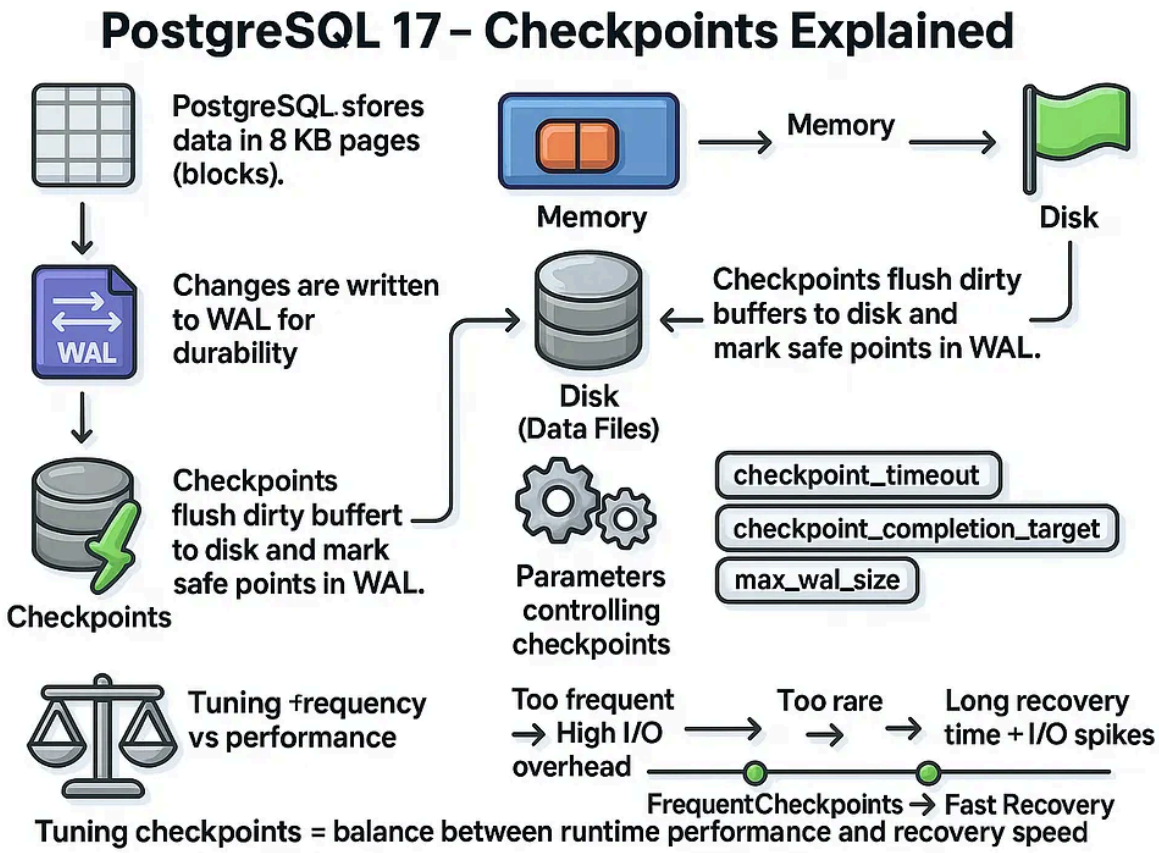
04 - PostgreSQL 17 Performance Tuning: Checkpoints Explained

18 min read · Aug 31, 2025

Jeyaram Ayyalusamy Following



Checkpoints are one of the most critical processes in PostgreSQL's performance and durability model. They ensure that changes in memory eventually make it safely to disk. Without checkpoints, PostgreSQL would risk losing large portions of data during crashes.

Let's break it down step by step.

Step 1 — Blocks and Pages

understand how PostgreSQL stores and works with data — and that begins with **blocks (pages)**.

What Are Blocks (Pages)?

PostgreSQL never writes data byte by byte. Instead, it works with **fixed-size blocks**, also called **pages**.

- By default, each block in PostgreSQL is **8 KB**.
- Whether your row takes up 10 bytes or 2 KB, PostgreSQL still allocates a **full 8 KB block**.
- These blocks are the fundamental unit of storage and I/O in PostgreSQL.

This design makes data management predictable and efficient because PostgreSQL always knows exactly how much space one page takes.

Example: A Tiny Table Still Takes 8 KB

Let's look at a simple real-world example.

```
CREATE TABLE employee (
  id SERIAL,
  last_name VARCHAR
);
```

```
postgres=# CREATE TABLE employee (
  id SERIAL,
  last_name VARCHAR
);
CREATE TABLE
postgres=#
```

```
INSERT INTO employee (last_name) VALUES ('Smith');
```

```
postgres=# INSERT INTO employee (last_name) VALUES ('Smith');
INSERT 0 1
postgres=#
```

Here's what happens:

- The row (`id = 1, last_name = 'Smith'`) might take **just a few bytes.**
- But PostgreSQL **still allocates a full 8 KB block** for it.
- If you check the file size on disk, you'll see:

Check Block Size in PostgreSQL

PostgreSQL stores data in **fixed-size blocks (pages)**. By default, this is **8 KB**, unless you built PostgreSQL from source with a different size.

You can check it using SQL:

```
SHOW block_size;
```

✅ Example output:

```
postgres=# show block_size
postgres-# ;
 block_size
------------
 8192
(1 row)

postgres=#
```

Check Physical Block Size (Operating System / Disk)

On Linux/Unix systems, you can check the **filesystem block size** using OS commands:

- To check file system block size:

```
postgres=# SHOW data_directory;
     data_directory
------------------------
 /var/lib/pgsql/16/data
(1 row)

postgres=#

postgres=# exit
[postgres@ip-172-31-25-61 ~]$ stat -f -c "%s" /var/lib/pgsql/16/data
4096
[postgres@ip-172-31-25-61 ~]$
```

✅ Summary

- `SHOW block_size;` → PostgreSQL's internal block size (usually 8192 = 8 KB).
- `stat -f -c "%s" <data_directory>` → OS filesystem block size (commonly 4096 = 4 KB).

✅ Even though you only stored one small row, the minimum size of the table is 8 KB because that's the block size PostgreSQL works with.

Why 8 KB Blocks?

You might wonder: why not store data more compactly, byte by byte?

Here's why PostgreSQL (and most databases) use fixed blocks:

1. **Efficiency** → Reading and writing in consistent chunks is faster than handling variable-length byte operations.
2. **Caching** → PostgreSQL can keep recently used blocks in memory (shared buffers). Since they're always the same size, memory management becomes much simpler.
3. **Indexing** → Indexes are also organized in terms of blocks, making lookups predictable.
4. **Durability** → Checkpoints and crash recovery rely on flushing entire blocks, not arbitrary byte ranges.

👉 Analogy: Think of blocks like **shipping containers**. Whether you put one box or fill it completely, the container is always the same size. This makes transport, storage, and logistics much simpler.

