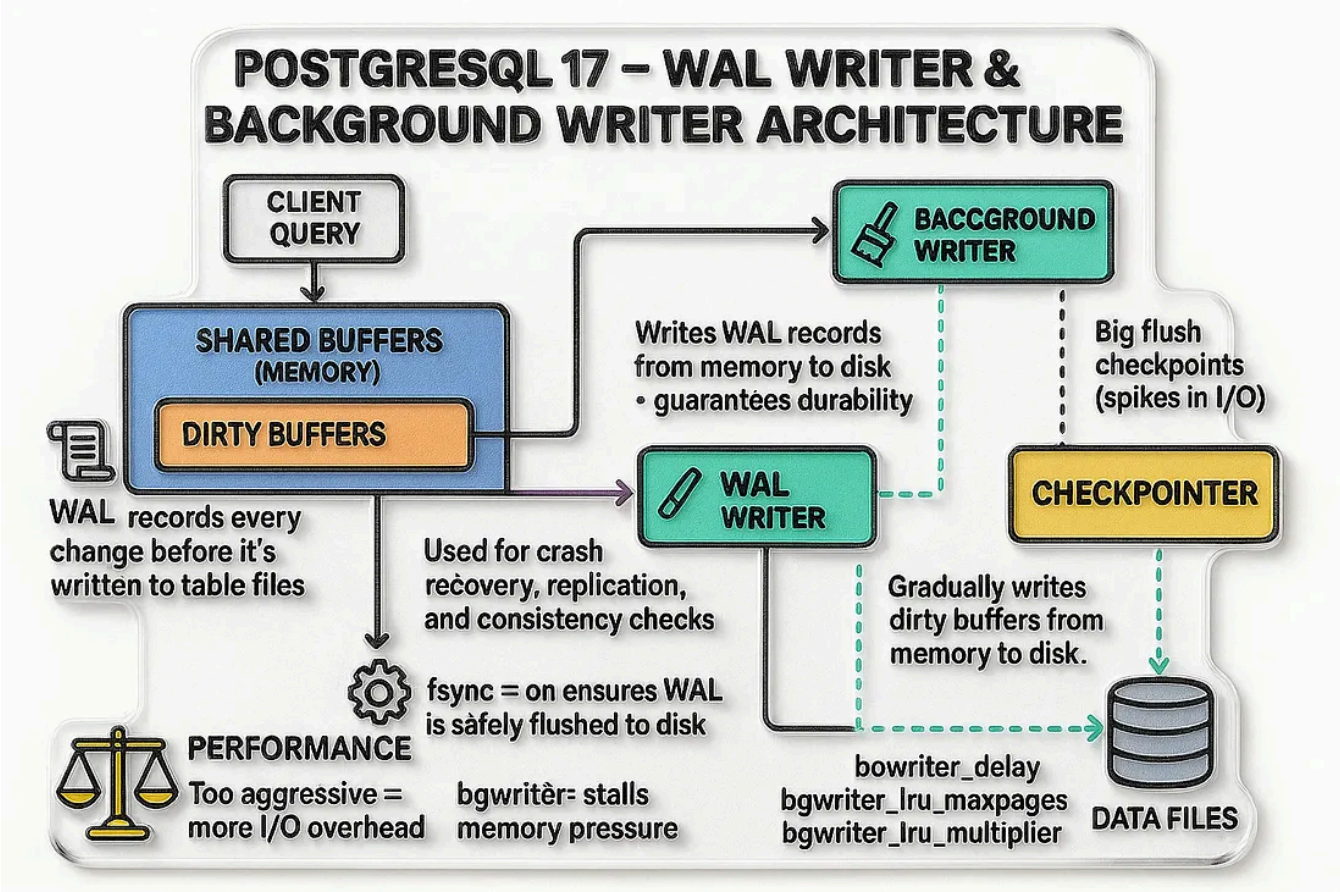
05 -PostgreSQL 17 Performance Tuning: The WAL Writer and Background Writer

24 min read · Aug 31, 2025

Jeyaram Ayyalusamy Following



PostgreSQL provides both **durability** (no committed data is lost after a crash) and **performance** (queries run fast). This balance is achieved through two important background processes:

- **The WAL Writer** → Its job is to make sure every change in the database is safely recorded in the Write-Ahead Log (WAL). This guarantees that even if the server crashes, PostgreSQL can recover committed transactions from the WAL.
- **The Background Writer** → Its job is to keep the database's memory buffers free and efficient. It does this by writing some dirty buffers (modified data in memory) to disk in the background, preventing memory from becoming full and slowing down queries.

Together, these two processes ensure that PostgreSQL remains both **reliable** (your data is safe) and **efficient** (your queries stay fast).

Step 1 — The Role of WAL (Write-Ahead Log)

One of the main reasons PostgreSQL is considered both **highly reliable** and **high-performing** is its use of the **Write-Ahead Log (WAL)**. Before understanding WAL-related background processes, let's first break down *what WAL is, why it exists, and how it works*.

What Is WAL?

The **Write-Ahead Log (WAL)** is a **sequential log** where PostgreSQL records **every single change** to the database.

- **Golden Rule of WAL:**
  All changes must be written to WAL before they are written to actual data files.

This ensures that PostgreSQL can always recover to a consistent state, even after a crash, because WAL holds the exact sequence of operations.

👉 WAL is the **foundation** of:

- Crash recovery
- Replication
- Durable commits

Why Does PostgreSQL Use WAL?

Without WAL, PostgreSQL would have to:

- Write every change **directly to the table files** on disk.
- Handle multiple random writes (slow).
- Risk incomplete writes in the event of a crash.

With WAL, PostgreSQL:

- Writes changes **sequentially** to the WAL file (fast and efficient).
- Delays updating the actual table files until background processes (like checkpoints).
- Guarantees durability because WAL can be replayed after crashes.

👉 Sequential writes = much faster than random I/O.

Step-by-Step: How WAL Works

Let's walk through an example:

```
UPDATE customers
SET email = 'new_email@example.com'
WHERE id = 101;
```

1: Change Is Made in Memory

- PostgreSQL updates the row (`id = 101`) in the **shared buffer** (memory).
- This change is not immediately reflected in the `customers` table file.

2: Record Is Written to WAL

- A WAL entry is created describing the update (e.g., `email: old → new`).
- This WAL record is written sequentially to the WAL buffer.

3: WAL Is Flushed at Commit

- When you issue `COMMIT`, PostgreSQL **flushes the WAL record to disk.**
- ✅ At this point, the transaction is considered *durable*.

4: Data File Updated Later

- The actual `customers` table file on disk may only be updated later — during a checkpoint or background writer flush.
- Even if the system crashes before this update, PostgreSQL can recover using WAL.

What Happens During a Crash?

Imagine a crash happens just after WAL is flushed, but before the data file is updated:

1. PostgreSQL restarts.
2. It finds the **last WAL record.**
3. It replays the WAL changes into the table file.
4. ✅ The committed transaction is restored.

