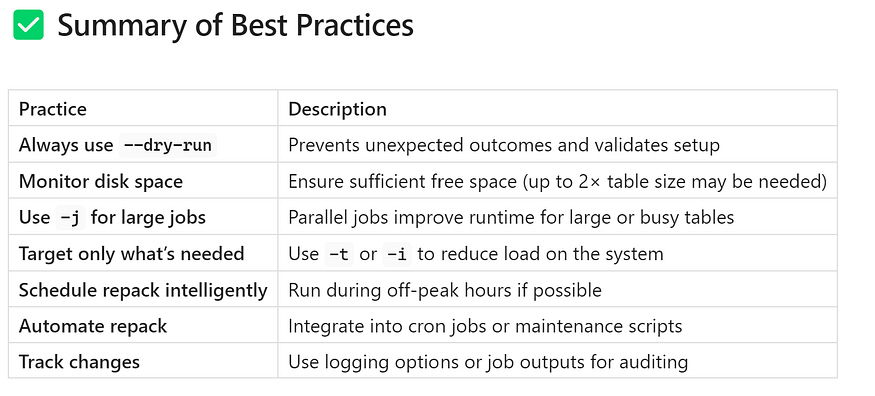
DROP EXTENSION pg\_repack;

*This SQL command removes the extension and its associated metadata from the database. It does****not****delete any data or affect repacked tables.*

[postgres@ip-172-31-25-61 ~]$ psql -d productiondb  
psql (17.6)  
Type "help" for help.  
  
productiondb=#  
productiondb=# DROP EXTENSION pg\_repack;  
DROP EXTENSION  
productiondb=#

*Use this step during cleanup, decommissioning, or if switching to other maintenance strategies.*

Press enter or click to view image in full size



By following these detailed steps and best practices, you can maintain a healthy, performant PostgreSQL 17 environment on Ubuntu — all without compromising availability.

## **🔍 Why**pg\_repack**Is Critical for Production PostgreSQL Environments**

Maintaining PostgreSQL performance in production environments becomes increasingly challenging as tables grow over time. This is especially true in ****OLTP systems**** where frequent INSERT, UPDATE, and DELETE operations create internal overhead. One often-overlooked issue in such workloads is ****table and index bloat****—a condition where dead tuples consume space, slow down queries, and increase disk costs.

While PostgreSQL provides built-in mechanisms like autovacuum and VACUUM FULL, they come with limitations that make them less suitable for mission-critical systems running 24/7.

## **🧨 The Problem with Traditional Maintenance**

## **❌ Autovacuum**

PostgreSQL’s autovacuum process helps remove dead tuples created during data modifications. However, it only ****marks space as reusable****; it ****does not shrink the physical file size****. As a result, tables can continue to grow even after vacuuming.

## **❌ VACUUM FULL**

VACUUM FULL addresses this by rewriting and compacting the entire table, thus physically shrinking it. However, it requires ****an exclusive lock on the table****, blocking all reads and writes during execution.

This makes VACUUM FULL unsuitable for production environments with high availability requirements or active user sessions.

## **✅ The**pg\_repack**Advantage**

pg\_repack fills the gap by providing an efficient, non-disruptive method to compact PostgreSQL tables and indexes without service interruption.

## **Key Benefits:**

* ****🛠️ Non-blocking Compaction:****  
  Operates online with minimal locking, allowing queries and transactions to continue normally.
* ****📉 Lower Storage Costs:****  
  Physically reclaims space from bloated tables and indexes, reducing disk usage and improving I/O performance.
* ****⚡ Enhanced Performance:****  
  Optimized tables and indexes result in faster queries, better caching behavior, and improved planner statistics.
* ****🧘 Minimal Operational Impact:****  
  Can be scheduled during business hours without impacting end users or application availability.

## **🏁 Conclusion**

If you’re managing ****PostgreSQL in a production setting**** — particularly in ****high-write**** or ****high-availability (HA)**** environments — then pg\_repack is not optional. It is an essential part of your ongoing maintenance strategy.

pg\_repack is especially valuable for:

* ✅ Systems with frequent updates and deletes
* ✅ Databases requiring ****24/7 uptime****
* ✅ Teams needing to ****optimize performance**** without service disruption

With ****minimal effort****, pg\_repack enables you to:

* ✅ Reclaim wasted storage space
* ✅ Improve query execution speed
* ✅ Maintain application stability and uptime

For modern PostgreSQL workloads, pg\_repack is not just a tool—it’s a ****best practice****.