Practical Observation

You can see this behavior in action. After creating the `employee` table with just one row, locate its data file in PostgreSQL's data directory and check its size.

It will be exactly **8192 bytes (8 KB)** — proof that PostgreSQL allocates in blocks, not in variable bytes.

Key Takeaway

- PostgreSQL **always works with blocks (pages)**, not raw bytes.
- The default block size is **8 KB**.
- Even the smallest table with one row will still consume 8 KB on disk.

- This block-based design is the foundation for how PostgreSQL manages performance, memory buffers, and ultimately **checkpoints**.

✅ By understanding blocks, you've learned the first piece of the checkpoint puzzle. Next, we'll dive into how PostgreSQL handles **dirty buffers** — the changes made in memory before checkpoints write them safely to disk.

Step 2 — Dirty Buffers

In the previous step, we learned that PostgreSQL stores and works with data in **8 KB blocks (pages)**. Now, let's see what happens when a user **updates or inserts data** into a table.

PostgreSQL does **not immediately write data changes to disk**. Instead, it temporarily stores them in memory for performance reasons. These in-memory changes are called **dirty buffers**.

What Exactly Is a Dirty Buffer?

A **dirty buffer** is a page (block) in PostgreSQL's **shared buffer memory** that has been changed but not yet written to the table file on disk.

- "Dirty" means the copy in memory is **different** from the copy on disk.
- The changes in dirty buffers will eventually be written to disk during a **checkpoint** or background write.

Step-by-Step: What Happens During an Update

Imagine we have an `employee` table:

```
CREATE TABLE employee (
  id SERIAL,
  last_name VARCHAR
);
```

```
INSERT INTO employee (last_name) VALUES ('Smith');
```

Now let's update the record:

```
UPDATE employee
SET last_name = 'Johnson'
WHERE id = 1;
```

Here's what PostgreSQL does internally:

Step 1: Load the Block into Memory

- PostgreSQL checks if the block containing `id = 1` is already in the shared buffer.
- If not, it loads the block from disk into memory.

Step 2: Modify the Block in Memory

- PostgreSQL changes the value from `'Smith'` → `'Johnson'` directly in memory.
- At this moment, the block in memory no longer matches the block on disk.
- This block is now called a **dirty buffer**.

Step 3: Record the Change in the WAL

- Before telling the user the update was successful, PostgreSQL writes an entry in the **Write-Ahead Log (WAL)**.
- The WAL is a sequential log of all changes.
- WAL ensures that even if PostgreSQL crashes, it can replay these changes during recovery.

Step 4: Commit

- When the user runs `COMMIT`, PostgreSQL guarantees that the WAL entry has been written to disk.
- ✅ This means the change is durable.
- ❌ But the actual table file still hasn't been updated yet — the block is still dirty in memory.

Why Does PostgreSQL Use Dirty Buffers?

You might wonder: why not write changes to disk immediately?

The answer is **performance and safety**.

1. **Performance**

- Writing to memory is much faster than writing to disk.
- PostgreSQL can batch multiple changes together and flush them later.

2. **Durability**

- WAL provides crash protection. Even if the database crashes before dirty buffers are written, PostgreSQL can replay WAL to recover changes.

Real-World Analogy

Think of dirty buffers like **unsaved work in a text editor**:

- You type changes (update rows) → they go into memory (dirty buffers).
- The program writes a small autosave log (WAL) → so your edits aren't lost if the computer crashes.
- The file on disk (the actual table file) is only updated when you hit **Save** (checkpoint).

👉 In PostgreSQL, dirty buffers are your unsaved edits, and WAL is the autosave safety net.

Example Walkthrough

1. Initial row:

```
id = 1, last_name = 'Smith'
```

2. Run update:

```
UPDATE employee SET last_name = 'Johnson' WHERE id = 1;
```

3. PostgreSQL steps:

- Loads the block into shared buffer.
- Changes last name to `'Johnson'` in memory.
- Marks page as dirty.
- Writes WAL entry: *Row with id=1 updated: Smith → Johnson.*

4. On commit:

- WAL is flushed to disk.
- Dirty buffer still in memory, waiting for a checkpoint to be written to disk.

⚠️ Important Point

A `COMMIT` in PostgreSQL guarantees **WAL durability**, not immediate table file updates.

- ✅ WAL ensures no committed transaction is lost.
- ❌ Table files may still have old data until dirty buffers are flushed by a checkpoint.

✅ Key Takeaways

- Dirty buffers = blocks in memory modified but not yet written to disk.
- COMMIT ensures WAL durability, not immediate data file update.
- This design makes PostgreSQL:
- **Fast** → fewer direct disk writes.
- **Safe** → WAL ensures crash recovery.
- Dirty buffers are later flushed to disk during a **checkpoint**.

Step 3 — The Role of Checkpoints

So far, we've seen how PostgreSQL stores changes in **dirty buffers** (memory) and ensures safety with the **Write-Ahead Log (WAL)**. But those changes in memory cannot stay there forever — they must eventually reach the **data files on disk**.

That's where **checkpoints** come in.

What Is a Checkpoint?

A **checkpoint** is a background process in PostgreSQL that ensures data in memory (dirty buffers) is safely and consistently written to disk.

During a checkpoint, PostgreSQL does four key things:

1. **Flushes all dirty pages to disk**

- Any modified blocks in memory are written to their permanent storage (table files and index files).

2 . **Marks dirty pages as clean**

- Once written, those memory pages are no longer "dirty" because they match what's on disk.

3. **Writes a checkpoint record (Redo record) to WAL**

- This record marks a "save point" in the log. Everything before this point is guaranteed to be safely stored on disk.

4. **Marks WAL entries up to this point as applied**

- Older WAL segments are no longer needed for crash recovery and can be recycled or removed.

Why Are Checkpoints Critical?

Checkpoints give PostgreSQL a **recovery guarantee**:

- All changes before the checkpoint are permanently stored in data files.
- In the event of a crash, PostgreSQL only needs to replay WAL **after** the last checkpoint.

👉 Without checkpoints, WAL would grow indefinitely, and crash recovery could take hours (or days) because PostgreSQL would need to replay the entire log history.

Step-by-Step Example

Let's walk through an example with the `employee` table.

Update Operation

```
UPDATE employee
SET last_name = 'Johnson'
WHERE id = 1;
```

Internal Process

1. The row (`Smith`) is changed to (`Johnson`) in **memory** (dirty buffer).
2. WAL records the update (`Smith → Johnson`) and writes it to disk.
3. At this point:

- WAL = updated and durable ✅
- Table file = still outdated ❌

At Checkpoint

- PostgreSQL flushes the dirty buffer to disk → the `employee` table file is updated with `'Johnson'`.
- Memory page is marked clean.
- A **checkpoint record** is written in WAL to mark this safe point.

✅ Now both WAL and the table file are consistent.