👉 WAL ensures **no committed transaction is lost.**

WAL and Replication

Since PostgreSQL 9.0, WAL has also been used for **replication**:

- Every change recorded in WAL is shipped to a standby server.
- The standby replays WAL entries in real time, staying synchronized with the primary.
- This makes **high availability (HA)** possible.

Analogy: WAL as a Journal 📓

Think of WAL like your **personal diary**:

- Every time you make an important decision, you jot it down in the diary (WAL).
- Later, you update your official records (data files).
- If the official record is damaged or incomplete (crash), you can always reconstruct it from your diary.

👉 Without the diary, important details would be lost. WAL plays this protective role in PostgreSQL.

Example in Numbers

Let's say you update **1,000 rows** in a table:

- **Without WAL** → PostgreSQL would update 1,000 rows in the data file directly. That's 1,000 random writes.
- **With WAL** → PostgreSQL writes 1,000 changes sequentially to WAL (very fast). Later, the data file is updated in batches.

✅ Result → Transactions commit faster, recovery is safer, and I/O is more efficient.

✅ Key Takeaways

- WAL = **sequential log of changes.**
- **Rule:** WAL is always written before data files.
- Guarantees:
- **Durability** → No committed transaction is lost.
- **Recovery** → PostgreSQL can replay WAL after crashes.
- **Replication** → WAL streams changes to standby servers.
- Analogy: WAL = **journal** before updating the official record book.
- PostgreSQL 17 relies on WAL as the backbone of its crash safety and replication system.

⚡ By understanding WAL, you've learned the **foundation of PostgreSQL performance and durability.** Every tuning decision that follows builds on this concept.

Step 2 — The WAL Writer Process

We now know that PostgreSQL uses the **Write-Ahead Log (WAL)** as a journal to record every change before it's written to actual table files. But who ensures that this journal (WAL) is safely written from memory to disk? That's the job of the **WAL Writer**.

What Is the WAL Writer?

The **WAL Writer** is a **background process** in PostgreSQL. Its job is to:

1. Collect WAL records stored in memory (WAL buffers).

2. Write them **sequentially** to WAL segment files on disk.
3. Guarantee that committed transactions are **durable** — even if PostgreSQL crashes.

👉 You can think of the WAL Writer as a **scribe** who takes notes written on a scratchpad (memory) and copies them into a permanent ledger (disk).

Why the WAL Writer Matters

1. Ensures Crash Recovery

- Imagine PostgreSQL crashes right after you run `COMMIT`.
- The data file (table) may not have been updated yet.
- But if the WAL Writer has already flushed the WAL record → PostgreSQL can **replay WAL on restart** and recover the transaction.

✅ This means **no committed transaction is lost.**

2. Improves Performance

- Data files are scattered across disk → random writes are expensive.
- WAL is written **sequentially** → one entry after another → very efficient.
- Instead of updating 50 different tables in different places on disk, PostgreSQL simply appends 50 entries in WAL.

✅ Sequential writes = faster commits, especially under workloads with many small transactions.

Step-by-Step: How WAL Writer Works

Let's see how this plays out when a transaction is executed.

Example Query

```
UPDATE customers
SET email = 'new_email@example.com'
WHERE id = 101;
```

1: Update in Memory

- PostgreSQL modifies the row (`id=101`) inside the **shared buffer** (in-memory cache).
- The `customers` table file on disk is **not touched yet.**

2: WAL Entry Created

- PostgreSQL generates a WAL record describing this change:
- Table → `customers`
- Row → `id=101`
- Old value → `old_email@example.com`
- New value → `new_email@example.com`

This WAL record is placed in the **WAL buffer (memory).**

3: Commit → WAL Flushed

- When the user issues `COMMIT` , PostgreSQL flushes the WAL record from memory to the WAL file on disk.
- ✅ At this exact moment, the transaction is considered **durable.**
- Even if PostgreSQL crashes now, WAL has the change.

4: Data File Updated Later

- The actual `customers` table file may be updated minutes later — during a checkpoint or background writer flush.
- But WAL ensures recovery no matter what.

Example: Many Small Transactions

Imagine 100 users commit transactions within a few seconds.

- **Without WAL**
- PostgreSQL would need to update 100 different data files.
- Each write would happen in a random part of the disk → slow, chaotic I/O.
- **With WAL**
- Each transaction is appended sequentially to the WAL.
- WAL Writer flushes them efficiently in order.
- ✅ End result → a few sequential writes instead of 100 random ones.

Analogy: WAL Writer as a Mail Courier 📬

Think of a busy office:

- Employees (transactions) write letters.
- **Without WAL** → Each employee runs individually to the post office. The streets get crowded, and deliveries are inefficient.
- **With WAL** → Employees drop their letters in a central mailbox (WAL buffer). The courier (WAL Writer) picks up the letters and delivers them in bulk, neatly in order.

👉 This saves time, reduces chaos, and ensures that no letter is lost.

✅ Key Takeaways

- **The WAL Writer is a background process** that moves WAL entries from memory to disk.
- **Crash Safety** → Even if the data files aren't updated, WAL guarantees recovery of committed transactions.
- **Performance** → Sequential writes make handling many small transactions much faster.
- **Example** → 100 commits become a few efficient sequential writes, instead of 100 scattered disk operations.

⚡ In PostgreSQL 17, the WAL Writer continues to be a cornerstone of the database's **durability and performance model**. Without it, PostgreSQL wouldn't be able to guarantee both **fast transactions** and **safe crash recovery**.

Step 3 — When WAL Is Read

Most of the time, PostgreSQL **writes** to the Write-Ahead Log (WAL). But WAL isn't only about writing. There are a few critical situations where PostgreSQL must **read from WAL** to ensure durability, consistency, and replication. Let's break down these cases step by step.

1. Crash Recovery

When PostgreSQL crashes (power failure, hardware fault, or forced shutdown), data files may not contain the latest changes.

- **Problem:** Some transactions may have been committed, but their changes were only in memory (buffers) and not yet written to the data files.
- **Solution:** WAL contains the record of these committed changes. On restart, PostgreSQL replays WAL entries after the last checkpoint.

👉 **Example:**
You update a row in the `customers` table:

```
UPDATE customers
SET email = 'new_email@example.com'
WHERE id = 101;
```

- WAL records this change and flushes it to disk at commit.
- PostgreSQL crashes before the `customers` table file is updated.
- On restart, PostgreSQL **reads WAL**, replays the change, and ensures the email update is not lost.

✅ WAL guarantees that committed changes survive crashes.

2. Startup Consistency

Every time PostgreSQL starts, it checks WAL to confirm the database is in a **consistent state**.

- If PostgreSQL was shut down cleanly → all data files are in sync, and WAL replay isn't needed.
- If PostgreSQL was shut down abruptly → some pages might be partially written. WAL is used to:
- **Roll forward** → apply missing committed changes.
- **Roll back** → ignore incomplete transactions.

