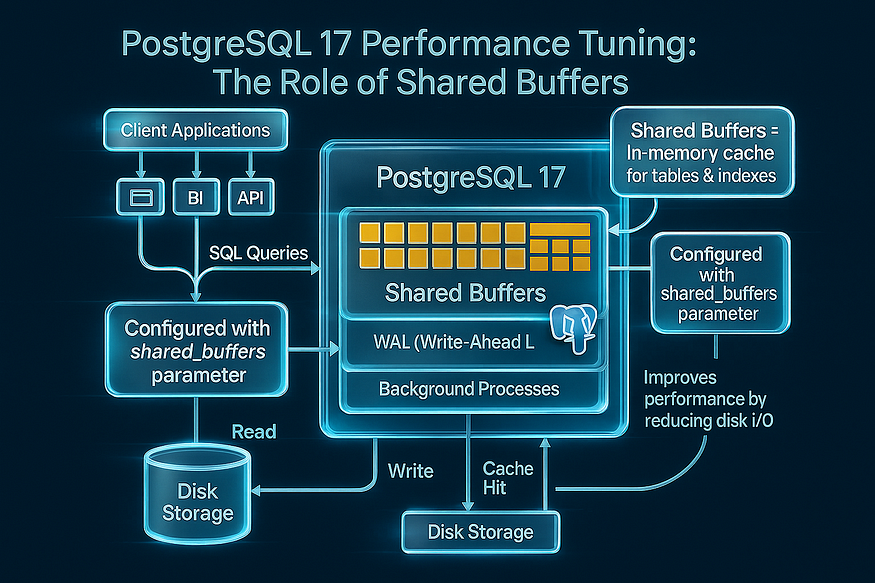
# **31 - PostgreSQL 17 Performance Tuning: Mastering Shared Buffers**



Performance tuning in PostgreSQL often starts with a single parameter: ****shared\_buffers****. This setting defines how much memory PostgreSQL will use to cache table and index blocks before reaching back to disk. Get it wrong, and even simple queries feel sluggish. Get it right, and PostgreSQL becomes lightning-fast.

To illustrate this in practice, we’ll walk through:

* Creating a large dataset,
* Benchmarking with PostgreSQL’s built-in pgbench,
* Observing how shared buffers are used,
* And tuning the parameter to improve performance.

## **Step 1: Creating a Large Dataset**

Let’s start with a simple but heavy dataset. We’ll build a table called items, insert ****10 million rows****, and add indexes to mimic a real-world workload.

CREATE TABLE items (  
 id BIGSERIAL PRIMARY KEY,  
 name TEXT,  
 category TEXT,  
 price NUMERIC(10,2),  
 created\_at TIMESTAMP DEFAULT now()  
);

postgres=# CREATE TABLE items (  
 id BIGSERIAL PRIMARY KEY,  
 name TEXT,  
 category TEXT,  
 price NUMERIC(10,2),  
 created\_at TIMESTAMP DEFAULT now()  
);  
CREATE TABLE  
postgres=#

-- Insert 10 million rows  
INSERT INTO items (name, category, price, created\_at)  
SELECT  
 md5(random()::text) AS name,  
 CASE WHEN random() < 0.5 THEN 'book' ELSE 'clothing' END AS category,  
 (random() \* 200)::numeric(10,2) AS price,  
 now() - (random() \* interval '180 days') AS created\_at  
FROM generate\_series(1, 10000000);

postgres=# INSERT INTO items (name, category, price, created\_at)  
SELECT  
 md5(random()::text) AS name,  
 CASE WHEN random() < 0.5 THEN 'book' ELSE 'clothing' END AS category,  
 (random() \* 200)::numeric(10,2) AS price,  
 now() - (random() \* interval '180 days') AS created\_at  
FROM generate\_series(1, 10000000);  
  
INSERT 0 10000000  
postgres=#

-- Indexes for better query performance  
CREATE INDEX idx\_items\_category ON items(category);  
CREATE INDEX idx\_items\_price ON items(price);  
CREATE INDEX idx\_items\_created\_at ON items(created\_at);

postgres=# CREATE INDEX idx\_items\_category ON items(category);  
CREATE INDEX  
postgres=# CREATE INDEX idx\_items\_price ON items(price);  
CREATE INDEX  
postgres=# CREATE INDEX idx\_items\_created\_at ON items(created\_at);  
CREATE INDEX  
postgres=#

This table now behaves like a catalog in an e-commerce application. Customers can search by category, filter by price, or browse items created in a certain time frame. But how PostgreSQL handles this efficiently depends heavily on memory allocation.