Analogy: Checkpoints as Save Points 🎮

Think of checkpoints like **saving progress in a video game**:

- While playing, your moves are in temporary memory (dirty buffers).

- WAL is like an autosave journal, recording what you did step by step.
- A checkpoint is when you actually hit **Save Game** — locking your progress into the permanent file.
- If your console crashes, you restart from the last save point (checkpoint), not from the beginning.

What Happens During Recovery?

If PostgreSQL crashes:

1. On restart, it finds the **last checkpoint record** in WAL.
2. It replays all changes **after** that checkpoint to bring the database back to a consistent state.
3. Since all data before the checkpoint is guaranteed safe, only recent transactions need replay.

👉 This dramatically reduces recovery time compared to replaying the entire WAL history.

✅ Key Takeaways

- A **checkpoint** ensures dirty buffers are flushed to disk and WAL progress is marked safe.
- All changes before a checkpoint are guaranteed to be in permanent storage.
- WAL entries before the checkpoint can be recycled, keeping logs manageable.
- Example: Updating `'Smith → Johnson'` is only truly consistent once the dirty buffer is flushed at a checkpoint.
- Analogy: Like a **game save point** — it guarantees you won't lose progress before that save.

✅ By understanding checkpoints, you now see how PostgreSQL keeps a balance between **performance** (delaying direct disk writes) and **durability** (ensuring recovery after a crash).

Step 4 — Checkpoints and Crash Recovery

So far, we've seen that checkpoints flush dirty buffers to disk and mark safe points in the Write-Ahead Log (WAL). But their most critical role is during **crash recovery**.

If PostgreSQL crashes due to a power failure, OS error, or hardware issue, checkpoints are what make the difference between quick recovery and hours of downtime.

Why Checkpoints Are Essential

Checkpoints provide **safe recovery points**.

- All data changes before a checkpoint are guaranteed to be on disk.
- WAL entries after a checkpoint describe only the recent changes that may not have been flushed yet.
- During a crash, PostgreSQL doesn't need to start from scratch — it only needs to **replay WAL from the last checkpoint forward**.

👉 This drastically reduces recovery time.

How Recovery Works (Step by Step)

Let's break down PostgreSQL's recovery process:

1. Crash occurs

PostgreSQL stops suddenly — some dirty buffers may still be in memory, not yet written to disk.

2. Restart begins

On restart, PostgreSQL searches the WAL for the **last checkpoint record**.

3. Identify the safe point

The checkpoint record says:

> "All changes up to this point are already written to disk."

4. Replay WAL forward (Redo Phase)

- PostgreSQL re-applies WAL entries from the checkpoint forward.
- This ensures any transactions committed after the checkpoint but before the crash are restored.

5. Database is consistent again

Once WAL replay finishes, PostgreSQL is back in a clean, usable state.

Example Timeline

- **10:00 AM** → Checkpoint occurs.
- All changes before 10:00 are flushed to disk.
- Checkpoint record written in WAL.
- **10:00–10:05 AM** → Users insert, update, and delete rows.
- WAL logs these operations.
- Some pages are still dirty in memory.
- **10:05 AM** → PostgreSQL crashes unexpectedly.

On Restart

- PostgreSQL finds the checkpoint record from **10:00 AM**.

- It knows everything before 10:00 is safe on disk.
- It **replays WAL entries from 10:00–10:05.**
- Once replay is done, the database is consistent.

✅ Recovery time = only the amount of WAL generated between 10:00 and 10:05.

Why Checkpoint Frequency Matters

Checkpoint frequency directly affects recovery time:

Frequent checkpoints

- ✅ Shorter recovery (less WAL to replay).
- ❌ More I/O overhead during normal operations (flushing buffers more often).

Less frequent checkpoints

- ✅ Lower I/O load during workload.
- ❌ Longer recovery time (more WAL to replay).

👉 Database administrators (DBAs) must balance between **system performance during normal operation** and **fast recovery during crashes**.

Analogy: Autosave in a Game 🎮

Think of checkpoints like **autosave points in a video game**:

- If the game autosaves every 5 minutes and you crash, you only need to replay the last 5 minutes.
- If it autosaves every 30 minutes, you might have to replay half an hour of progress.

PostgreSQL works the same way:

- **Checkpoint = autosave**
- **WAL replay = replaying the moves since last save**

The closer the last checkpoint is to the crash, the faster recovery will be.

Real-World Example

Consider a busy e-commerce site:

- With checkpoints every **5 minutes**:
- Crash at 2:05 → recovery replays only 5 minutes of WAL.
- Site is back online quickly.
- With checkpoints every **30 minutes**:
- Crash at 2:05 → recovery replays 30 minutes of WAL.
- Site downtime is much longer.

This shows why checkpoint tuning is essential in PostgreSQL 17, especially for high-traffic systems where downtime costs money.

✅ Key Takeaways

- Checkpoints = **safe recovery points**.
- On crash, PostgreSQL finds the last checkpoint and replays WAL entries **after** it.
- Recovery is much faster than starting from the beginning of WAL.
- **Frequent checkpoints** → faster recovery, more I/O load.
- **Infrequent checkpoints** → lighter load, longer recovery.
- Example: Checkpoint at **10:00 AM**, crash at **10:05 AM** → only 5 minutes of WAL needs replay.

By combining **WAL logging** with **checkpoints**, PostgreSQL 17 ensures a balance between **performance** and **durability**, keeping your database reliable even in the face of unexpected failures.

Step 5 — Parameters that Control Checkpoints

We know checkpoints are essential for durability and crash recovery. But how often they happen, how aggressively they flush data, and how evenly they spread writes — all of this is controlled by parameters in PostgreSQL.

Tuning these parameters allows you to balance **system performance during normal operation** with **faster recovery after crashes**. Let's break them down one by one.

- `checkpoint_flush_after`

What It Does

This parameter defines how many **pages (blocks of 8 KB each)** should be written before PostgreSQL forces a flush to disk.

- **Default:** `0` → disabled (let the operating system decide when to flush).
- **Smaller value:** More frequent flushes.
- **Larger value:** Flushes happen less often, meaning more dirty pages can accumulate before being written.

Example

Suppose `checkpoint_flush_after = 32`.

- PostgreSQL will flush every time 32 pages (≈ 256 KB) are written.
- This prevents too many dirty buffers from piling up in memory, which could cause a big I/O storm later.

Analogy

Think of this as **taking out the trash**:

- If you take it out after a few items (smaller value), the house stays clean, but you make more trips.
- If you wait until the bin is full (larger value), you make fewer trips, but it's heavier and harder to carry.

- `checkpoint_timeout`

What It Does

This parameter sets the **maximum amount of time** that can pass before a checkpoint is forced.

- **Default:** `300 seconds = 5 minutes`.
- Even if `checkpoint_flush_after` hasn't triggered, PostgreSQL will perform a checkpoint after the timeout expires.