👉 **Example:**
Suppose PostgreSQL was importing thousands of rows when the server lost power.

- On startup, PostgreSQL scans WAL.
- Committed rows are reapplied.
- Incomplete ones are ignored.
- ✅ The database starts in a safe, consistent state without corruption.

3. Replication

Since **PostgreSQL 9.0**, WAL has been the foundation of **replication**.

- WAL changes from the primary server are streamed to standby servers.
- Standby servers replay WAL in real-time, applying the same changes as the primary.
- This makes **high availability (HA)** and **disaster recovery (DR)** possible.

👉 **Example:**
On the primary:

```
INSERT INTO orders (customer_id, total_amount)
VALUES (101, 250.00);
```

- WAL logs this insert.
- WAL is immediately shipped to the standby.
- The standby replays WAL and inserts the same row into its `orders` table.

✅ The standby stays in sync with the primary without directly processing the client's transaction.

⚠️ Critical Point: Losing WAL = Losing Transactions

WAL is the **first place** where changes are recorded.

- If WAL files are lost or corrupted before they are applied to data files, those changes cannot be recovered.
- This is why PostgreSQL DBAs often configure **WAL archiving**, **replication**, or both — to protect WAL as the most critical component of durability.

👉 WAL is not just a log; it's the lifeline of PostgreSQL's reliability.

Analogy: WAL as a Black Box Recorder ✈️

Think of WAL like the **black box in an airplane**:

- Every action (transaction) is logged in order.
- If the plane crashes, investigators (PostgreSQL) use the black box to reconstruct what happened.
- Without the black box, much of the data would be lost forever.

✅ Key Takeaways

- WAL is usually written to, but **read in three critical cases**:

1. **Crash Recovery** → Replays committed changes not yet flushed to data files.
2. **Startup** → Ensures the database starts in a consistent state, rolling forward or rolling back as needed.
3. **Replication** → Streams WAL to standby servers for near real-time synchronization.

- ⚠️ Losing WAL means losing committed transactions — protecting WAL files is essential.

⚡ In PostgreSQL 17, WAL isn't just about **writing changes safely**. It's equally critical when **reading WAL** for recovery, startup consistency, and replication — making it the backbone of PostgreSQL's reliability.

Step 4 — WAL and the `fsync` Setting

PostgreSQL's reputation for **reliability** comes from its careful handling of data safety. A key part of this system is the **`fsync` setting**, which directly controls how PostgreSQL guarantees that committed transactions are safe on disk.

What Is `fsync`?

`fsync` is a parameter that determines **how PostgreSQL interacts with the operating system and disk when writing WAL records.**

- **With `fsync = on` (default)** → PostgreSQL waits for confirmation from the operating system that WAL changes have been **physically written to disk** before completing a commit.
- **With `fsync = off`** → PostgreSQL does not wait. It assumes the operating system will eventually flush WAL to disk. This makes commits faster but introduces risk.

👉 In simple terms, `fsync` answers the question:
"Should PostgreSQL double-check that your data is really safe on disk before saying COMMIT is done?"

1: Behavior with `fsync = on` (Default)

This is the **safe mode** and the default in PostgreSQL 17.

- Every commit forces WAL records to be flushed to disk.

- PostgreSQL waits until the OS confirms the flush is successful.
- ✅ This guarantees **durability** — even if the server crashes immediately after a commit, the data is safe in WAL files.

Example: Safe Transactions

Suppose you commit 5 updates to the `customers` table:

```
UPDATE customers SET email = 'a1@test.com' WHERE id = 1;
UPDATE customers SET email = 'a2@test.com' WHERE id = 2;
UPDATE customers SET email = 'a3@test.com' WHERE id = 3;
UPDATE customers SET email = 'a4@test.com' WHERE id = 4;
UPDATE customers SET email = 'a5@test.com' WHERE id = 5;
```

- With `fsync = on`, each transaction forces WAL to disk.
- If PostgreSQL crashes after the 5th commit, all 5 updates are preserved.

✅ **Result:** Safe, reliable, and consistent database.

2: Behavior with `fsync = off`

This is the **fast but risky mode**.

- PostgreSQL does not force WAL to disk at every commit.
- The OS still writes WAL eventually, but PostgreSQL doesn't wait for confirmation.
- Transactions appear faster because PostgreSQL skips the waiting step.
- ❌ If the server crashes before WAL is flushed, committed transactions may be lost.

Example: Risky Transactions

Suppose the same 5 updates are committed with `fsync = off`:

- PostgreSQL considers them committed immediately, but WAL is still in memory.

- If the system crashes before the OS flushes WAL → some or all updates are gone.

❌ **Result:** Faster performance, but you risk losing "committed" data.

3: When Should You Disable `fsync`?

Disabling `fsync` should **almost never be done** in production. It is only safe in temporary, non-critical situations such as:

- **Development environments** → where data loss is acceptable.
- **Testing and benchmarking** → when you only care about raw speed.
- **Bulk data loading** → when you can easily reload the data if something goes wrong.

👉 In production systems (banking, e-commerce, healthcare, etc.), `fsync` must **always remain ON**.

4: Analogy — Saving a Document 📝

Think of `fsync` like saving a document on your laptop:

- **With `fsync = on`** → You press *Save* after each paragraph. Even if your laptop crashes, you only lose the last few keystrokes.
- **With `fsync = off`** → You type continuously but don't save until the end. It feels faster, but if your laptop crashes, you lose everything since the last save.

👉 PostgreSQL's `fsync` works the same way. Turning it off is like writing a book without saving — fast but dangerous.

5: Performance vs Safety

`fsync = on`

- ✅ Safe, durable, consistent.
- ❌ Slightly slower, especially for workloads with thousands of tiny commits.

```
fsync = off
```

- ✅ Faster commit speed.
- ❌ Risk of data loss or corruption on crash.

👉 PostgreSQL chooses **safety first by default** because for most applications, losing committed data is unacceptable.

✅ Key Takeaways

- `fsync` controls whether PostgreSQL **forces WAL changes to disk** at commit.
- **`fsync = on` (default)** → Guarantees durability, safest for production.
- **`fsync = off`** → Boosts performance but risks data loss, only safe for non-critical or temporary setups.
- Analogy: `fsync` is like pressing *Save* — turning it off is like working without saving.
- **Best Practice:** Never disable `fsync` in production.

⚡ In PostgreSQL 17, `fsync` continues to be one of the most important safety switches — protecting your database from corruption while balancing performance.

Step 5 — The Background Writer Process

PostgreSQL relies on different background processes to keep the database **safe** and **fast**. The **WAL Writer** ensures durability by logging all changes, but keeping the

system responsive requires another helper — the **Background Writer**. Its main job is to manage memory smoothly so queries don't run into bottlenecks.

1: Shared Buffers and Dirty Pages

- PostgreSQL has a memory area called **shared buffers**.
- Whenever a query reads or writes data, it works with these memory buffers instead of directly touching disk.
- When data inside a buffer is modified, that buffer becomes **dirty**.