## **Step 2: How Much Memory Should PostgreSQL Use?**

By default, PostgreSQL is extremely conservative. In version 17, the shared\_buffers parameter is still set very low (often just ****128MB****). That’s enough to boot the database on small servers, but it’s far too small for production workloads.

A common ****rule of thumb**** is:

👉 Allocate ****25% of your server’s total RAM**** to shared\_buffers.

So, if your server has 32GB of RAM, a reasonable starting point would be 8GB for PostgreSQL’s cache.

This is not always perfect, but it’s much better than the default.

## **Analogy: Restaurant Seating**

Think of your server as a restaurant. Shared buffers are the number of tables available for customers. The default setting is like running a restaurant with only two tables — technically functional, but hopelessly inefficient. Setting aside 25% of your dining hall space for tables ensures most guests are seated quickly. You might still need to fine-tune later, but it’s a solid start.

## **Step 3: Going Beyond Rules of Thumb**

Rules of thumb are only starting points. To really know if your shared buffer size is right, you have two options:

1. ****Run benchmarks against your own application.**** Measure query times and throughput directly.
2. ****Use PostgreSQL’s internal knowledge of how caching works.****

PostgreSQL ships with a handy benchmarking tool called ****pgbench****. It simulates a simplified bank application, complete with branches, tellers, and accounts. This model is small enough to be easy to run, but large enough to stress test the caching system.

## **Step 4: Running a Benchmark with pgbench**

First, create a test database and initialize it with data:

pgbench -i -s 50 testdb

[postgres@ip-172-31-31-230 ~]$ pgbench -i -s 50 testdb  
dropping old tables...  
NOTICE: table "pgbench\_accounts" does not exist, skipping  
NOTICE: table "pgbench\_branches" does not exist, skipping  
NOTICE: table "pgbench\_history" does not exist, skipping  
NOTICE: table "pgbench\_tellers" does not exist, skipping  
creating tables...  
generating data (client-side)...  
vacuuming...  
creating primary keys...  
done in 19.18 s (drop tables 0.00 s, create tables 0.01 s, client-side generate 8.05 s, vacuum 4.21 s, primary keys 6.91 s).  
[postgres@ip-172-31-31-230 ~]$

Here:

* -i initializes the schema.
* -s 50 sets the scale factor. Each scale adds 1 branch, 10 tellers, and 100,000 accounts. At scale 50, you get millions of rows—more than most small shared buffer caches can hold.

Now run a workload:

pgbench -S -c 8 -t 25000 testdb

* -S runs only ****SELECT queries**** (read-only).
* -c 8 means ****8 clients**** connect in parallel.
* -t 25000 runs ****25,000 transactions per client****.

[postgres@ip-172-31-31-230 ~]$ pgbench -S -c 8 -t 25000 testdb  
pgbench (18.0)  
starting vacuum...end.  
  
transaction type: <builtin: select only>  
scaling factor: 50  
query mode: simple  
number of clients: 8  
number of threads: 1  
maximum number of tries: 1  
number of transactions per client: 25000  
number of transactions actually processed: 200000/200000  
number of failed transactions: 0 (0.000%)  
latency average = 2.818 ms  
initial connection time = 19.305 ms  
tps = 2839.290776 (without initial connection time)  
[postgres@ip-172-31-31-230 ~]$  
[postgres@ip-172-31-31-230 ~]$

This stresses the largest table (accounts) and pushes PostgreSQL to reveal how it uses shared buffers.

## **Installing Required Package:**postgresql-contrib

Before we can use the pg\_buffercache extension in PostgreSQL, we need to ensure that the ****contrib package**** is installed. This package includes additional modules and extensions that do not come with the base PostgreSQL installation but are essential for advanced functionality like buffer cache inspection.