Example

- If `checkpoint_timeout = 5 minutes`, PostgreSQL guarantees that at least one checkpoint will occur every 5 minutes.
- If your workload is **write-heavy**, dirty pages may fill buffers before 5 minutes, triggering a checkpoint earlier.
- If your workload is **light**, checkpoints may only occur at the timeout interval.

Analogy

Think of this as **changing the oil in your car**:

- You can drive until the dashboard light comes on (flush threshold),
- But even if it doesn't, you must change the oil after a set time (timeout) to keep the engine safe.

- `checkpoint_completion_target`

What It Does

This parameter controls **how evenly PostgreSQL spreads the work of a checkpoint** across time.

- **Default:** `0.5` → PostgreSQL tries to finish the checkpoint in half the time before the next checkpoint is due.
- **Higher values (e.g., 0.9):** Spread writes more smoothly across the whole interval → fewer I/O spikes.
- **Lower values (e.g., 0.3):** Finish checkpoints quickly, but risk causing sudden bursts of disk activity.

Example

- With `checkpoint_timeout = 10 minutes` and `checkpoint_completion_target = 0.5`:
- PostgreSQL tries to finish the checkpoint in 5 minutes.
- With `checkpoint_completion_target = 0.9`:
- It spreads the writes over 9 minutes, leading to smoother performance but leaving less buffer time before the next checkpoint.

Analogy

Think of this as **eating a big meal**:

- If you eat quickly (low value), you finish fast but might feel overloaded.
- If you spread it out slowly (high value), your body handles it better without spikes of discomfort.

✅ Key Takeaways

- **`checkpoint_flush_after`** → Controls how often buffers are flushed. Smaller values = more frequent flushes, smoother performance, but higher overhead.

- **`checkpoint_timeout`** → Maximum time between checkpoints (default 5 minutes). Shorter time = quicker recovery after crashes, but more frequent checkpoints.
- **`checkpoint_completion_target`** → Controls how evenly writes are spread. Higher values = smoother performance, fewer I/O spikes.

Real-World Example: Tuning for Workloads

1. **OLTP system (e.g., banking, e-commerce)**

- Many small, continuous writes.
- Recommended:
- `checkpoint_timeout = 5 min` (or lower).
- `checkpoint_flush_after = 32` (to avoid too many dirty buffers).
- `checkpoint_completion_target = 0.9` (to smooth out I/O).

2. **Data warehouse (batch writes, large imports)**

- Bulk data loads with fewer frequent commits.
- Recommended:
- `checkpoint_timeout = 15–30 min` (fewer checkpoints during long batch jobs).
- `checkpoint_flush_after = 256` (reduce background flush overhead).
- `checkpoint_completion_target = 0.7` (balance between spreading writes and finishing early).

Final Thought

Tuning checkpoint parameters in PostgreSQL 17 is about finding the right balance:

- Too aggressive → frequent I/O, reduced throughput.

- Too relaxed → long recovery times after crashes.

👉 By carefully adjusting `checkpoint_flush_after`, `checkpoint_timeout`, and `checkpoint_completion_target` based on your workload, you can ensure PostgreSQL remains both **high-performing** and **reliable**.

Step 6 — Balancing Frequency vs Performance

One of the trickiest parts of PostgreSQL performance tuning is deciding **how often checkpoints should run**.

- Run them too often → you hurt performance because of constant flushing.
- Run them too rarely → you risk big I/O spikes and painfully long crash recovery.

The challenge is to find the right **balance** between performance during normal operations and recovery speed after a crash.

Frequent Checkpoints

With frequent checkpoints, PostgreSQL writes dirty buffers to disk more often.

✅ Benefits

- **Faster crash recovery** → WAL replay is shorter because data is flushed more frequently.
- **Lower recovery risk** → Databases come back online quickly after a failure.

❌ Drawbacks

- **Higher I/O load** → Writing to disk frequently consumes more I/O bandwidth.
- **Reduced throughput** → Queries may slow down because PostgreSQL is busy flushing.

👉 Example:

- `checkpoint_timeout = 5 minutes`

- Crash occurs at 4 minutes into the interval.
- PostgreSQL replays **only 4 minutes of WAL.**
- Recovery is quick, but normal performance suffers due to constant flushing.

Infrequent Checkpoints

With less frequent checkpoints, PostgreSQL delays flushing dirty buffers to disk.

✅ Benefits

- **Better throughput** → PostgreSQL can handle more queries without frequent I/O interruptions.
- **Ideal for heavy-write workloads** → Systems with bulk writes benefit from fewer checkpoints.

❌ Drawbacks

- **I/O spikes** → When a checkpoint finally occurs, a large number of dirty buffers are flushed at once, slowing down queries.
- **Longer crash recovery** → More WAL entries must be replayed.

👉 Example:

- `checkpoint_timeout = 30 minutes`
- Crash occurs at 29 minutes into the interval.
- PostgreSQL replays **29 minutes of WAL.**
- Throughput is high during normal operations, but crash recovery takes much longer.

Real-World Example: Heavy Write Workload

Imagine a production server with **16 GB shared buffers** handling continuous inserts and updates:

Low `checkpoint_timeout` (5 minutes)

- Dirty buffers flushed often.
- Crash recovery = quick (only 5 minutes of WAL to replay).
- ❌ But I/O is constantly busy → reduces system throughput.

High `checkpoint_timeout` (30 minutes)

- Dirty buffers accumulate for half an hour.
- Crash recovery = long (up to 30 minutes of WAL replay).
- ❌ When checkpoint runs, it creates a huge I/O storm, slowing queries temporarily.

This is why DBAs must carefully tune checkpoint frequency based on workload type.

Analogy: Cleaning Your Kitchen 🍽️

Think of checkpoints like cleaning up while cooking:

Frequent cleaning

- ✅ Kitchen stays tidy.
- ✅ If guests arrive (crash), you don't have much to clean up.
- ❌ But you spend more time cleaning than cooking.

Infrequent cleaning

- ✅ You cook faster without interruptions.
- ❌ But when it's finally time to clean, the mess is huge.
- ❌ If guests arrive unexpectedly, recovery takes longer.

PostgreSQL works the same way — frequent checkpoints keep things safe but add overhead, while infrequent checkpoints improve performance but risk longer clean-

up times.

Step-by-Step Recap

1. **Checkpoint Frequency = Trade-off**

- Frequent = safer, but higher cost.
- Infrequent = faster workload, but riskier recovery.

2. **Impact on WAL Replay**

- Short intervals = replay a few minutes of WAL.
- Long intervals = replay up to 30+ minutes of WAL.

3. **Impact on I/O**

- Frequent = constant small flushes.
- Infrequent = occasional massive flush (I/O spike).

4. **Choosing the Right Balance**

- OLTP systems (banking, e-commerce): prefer frequent checkpoints for fast recovery.
- OLAP/data warehouses: prefer less frequent checkpoints to maximize throughput.