👉 **Dirty buffer** = a memory page that has been changed but not yet written to disk.

If too many buffers are dirty, PostgreSQL risks running out of free memory to handle new queries.

2: Why Do Dirty Buffers Accumulate?

- PostgreSQL delays writing changes to disk for performance reasons.
- Instead of writing every small change immediately, it keeps them in shared buffers for a while.
- Over time, especially in a write-heavy workload, thousands of dirty buffers can accumulate.

👉 This is efficient in the short term, but if buffers fill up, PostgreSQL has no space left to process new queries.

3: The Role of the Background Writer

This is where the **Background Writer** comes into action:

- It runs continuously in the background.

- It **periodically scans shared buffers**.
- When it finds dirty buffers, it writes a batch of them to disk.
- Once written, those buffers are marked as **clean** and become reusable by new queries.

✅ This ensures there's always space in memory for upcoming transactions.

4: What If There Was No Background Writer?

Without the Background Writer:

- PostgreSQL would wait until a **checkpoint** to write dirty buffers.
- During the checkpoint, **all dirty buffers** would need to be written at once.

This could cause:

- Huge I/O spikes.
- Queries pausing or stalling while buffers are flushed.

👉 By spreading the work gradually, the Background Writer avoids these sudden bottlenecks.

5: Real-World Example

Let's say your server has **16 GB of shared buffers**.

- Heavy user activity dirties thousands of buffers in just a few minutes.
- If PostgreSQL waits for the next checkpoint, buffers might fill up completely.
- New queries would then be forced to wait for free buffers, slowing down response times.

With the Background Writer:

- It writes out dirty buffers in smaller batches.
- Free buffers are always available.
- ✅ Queries continue running smoothly, even during peak load.

6: Analogy — The Cleaning Crew in a Restaurant 🧹🍽️

Think of PostgreSQL's shared buffers as a busy restaurant kitchen:

- Chefs (queries) keep using pots and pans (buffers) to cook meals (transactions).
- Every pot used becomes dirty (dirty buffer).
- If nobody washes dishes until the end of the day (checkpoint), the kitchen will run out of pots during lunch rush.
- The cleaning crew (Background Writer) works throughout the day, washing dishes in small batches.

👉 This way, chefs always have clean pots, and the kitchen never slows down.

7: Benefits of the Background Writer

- **Smooth memory management** → Keeps buffers from filling up with dirty pages.
- **Reduced I/O spikes** → Writes are spread out instead of bunched at checkpoints.
- **Better query performance** → Queries don't stall waiting for free buffers.
- **Continuous efficiency** → Works silently in the background, without blocking user queries.

✅ Key Takeaways

- The **Background Writer** ensures PostgreSQL has enough clean buffers ready for new queries.

- It prevents shared buffers from becoming overloaded with dirty pages.
- **Without it** → Checkpoints would cause I/O spikes and query delays.
- **With it** → Writes are spread gradually, keeping performance stable.
- **Example**: In a 16 GB shared buffer system under heavy writes, the Background Writer flushes batches of dirty buffers early so queries don't get stuck waiting.
- **Analogy**: Like a cleaning crew in a restaurant, it keeps resources available by working continuously instead of waiting for the end of the day.

⚡ In PostgreSQL 17, the Background Writer is a critical process for **performance stability**. It doesn't make transactions durable (that's WAL's job), but it makes sure the system always has free memory to handle new work — keeping your database responsive even during heavy loads.

Step 6 — Difference Between Checkpointer and Background Writer

PostgreSQL is built to balance **durability** (keeping your data safe) and **performance** (keeping queries fast). Two important background processes help manage dirty buffers in memory: the **Checkpointer** and the **Background Writer**.

At first, they might sound like they do the same thing — writing dirty buffers to disk — but their roles are actually different. Together, they ensure PostgreSQL runs smoothly without slowing down under heavy workloads.

1: The Checkpointer

The **Checkpointer** is triggered during a **checkpoint event**.

- Its job: **write all dirty buffers** from shared memory (shared buffers) to disk.
- After this, PostgreSQL marks the point in the Write-Ahead Log (WAL) as safe.

- ✅ This ensures that if the database crashes, recovery can start from the last checkpoint instead of replaying the entire WAL history.

👉 **Key Point:** The Checkpointer guarantees **durability** and **faster crash recovery**.

Example:

- Suppose 10,000 buffers in shared memory have been modified.
- At a checkpoint, the Checkpointer writes **all 10,000** dirty buffers to disk.
- Now, PostgreSQL knows that everything up to this checkpoint is permanent.
- If a crash happens right after, recovery starts from this checkpoint.

2: The Background Writer

The **Background Writer** works differently.

- It runs **regularly at short intervals** (based on settings).
- Instead of writing all dirty buffers, it writes **just a portion** of them.
- Its goal is not durability but **smooth memory management**.
- By flushing some dirty buffers to disk, it makes sure new queries always find free memory space in shared buffers.

👉 **Key Point:** The Background Writer prevents **query stalls** by cleaning up memory gradually.

Example:

- Out of 10,000 dirty buffers in memory, the Background Writer might write 500 to disk during one cycle.
- This frees up space for new queries without overwhelming the disk.
- Over time, it keeps the memory buffer pool balanced.

3: How They Work Together

PostgreSQL needs **both** processes:

- The **Background Writer** → keeps memory free and avoids query stalls.
- The **Checkpointer** → guarantees durability and defines safe recovery points.

Combined Workflow Example:

- Heavy workload dirties 20,000 buffers in a 16 GB shared buffer pool.
- The Background Writer starts writing small chunks (e.g., 1,000 at a time) to disk in the background.
- When a checkpoint occurs, the Checkpointer writes the **remaining dirty buffers** and ensures all changes up to that point are safe.

✅ Together, they prevent PostgreSQL from stalling due to full memory and reduce crash recovery time.

4: Analogy — Kitchen Cleanup 🍳

Think of a busy restaurant kitchen:

- **Background Writer** → Like a dishwasher who cleans a few pots and pans during service. Chefs never run out of clean cookware, and work continues smoothly.
- **Checkpointer** → Like the end-of-day cleaning shift. The whole kitchen is scrubbed, every pot is washed, and everything is reset for the next day.

👉 Without the dishwasher (Background Writer), the kitchen would stall mid-service.
👉 Without the end-of-day cleaning (Checkpointer), the kitchen would not be ready for tomorrow.

5: Benefits of Each

Checkpointer

- ✅ Ensures data safety.
- ✅ Reduces crash recovery time.
- ❌ Can cause large I/O spikes if many dirty buffers are written at once.

Background Writer

- ✅ Keeps free buffers available for queries.
- ✅ Spreads I/O load gradually.
- ❌ Doesn't guarantee durability — it just helps with smooth performance.