On a Linux system, you can install it using the following command:

sudo yum install postgresql18-contrib

[ec2-user@ip-172-31-31-230 ~]$ sudo yum install postgresql18-contrib  
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: 0:08:55 ago on Tue Sep 30 00:58:54 2025.  
Dependencies resolved.  
====================================================================================================================================================================================================================  
 Package Architecture Version Repository Size  
====================================================================================================================================================================================================================  
Installing:  
 postgresql18-contrib x86\_64 18.0-1PGDG.rhel10 pgdg18 759 k  
Installing dependencies:  
 libxslt x86\_64 1.1.39-8.el10\_0 rhel-10-appstream-rhui-rpms 190 k  
  
Transaction Summary  
====================================================================================================================================================================================================================  
Install 2 Packages  
  
Total download size: 948 k  
Installed size: 3.4 M  
Is this ok [y/N]: y  
Downloading Packages:  
(1/2): postgresql18-contrib-18.0-1PGDG.rhel10.x86\_64.rpm 6.0 MB/s | 759 kB 00:00  
(2/2): libxslt-1.1.39-8.el10\_0.x86\_64.rpm 161 kB/s | 190 kB 00:01  
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------  
Total 774 kB/s | 948 kB 00:01  
Running transaction check  
Transaction check succeeded.  
Running transaction test  
Transaction test succeeded.  
Running transaction  
 Preparing : 1/1  
 Installing : libxslt-1.1.39-8.el10\_0.x86\_64 1/2  
 Installing : postgresql18-contrib-18.0-1PGDG.rhel10.x86\_64 2/2  
 Running scriptlet: postgresql18-contrib-18.0-1PGDG.rhel10.x86\_64 2/2  
Installed products updated.  
  
Installed:  
 libxslt-1.1.39-8.el10\_0.x86\_64 postgresql18-contrib-18.0-1PGDG.rhel10.x86\_64  
  
Complete!  
[ec2-user@ip-172-31-31-230 ~]$

## **Examining PostgreSQL Buffer Cache with**pg\_buffercache

After connecting to the target database, in this case testdb, as the postgres user, we begin by enabling the pg\_buffercache extension. This extension is extremely useful for performance tuning because it allows us to inspect the contents of the shared buffer cache in real time. By understanding what is currently loaded in memory, administrators can make better decisions about memory allocation and query optimization.

\c testdb

postgres=# \c testdb  
You are now connected to database "testdb" as user "postgres".  
testdb=#

Once connected, the first step is to create the extension if it does not already exist:

CREATE EXTENSION IF NOT EXISTS pg\_buffercache;

testdb=# CREATE EXTENSION IF NOT EXISTS pg\_buffercache;  
CREATE EXTENSION  
testdb=#

After creation, we can confirm that the extension is successfully installed by listing it with \dx:

\dx pg\_buffercache

testdb=# \dx pg\_buffercache  
 List of installed extensions  
 Name | Version | Default version | Schema | Description  
----------------+---------+-----------------+--------+---------------------------------  
 pg\_buffercache | 1.6 | 1.6 | public | examine the shared buffer cache  
(1 row)

Here, we can clearly see that pg\_buffercache is available, along with its version, schema, and description. The key point is the description: *“examine the shared buffer cache”*. This is precisely what enables us to observe the state of PostgreSQL’s buffer management system.

## **Querying the Buffer Cache**

With the extension installed, we can now query its view. For example, a simple count of rows in pg\_buffercache shows us the total number of buffers managed:

testdb=# SELECT count(\*) FROM pg\_buffercache;  
 count  
-------  
 16384  
(1 row)

This output tells us that the buffer cache in this PostgreSQL instance is composed of ****16,384 buffers****. Each buffer represents a fixed-size memory block (commonly 8 KB).

## **Breaking Down the Calculation**

The value 16384 is not random; it directly corresponds to PostgreSQL’s configuration parameter shared\_buffers. The formula is:

Total Cache Size = Number of Buffers × Size of Each Buffer

By default, each buffer is ****8 KB**** in size. Using the query result:

16384 buffers × 8 KB = 131,072 KB  
131,072 KB ÷ 1024 = 128 MB

So, the ****shared buffer cache is 128 MB**** in this instance.

This is a crucial observation because shared\_buffers is one of the most important parameters for PostgreSQL performance tuning. It determines how much memory PostgreSQL dedicates to caching table and index data, reducing the need for disk reads during query execution.

## **Why This Matters**

* If the cache is too ****small****, PostgreSQL will constantly evict pages to make room for new ones, leading to higher disk I/O and slower performance.
* If the cache is too ****large**** (beyond what the system can handle comfortably), it may cause memory pressure and slow down the operating system or other processes.

By checking the buffer cache through pg\_buffercache, administrators can see how many pages are being cached and later analyze *which* relations (tables or indexes) occupy those buffers. This visibility is the first step toward fine-tuning PostgreSQL memory usage and ensuring queries run efficiently.