✅ Key Takeaways

Frequent checkpoints

- ✅ Fast crash recovery.
- ❌ More I/O overhead, lower throughput.

Infrequent checkpoints

- ✅ Better write performance.
- ❌ Risk of I/O spikes and longer downtime after crashes.

👉 PostgreSQL 17 checkpoint tuning is about striking the balance:

- **Critical systems** → shorter intervals for safer recovery.
- **Analytics/bulk systems** → longer intervals for higher throughput.

⚖️ **Checkpoint tuning is like walking a tightrope:** lean too much on one side and you lose performance, lean too much on the other and you risk long downtime. The art lies in balancing both.

Step 7 — Recovery Trade-offs

Checkpoints play a vital role not only in normal database operations but also in **crash recovery**. How you tune checkpoint frequency directly impacts how quickly PostgreSQL can recover after an unexpected failure. This is where the concept of **recovery trade-offs** comes into play.

How PostgreSQL Recovers After a Crash

When PostgreSQL crashes (for example, due to power failure or hardware issues), it must ensure that the database is brought back to a **consistent state** before accepting new queries. This happens in three steps:

Step 1: Find the Last Checkpoint Record

- PostgreSQL looks into the **Write-Ahead Log (WAL)** to locate the most recent checkpoint record.
- This record guarantees that **all changes before it are safely on disk.**

Step 2: Replay WAL Entries After the Checkpoint

- Starting from the checkpoint, PostgreSQL begins the **redo phase**.
- It re-applies all changes recorded in WAL that happened after the checkpoint but before the crash.
- This ensures that every committed transaction is recovered.

Step 3: Stop at the Last Commit

- Once PostgreSQL has replayed WAL up to the last commit, the system is consistent again.
- At this point, it is safe to accept new transactions.

The Recovery Trade-Off

The key factor that affects recovery time is **how recent the last checkpoint was:**

Recent Checkpoint

- Less WAL to replay.
- Recovery is quick, downtime is minimal.

Old Checkpoint

- More WAL to replay.
- Recovery takes longer, causing extended downtime.

Example Scenarios

Case 1: `checkpoint_timeout = 5 minutes`

- Checkpoints occur every 5 minutes.
- If the system crashes at **12:04 PM**, PostgreSQL only needs to replay at most 4 minutes of WAL.
- Recovery is fast, and downtime is short.

Case 2: `checkpoint_timeout = 30 minutes`

- Checkpoints occur every 30 minutes.
- If the system crashes at **12:29 PM**, PostgreSQL may need to replay **up to 29 minutes of WAL**.
- Recovery takes longer, leading to more downtime.

Analogy: Saving a School Essay ✍️

Think of checkpoints like **saving your essay while typing**:

- If you save every **5 minutes** and the computer crashes, you only lose a small portion of your work. Recovery is quick.
- If you save every **30 minutes**, you might lose half an hour of writing in a single crash, forcing you to retype a lot.

👉 PostgreSQL works the same way: the longer the gap between checkpoints, the more "work" (WAL entries) it has to replay to catch up.

Impact on Database Operations

Checkpoint frequency influences two things:

1. **Normal Performance**

- Frequent checkpoints = more I/O overhead during normal operations.
- Infrequent checkpoints = better performance in the short term.

2. **Crash Recovery Time**

- Frequent checkpoints = faster recovery, less downtime.
- Infrequent checkpoints = slower recovery, more downtime.

This is why DBAs must carefully tune checkpoint parameters based on workload and business needs.

✅ Key Takeaways

- During crash recovery, PostgreSQL:

1. Finds the last checkpoint record.
2. Replays WAL entries after it.
3. Stops at the last commit.

- **Frequent checkpoints** → less WAL to replay, faster recovery, minimal downtime.
- **Infrequent checkpoints** → more WAL to replay, slower recovery, longer downtime.
- Example:
- `checkpoint_timeout = 5 minutes` → replay up to 5 minutes of WAL.
- `checkpoint_timeout = 30 minutes` → replay up to 30 minutes of WAL.

⚖️ In PostgreSQL 17, tuning checkpoints is a **balancing act**: you must decide between **higher performance during normal workloads** or **faster recovery after a crash**. The right balance depends on whether your priority is **throughput** (e.g., data warehouse) or **availability** (e.g., banking, e-commerce).

Summary

- PostgreSQL always works with **8 KB blocks (pages)**.
- Updates go to memory first → buffers become **dirty**.
- **Checkpoints** flush dirty buffers to disk, mark them clean, and record progress in WAL.
- Crash recovery starts from the last checkpoint.

Parameters:

- `checkpoint_flush_after` → how many pages before flush.
- `checkpoint_timeout` → max time between checkpoints.
- `checkpoint_completion_target` → how spread out writes should be.
- Frequent checkpoints = faster recovery but more I/O load.
- Infrequent checkpoints = better short-term performance but longer crash recovery.
- Analogy: Like **game save points** — frequent saves reduce risk, infrequent saves are faster but riskier.

✅ By tuning checkpoint settings in PostgreSQL 17, DBAs can balance **throughput, recovery time, and stability** for their workloads.

Your support means a lot — and you'll never miss a practical guide again!

🔗 Let's Connect!

If you enjoyed this post or would like to connect professionally, feel free to reach out to me on LinkedIn:
👉 Jeyaram Ayyalusamy

I regularly share content on **PostgreSQL, database administration, cloud technologies, and data engineering.** Always happy to connect, collaborate, and discuss ideas!