✅ Key Takeaways

- The **Checkpointer** writes **all dirty buffers** during a checkpoint → ensures durability and faster recovery.
- The **Background Writer** writes **a subset of dirty buffers** at regular intervals → ensures smooth performance and avoids query stalls.
- **Together** → They balance durability and performance, preventing PostgreSQL from stalling under load.
- Example: With a 16 GB shared buffer pool and heavy writes, the Background Writer prevents stalls by cleaning up early, while the Checkpointer guarantees data safety at checkpoints.
- Analogy: The Background Writer is the **daytime cleaner**, and the Checkpointer is the **end-of-day deep clean**.

⚡ In PostgreSQL 17, tuning both processes properly is key to achieving **stable performance** while still guaranteeing **safe recovery after crashes**.

Step 7 — Impact on Performance

The **Background Writer** is a key process in PostgreSQL that helps manage dirty buffers in memory. By writing them out to disk gradually, it reduces stalls and keeps queries running smoothly. But like many optimizations, this comes with both **benefits** and **drawbacks**. Let's break them down.

1: The Positive Impact

1. Keeps Free Buffers Available

PostgreSQL stores frequently accessed data in **shared buffers**. As queries modify rows, these buffers become dirty. If too many buffers are dirty at once, PostgreSQL may run out of space for new queries.

- The Background Writer prevents this by flushing dirty buffers to disk in smaller batches.
- This frees up clean buffers for incoming queries.

👉 **Example:**
On a server with **16 GB shared buffers**, a heavy workload dirties thousands of buffers. Without the Background Writer, memory fills quickly, and queries stall while waiting for free buffers. With the Background Writer, buffers are freed steadily, so queries don't stop.

2. Avoids Query Slowdowns

If PostgreSQL waited for checkpoints to flush all dirty buffers, you'd see sudden pauses when the checkpoint runs.

- The Background Writer spreads the work out over time.
- This avoids sudden I/O spikes and keeps query performance consistent.

👉 **Think of it like a restaurant kitchen:** Instead of washing all the dishes at the end of the day (checkpoint), the staff (Background Writer) washes a few every hour. The

chefs (queries) always have clean plates to use.

2: The Negative Impact

1. Increased Total I/O Load

Because the Background Writer works continuously, it can sometimes write the **same buffer multiple times**.

- For example:
- A buffer is dirtied by a transaction.
- The Background Writer writes it to disk.
- Before the next checkpoint, another transaction dirties it again.
- It gets written twice.

✅ Safe, but ❌ less efficient.

👉 **Example:**
Suppose your workload frequently updates customer balances. The same row might be updated 10 times in a short period. With an aggressive Background Writer, each update could trigger a write, leading to extra I/O.

2. Risk of I/O Spikes

If the Background Writer is tuned too aggressively, it may flush too many buffers at once.

- This increases disk activity even when queries don't need it.
- Monitoring tools may show high disk writes, even when user activity looks normal.

👉 **Example:**
In a write-heavy system, if `bgwriter_lru_maxpages` (controls how many pages the

Background Writer writes at once) is set too high, you might see spikes of I/O every few seconds, reducing throughput for queries.

3: Analogy — Cleaning Too Often 🧹

Think of the Background Writer as cleaning staff in a busy office:

- **Positive side:** They clean desks regularly, so employees (queries) always have space to work.
- **Negative side:** If they clean the same desk 5 times in an hour, they waste energy and distract employees unnecessarily.

👉 The goal is balance — clean often enough to keep things running, but not so much that resources are wasted.

4: How to Spot Issues

- **Symptom:** Queries run smoothly, but you notice **sudden I/O spikes** in monitoring tools.
- **Likely cause:** Background Writer is writing too aggressively.
- **Fix:** Tune parameters such as `bgwriter_delay` (time between runs) and `bgwriter_lru_maxpages` (pages written per run) to reduce unnecessary writes.

✅ Key Takeaways

Positive Impact:

- Keeps free buffers available.
- Prevents query stalls.
- Spreads out I/O to avoid checkpoint bottlenecks.

Negative Impact:

- May increase total I/O load (same buffer written multiple times).
- Can cause I/O spikes if tuned too aggressively.

👉 Rule of Thumb: The Background Writer should smooth performance, not overwhelm the disk.

⚡ In PostgreSQL 17, the Background Writer is still a balancing act. When tuned correctly, it prevents slowdowns and keeps workloads smooth. When tuned poorly, it can create **extra disk activity** that hurts overall performance.

Step 8 — Parameters for Background Writer

The **Background Writer** is like PostgreSQL's "housekeeping service." It keeps memory (shared buffers) available by flushing dirty buffers to disk before they pile up. To get the best balance between performance and I/O usage, PostgreSQL gives us parameters that control how aggressively the Background Writer works.

You can check their current values using:

```
SELECT *
FROM pg_settings
WHERE name LIKE 'bgwriter%';
```

Let's go step by step through the key parameters.

1: `bgwriter_delay`

- **Definition:** The amount of time (in milliseconds) the Background Writer sleeps before waking up to do another cycle.
- **Default:** `200 ms` (0.2 seconds).

👉 **How it works:**

- **Lower value** = Background Writer runs more frequently.
- ✅ Keeps buffers clean more consistently.
- ✅ Reduces the chance of query stalls.
- ❌ Increases I/O activity, since it wakes up more often.
- **Higher value** = Background Writer runs less frequently.
- ✅ Fewer disk writes overall.
- ❌ Risk of running out of clean buffers during heavy workloads.

Example:

- On a small system with light writes → default 200 ms is fine.
- On a write-heavy OLTP system with bursts of transactions → lowering it to **100 ms** makes the Background Writer more responsive, preventing stalls when many users update data at the same time.

👉 **Analogy:** Imagine a janitor in an office. If they check desks every 20 minutes (200 ms), things stay tidy. If the office gets messy faster than that, they may need to check every 10 minutes (100 ms).

2: `bgwriter_lru_maxpages`

- **Definition:** Maximum number of buffers the Background Writer can write to disk in each cycle.
- **Default:** `100` .

👉 **How it works:**

- If **too low** → Background Writer may not write enough dirty buffers per cycle.
- ✅ Lower disk activity.
- ❌ PostgreSQL may run out of free buffers, forcing queries to wait.
- If **too high** → Background Writer flushes a lot of buffers per cycle.
- ✅ Plenty of clean buffers available.
- ❌ Can cause unnecessary I/O and slow down other operations.

Example:

- On a system with **16 GB shared buffers** and heavy write workloads, raising `bgwriter_lru_maxpages` above 100 ensures the Background Writer can keep up.
- But if you notice sudden I/O spikes in monitoring, the value may be set too high, and you should reduce it.

👉 **Analogy:** Think of a dishwasher in a restaurant:

- If they only wash 10 plates per cycle, the chefs may run out of clean dishes.
- If they try to wash 1,000 plates at once, they overload the system, wasting water and energy.