## **Step 5: Analyzing the Cache with pg\_buffercache**

Once the benchmark is running, you can inspect how memory is being used with the pg\_buffercache extension:

SELECT c.relname,  
 count(\*) AS buffers,  
 round(100.0 \* count(\*) /   
 (SELECT setting::int FROM pg\_settings WHERE name='shared\_buffers')) AS percent\_of\_cache  
FROM pg\_buffercache b  
JOIN pg\_class c ON b.relfilenode = pg\_relation\_filenode(c.oid)  
JOIN pg\_namespace n ON n.oid = c.relnamespace  
WHERE n.nspname NOT IN ('pg\_catalog', 'information\_schema')  
GROUP BY c.relname  
ORDER BY buffers DESC  
LIMIT 10;

testdb=# SELECT c.relname,  
 count(\*) AS buffers,  
 round(100.0 \* count(\*) /  
 (SELECT setting::int FROM pg\_settings WHERE name='shared\_buffers')) AS percent\_of\_cache  
FROM pg\_buffercache b  
JOIN pg\_class c ON b.relfilenode = pg\_relation\_filenode(c.oid)  
JOIN pg\_namespace n ON n.oid = c.relnamespace  
WHERE n.nspname NOT IN ('pg\_catalog', 'information\_schema')  
GROUP BY c.relname  
ORDER BY buffers DESC  
LIMIT 10;  
 relname | buffers | percent\_of\_cache  
-----------------------+---------+------------------  
 pgbench\_accounts | 8916 | 54  
 pgbench\_accounts\_pkey | 7115 | 43  
 pg\_toast\_2619\_index | 2 | 0  
 pg\_toast\_2619 | 1 | 0  
(4 rows)  
  
testdb=#

Ignoring system tables, you’ll likely see the ****accounts table**** and its ****primary key index**** dominating the cache.

## **Analogy: Supermarket Shelf Space**

The shared buffer cache is like the front shelves of a supermarket. Popular items like bread and milk (the accounts table and its index) take up the majority of space. Less popular items are stored in the backroom (disk) and are slower to retrieve.

## **Step 6: What Happens with Different Shared Buffer Sizes**

At the default size of ****128MB****, you might see:

* Accounts table = ~43.4% of cache.
* Primary key index = ~51.8% of cache.

Clearly, PostgreSQL considers these objects critical, but the cache is cramped.

Now, increase shared\_buffers to ****256MB****:

nano $PGDATA/postgresql.conf  
# Change:  
shared\_buffers = 256MB

Restart PostgreSQL, rerun the benchmark, and check the cache again.

This time, the accounts table might occupy close to ****90%**** of cache. More of your hot data stays in memory, reducing disk reads and speeding up queries significantly.

## **Step 7: Understanding Usage Counts**

Each block in shared buffers also has a ****usage count****, which shows how often it has been accessed:

* ****Counts 0–1:**** Rarely used, can be evicted easily.
* ****Counts 4–5:**** Frequently used, PostgreSQL strongly prefers to keep these in memory.

## **Analogy: Movie Theater Seats**

Shared buffers are like a theater with limited seats. If most are filled with people who only pop in for a scene (low usage count), you could shrink the theater and still serve them. But if most seats are filled with die-hard fans watching every show (high usage count), adding more seats (increasing shared buffers) makes the experience better for everyone.

## **Step 8: How to Interpret the Results**

* If most of your blocks have ****low usage counts****, your cache may be too large — you’re giving memory to data that rarely gets reused.
* If many blocks have ****high usage counts****, your cache is working hard and would benefit from being larger.

This way, you’re not guessing. You’re using hard evidence from your workload to guide memory tuning.

## **Final Takeaway**

Shared buffers are the beating heart of PostgreSQL performance. They decide which parts of your data stay instantly available and which must be fetched from disk.

Start by allocating ****25% of system RAM****, then use ****pgbench**** and ****pg\_buffercache**** to measure and adjust. Look not only at which tables dominate the cache, but also at usage counts to understand whether your memory is appropriately sized.

In short:

* ****Too small**** = PostgreSQL constantly reloads from disk.
* ****Too large**** = You may waste memory better left for the OS.
* ****Just right**** = Hot data stays cached, queries fly, and PostgreSQL feels smooth under load.

## **Analogy Recap:**

* ****Restaurant Tables**** → Number of seats (buffer size).
* ****Supermarket Shelves**** → Which products (tables/indexes) dominate cache space.
* ****Movie Theater**** → Usage counts show which blocks are worth keeping.

By carefully tuning shared\_buffers, you give PostgreSQL the space it needs to serve your data efficiently, even under heavy workloads with millions of rows.