Postgresql AWS Database Oracle Open Source

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

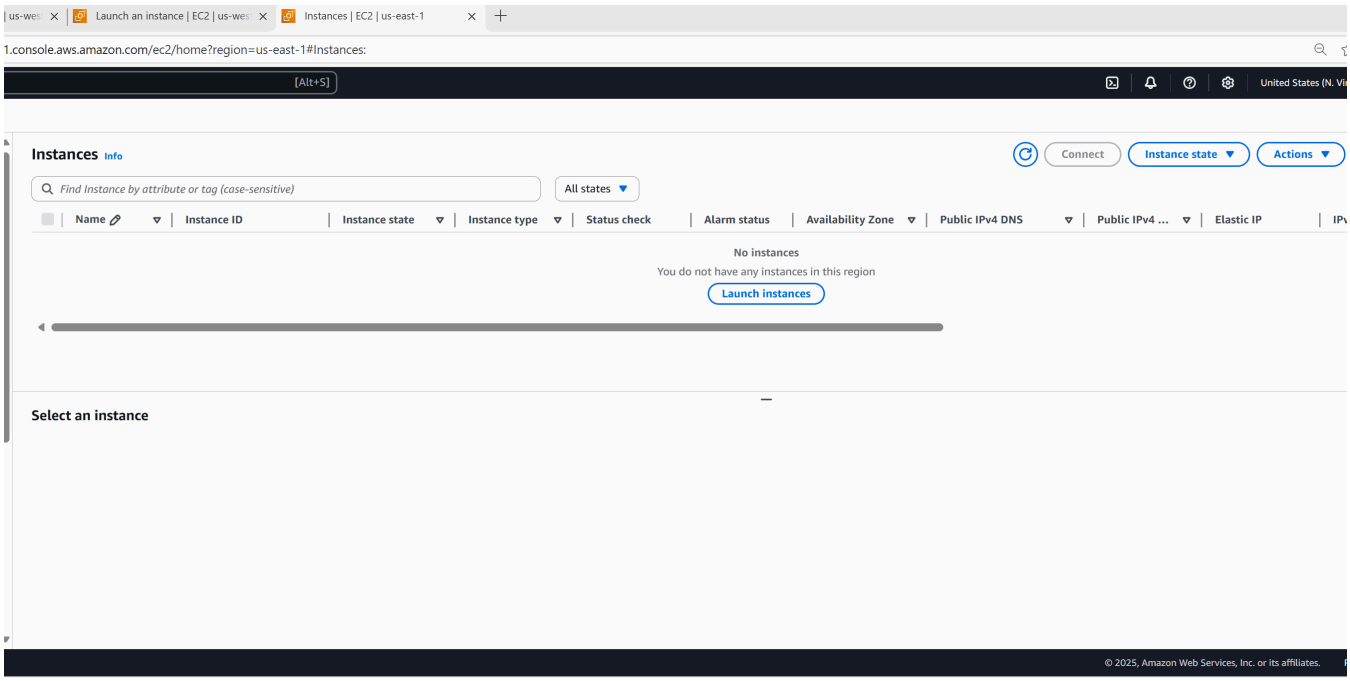
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

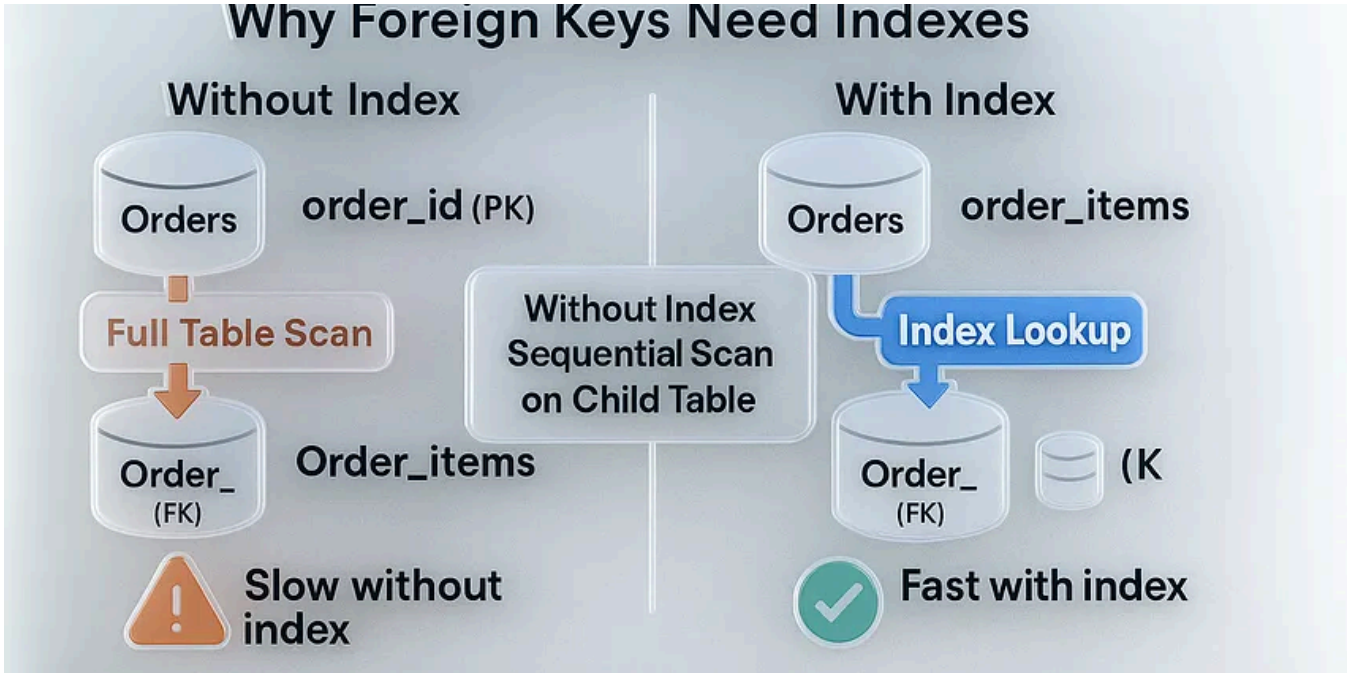


Jeyaram Ayyalusamy

Upgrading PostgreSQL from Version 16 to Version 17 Using pg_upgrade on a Linux Server AWS EC2...

🛠️ A Complete Step-by-Step Guide to Installing PostgreSQL 16 on a Linux Server (With Explanations)