3: Balancing the Two

- **`bgwriter_delay`** = *how often the janitor checks desks.*
- **`bgwriter_lru_maxpages`** = *how many desks they clean each time.*

The right balance ensures:

- Memory doesn't fill up with dirty buffers.
- Queries don't stall waiting for free buffers.
- Disk I/O stays smooth without overwhelming the system.

Step 4: Example Configuration in PostgreSQL 17

Let's say you have a busy e-commerce database:

- **Shared Buffers:** 16 GB.
- **Workload:** Thousands of small updates per second.

A possible tuning might be:

```
bgwriter_delay = 100ms          # Check more frequently
bgwriter_lru_maxpages = 500     # Write more buffers per cycle
```

👉 This ensures dirty buffers don't build up and queries don't stall, but still spreads writes gradually to avoid I/O spikes.

✅ Key Takeaways

- The Background Writer is tunable with key parameters.
- `bgwriter_delay` controls how often it runs.
- Lower = smoother, more I/O.
- Higher = less I/O, risk of stalls.
- `bgwriter_lru_maxpages` controls how many buffers it writes per cycle.
- Too low = not enough free buffers.
- Too high = extra I/O load.
- The right settings depend on workload: high-write systems need more aggressive tuning, while light workloads can stick to defaults.

⚡ In PostgreSQL 17, tuning the Background Writer with these parameters helps strike the balance between **performance stability** and **efficient I/O usage**, ensuring your queries run smoothly under heavy load.

Step 9 — Analogy to Real Life

PostgreSQL's background processes like **WAL Writer** and **Background Writer** are powerful but can seem abstract when described technically. To make them easier to understand, let's compare them to **real-life habits** — things we all do, often without thinking.

1: WAL Writer → Writing in a Diary 📓

Imagine you are someone who records every important decision in a diary before making it official.

- **Why do you do this?**
  Because if your main record book (like official reports or business files) gets destroyed, you still have your diary to reconstruct what happened.

👉 In PostgreSQL:

- The **official record book** = **data files** on disk.
- The **diary** = **Write-Ahead Log (WAL)**.
- The **WAL Writer** ensures every change (insert, update, delete) is written to WAL before it ever touches the data files.

Real-World Example:

- You decide to move $500 from your savings account to your checking account.
- The diary entry (WAL) says: "Transferred $500 from savings to checking."
- Later, the official bank ledger (data file) is updated.

- If the ledger is lost due to a system crash, the diary can be used to replay and confirm the transfer.

✅ **Lesson:** The WAL Writer ensures **durability** — no committed transaction is ever forgotten, even if the official records are temporarily lost.

2: Background Writer → Tidying Up Your Desk 🧹

Now think about your work desk.

- If you keep piling up papers without tidying, eventually your desk gets so messy you can't find space to work.
- If you tidy up too often, you waste time putting papers away that you might need again in 5 minutes.

👉 In PostgreSQL:

- The **desk** = shared buffers (memory).
- The **papers** = dirty buffers (data modified in memory but not yet written to disk).
- The **Background Writer** = the act of tidying. It cleans up gradually, writing some dirty buffers to disk so there's always room for new queries.

Real-World Example:

- Imagine you're in a busy office, and you receive 50 new documents every hour.
- If you wait until the end of the day to clean your desk (like a checkpoint), you might run out of space during the day.
- Instead, you tidy a little every hour (Background Writer), so you always have room to keep working.

✅ **Lesson:** The Background Writer ensures **performance stability** by keeping memory from filling up. But if it tidies too often, it may waste effort (extra disk writes).

3: How They Work Together

Both processes are essential, but for different reasons:

- **WAL Writer (Diary)** → Makes sure no decision is ever lost. Even if disaster strikes, you can always look back at the diary and recover.
- **Background Writer (Desk Tidying)** → Keeps the work environment functional. Without it, you'd run out of space and stall.

Combined Example:

- You write every decision in your diary first (WAL Writer).
- At the same time, you tidy your desk throughout the day (Background Writer).
- At the end of the day, your official report is up to date and your desk is still usable.

Together, they make you both **organized (efficient)** and **safe from disaster (durable)**.

4: Easy-to-Grasp Summary

WAL Writer = Diary

- Always records changes first.
- Guarantees recovery after crashes.
- Focus: **Durability**.
- **Background Writer = Desk Cleaning**
- Keeps space free by tidying gradually.
- Prevents slowdowns when memory fills.
- Focus: **Performance**.

👉 Without the diary (WAL Writer), you risk losing important information forever.
👉 Without tidying (Background Writer), you risk running out of workspace.

✅ Key Takeaways

WAL Writer

- Records changes sequentially in WAL.
- Guarantees safe recovery.
- Critical for replication and crash safety.

Background Writer

- Periodically writes dirty buffers to disk.
- Prevents buffer cache from filling up.
- Can increase I/O if tuned poorly.

👉 Together, these two processes are the backbone of PostgreSQL's durability and performance model in version 17.

📢 Stay Updated with Daily PostgreSQL & Cloud Tips!

If you've been finding my blog posts helpful and want to stay ahead with daily insights on PostgreSQL, Cloud Infrastructure, Performance Tuning, and DBA Best Practices — I invite you to subscribe to my Medium account.

🔔 Subscribe here 👉 https://medium.com/@jramcloud1/subscribe

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 Oracle Open Source Cloud Computing

J

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

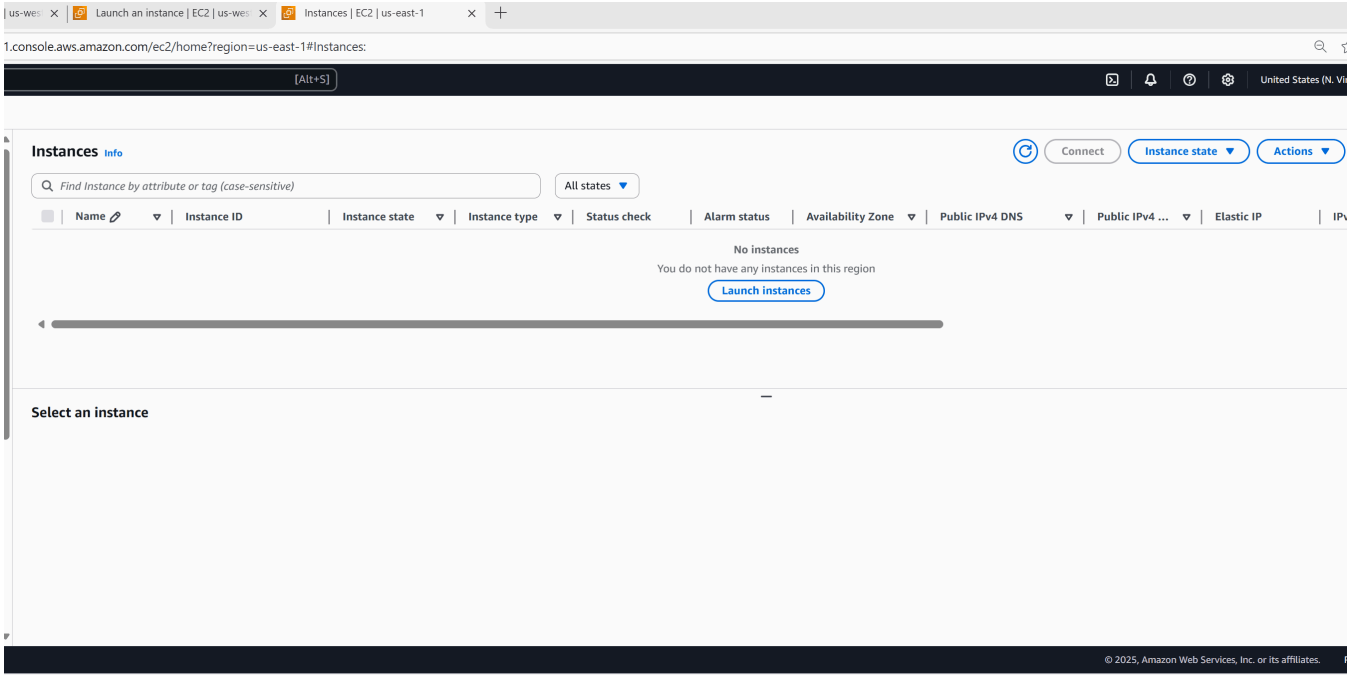
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