Aug 4 40

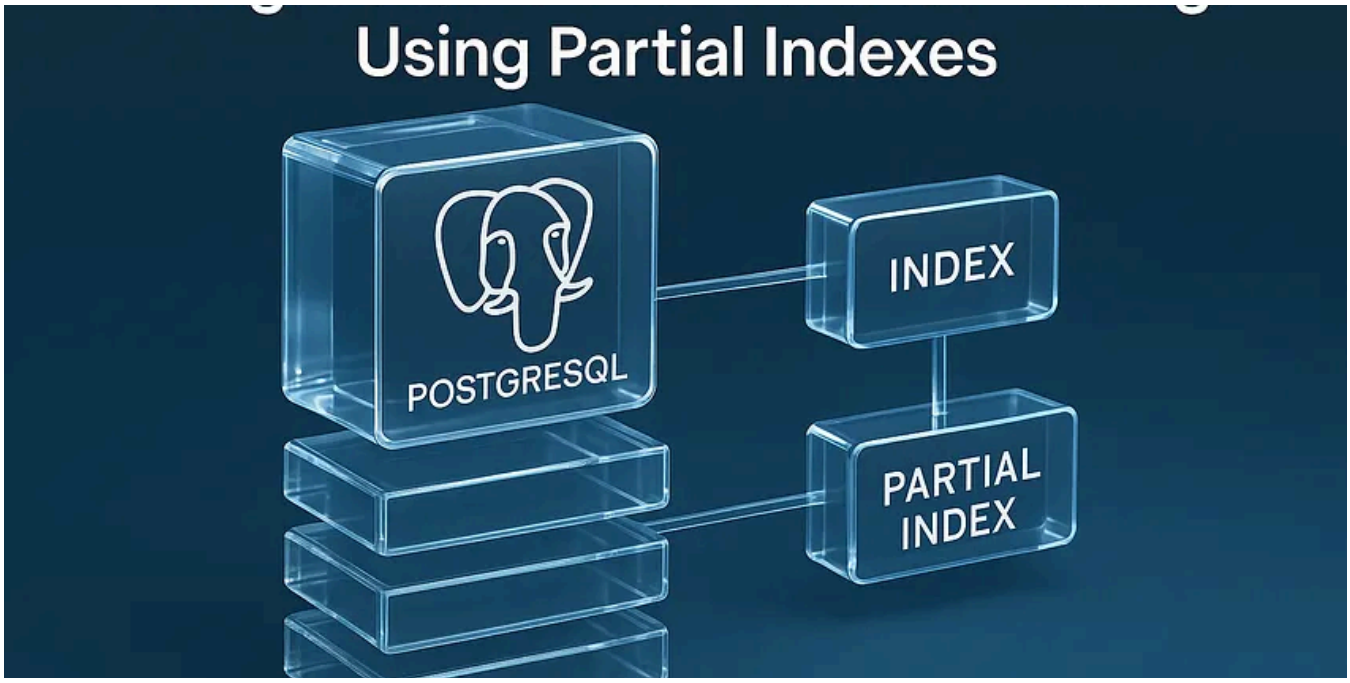


Jeyaram Ayyalusamy

16 -PostgreSQL 17 Performance Tuning: Why Foreign Keys Need Indexes

When designing relational databases, parent-child relationships are common. A classic case is the Orders table (parent) and the Items table...

Sep 3 3 2

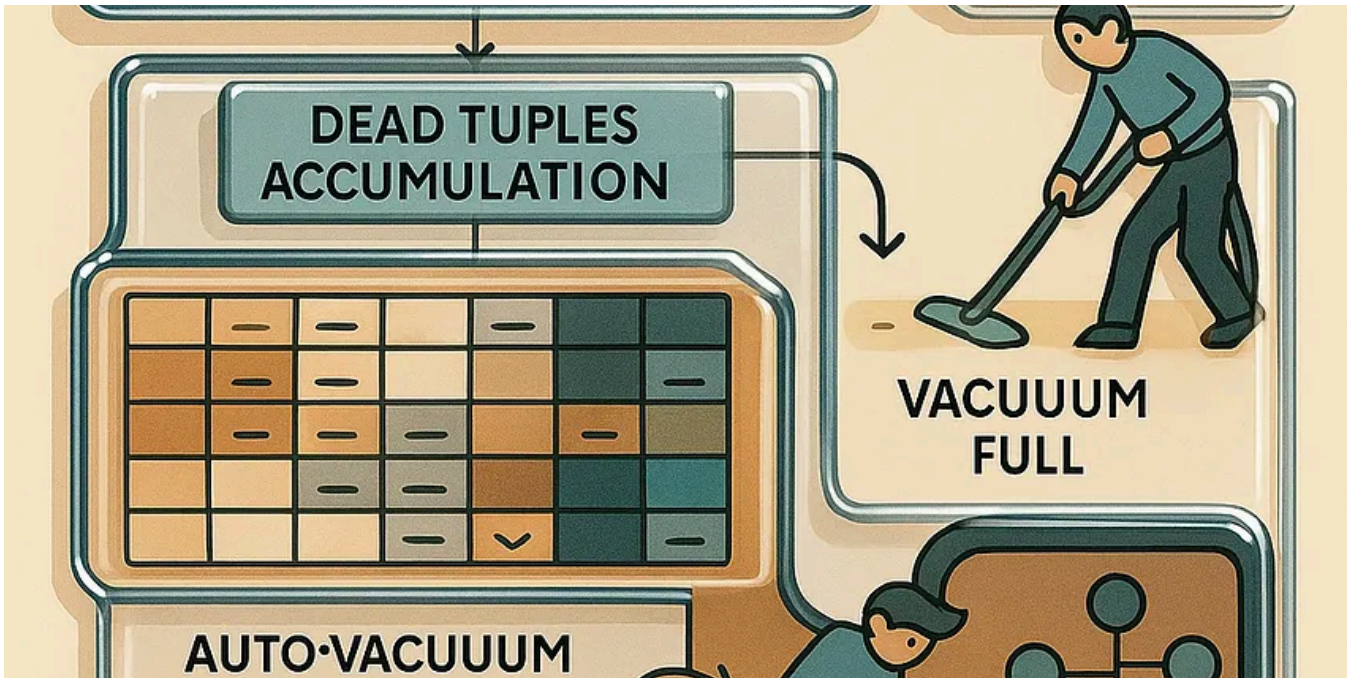


Jeyaram Ayyalusamy

17 - PostgreSQL 17 Performance Tuning: Using Partial Indexes

Even though we usually create indexes on all rows of a table, PostgreSQL also allows you to create an index on just a subset of records...

Sep 4 3



Jeyaram Ayyalusamy

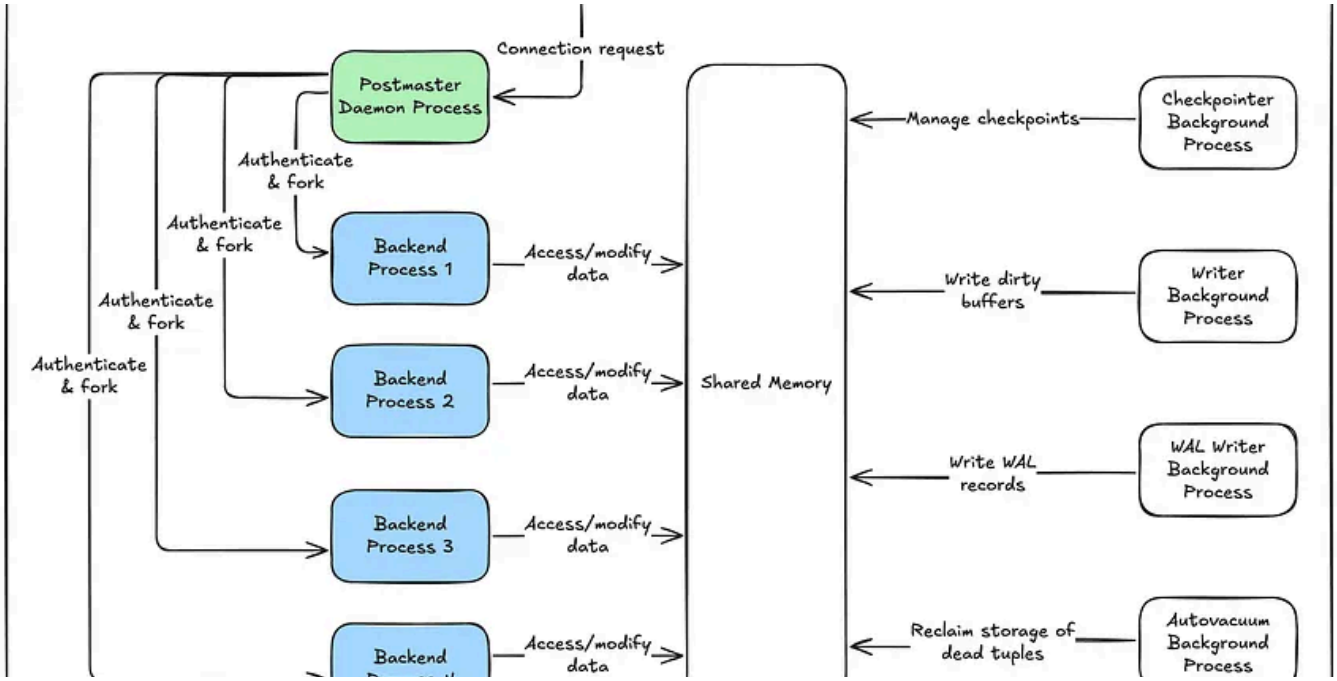
08-PostgreSQL 17: Complete Tuning Guide for VACUUM & AUTOVACUUM