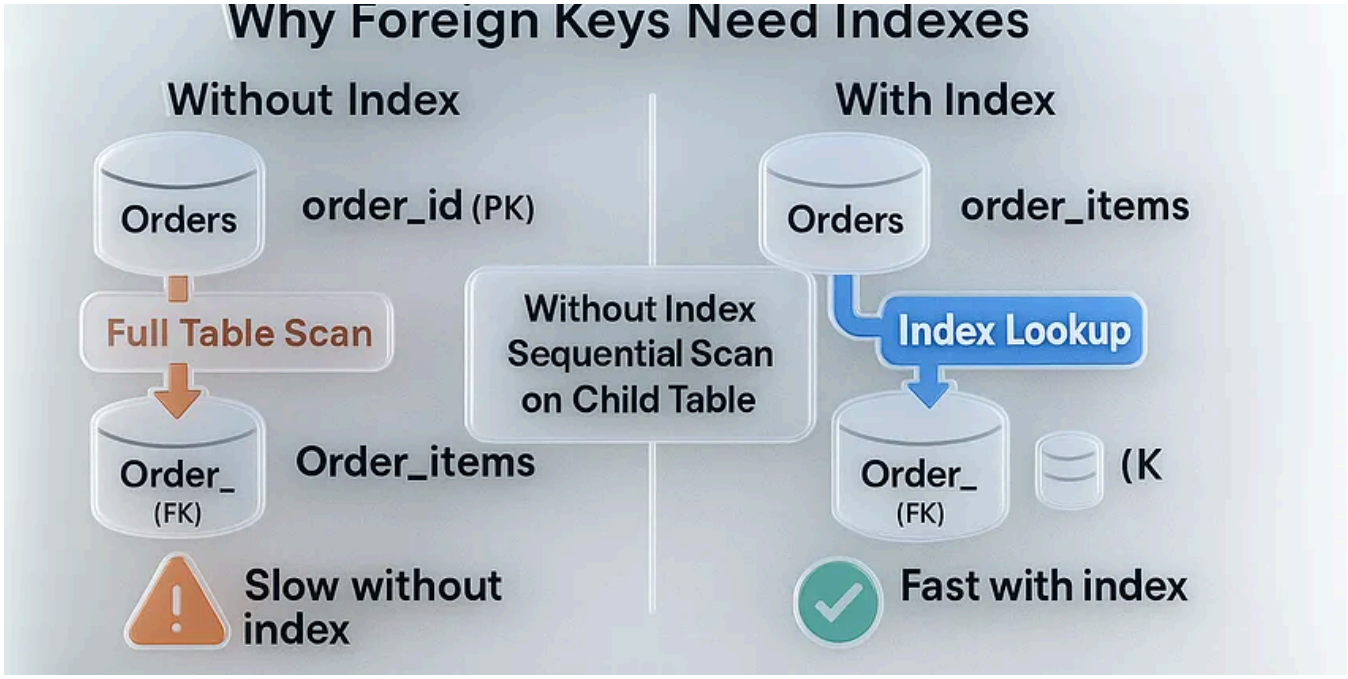


Jeyaram Ayyalusamy

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

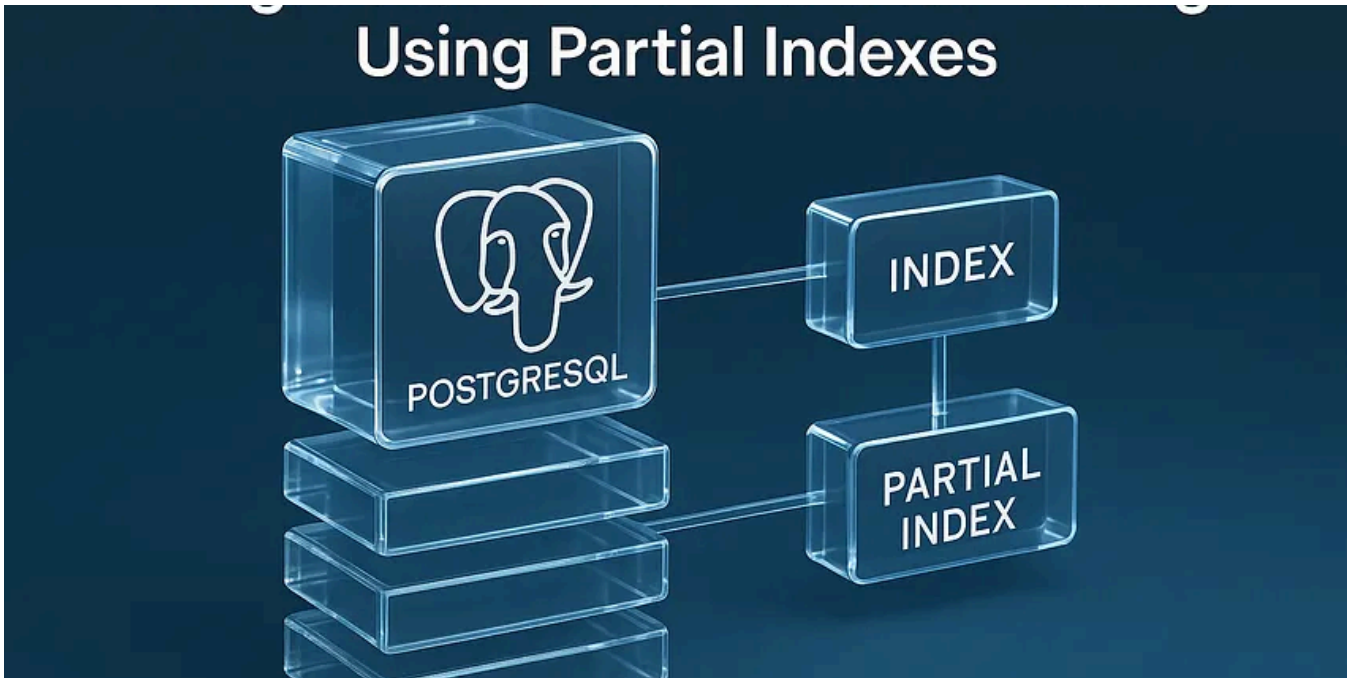


Jeyaram Ayyalusamy

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