PostgreSQL's MVCC design creates dead tuples during UPDATE/DELETE. VACUUM reclaims them; AUTOVACUUM schedules that work. Get these knobs...

Sep 1 26

See all from Jeyaram Ayyalusamy

Recommended from Medium

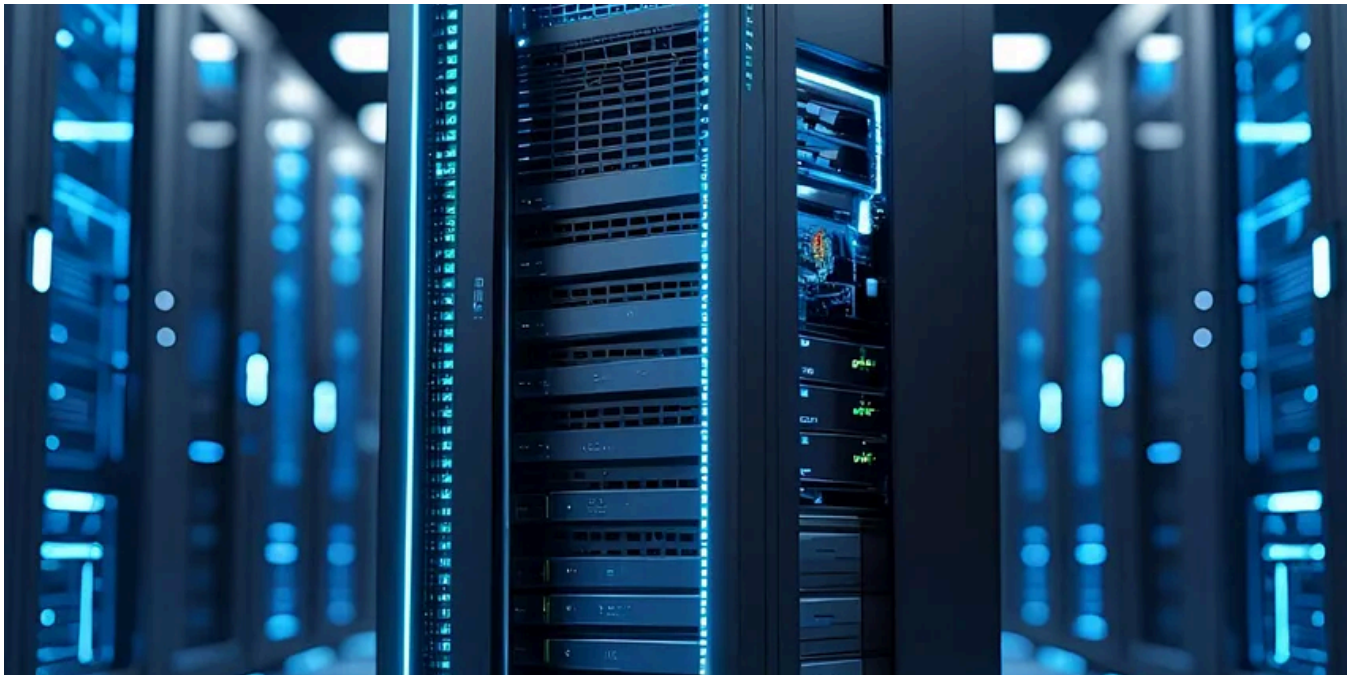


Kareem Mohllal

Database Connection Pooling with PgBouncer

Database connections are expensive, let's explore why and how PgBouncer can save your database from drowning in connections.

Aug 5 10



Rizqi Mulki

PostgreSQL Index Bloat: The Silent Killer of Performance

How a 10TB database became 3x faster by fixing the invisible problem that plagues 90% of production PostgreSQL deployments

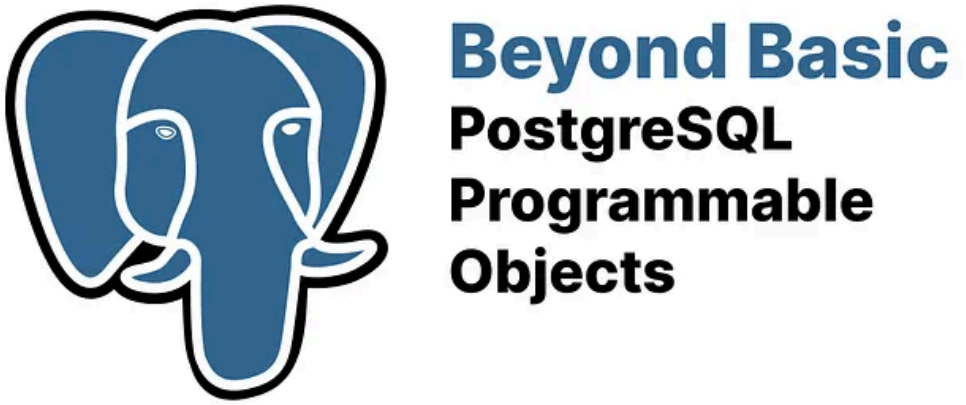
Sep 15 11 1

Tomasz Gintowt

Security in PostgreSQL

PostgreSQL is a powerful open-source database. Security is very important when working with sensitive data like passwords, customer...

6d ago 5



In Stackademic by bektiaw

Beyond Basics: PostgreSQL Programmable Objects (Automate, Optimize, Control)

Tired of repeating the same SQL logic? Learn how PostgreSQL can do the work for us.

Sep 1 68 1

Thinking Loop

5 DuckDB–Arrow–Polars Workflows in Minutes

Turn day-long pipelines into small, local, reproducible runs without clusters or drama.

5d ago 14

Ajaymaurya

Advance PostgreSQL Indexing: Optimize Queries for Faster Database Performance

Ever wondered why some database queries feel like lightning while others crawl like a snail? The secret often lies in how PostgreSQL...

Sep 8 2

See more recommendations