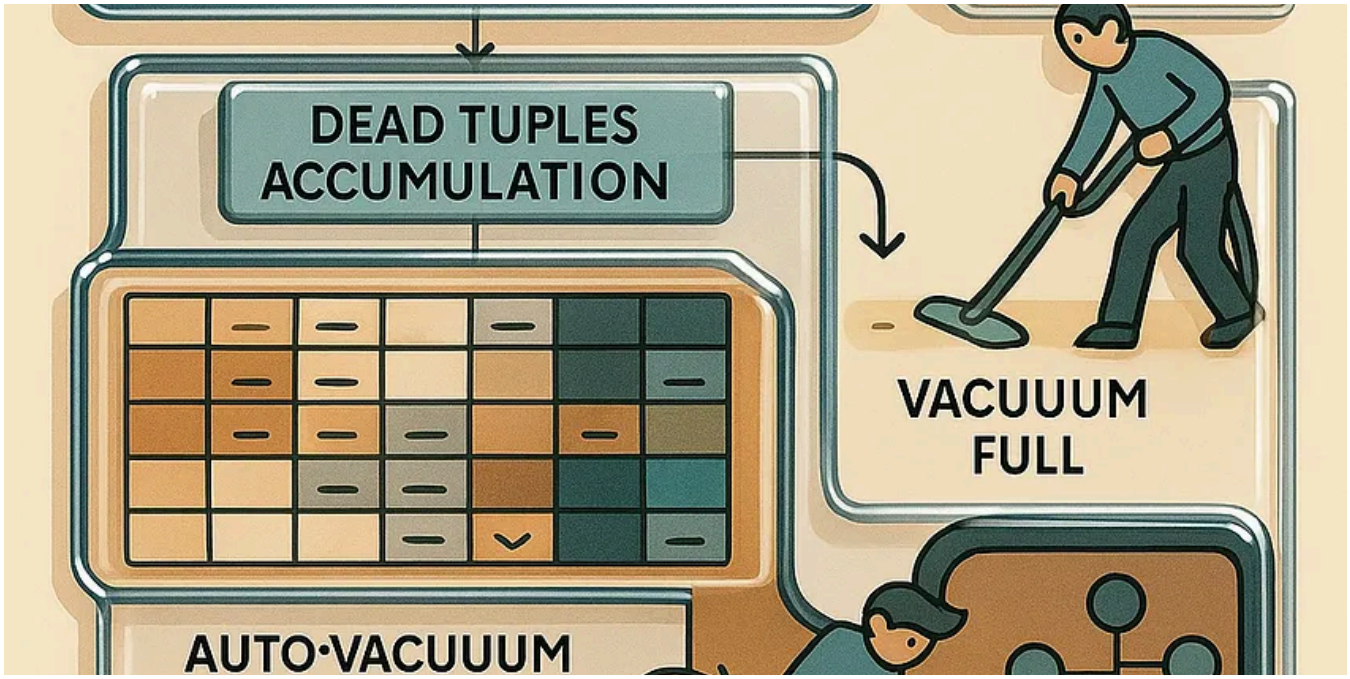


Jeyaram Ayyalusamy

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

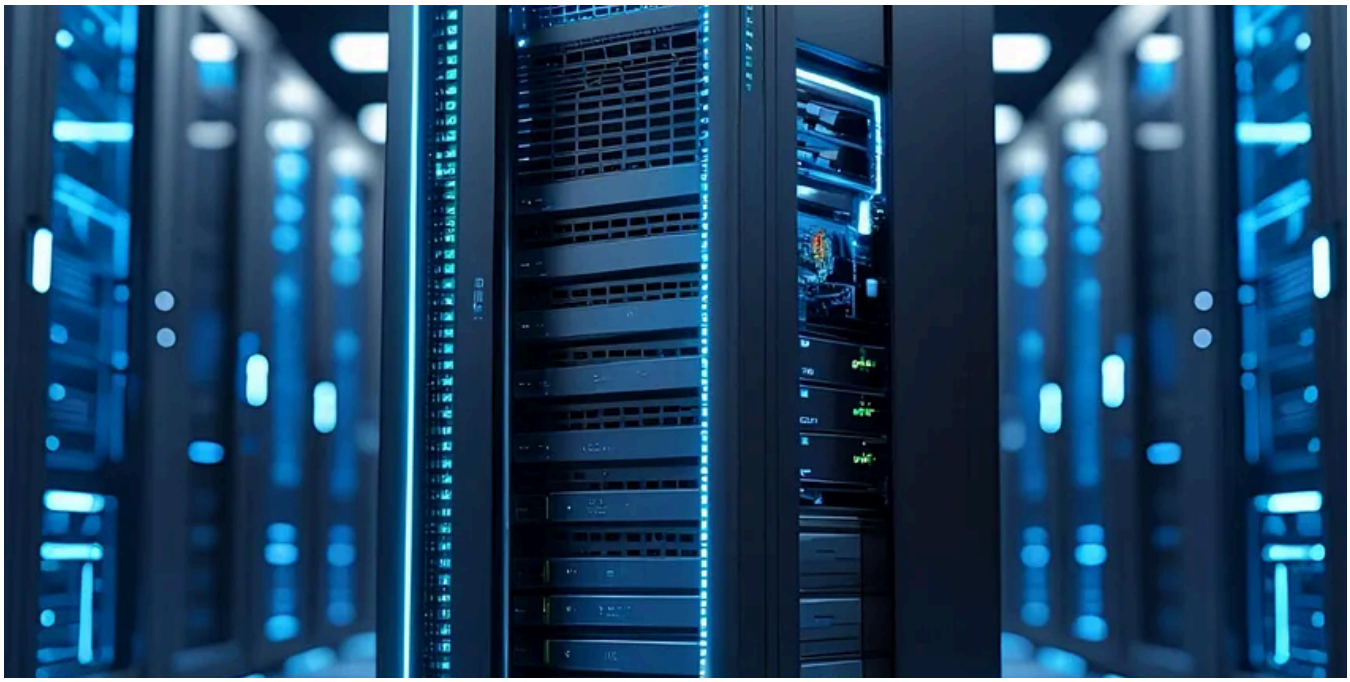
Recommended from Medium

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

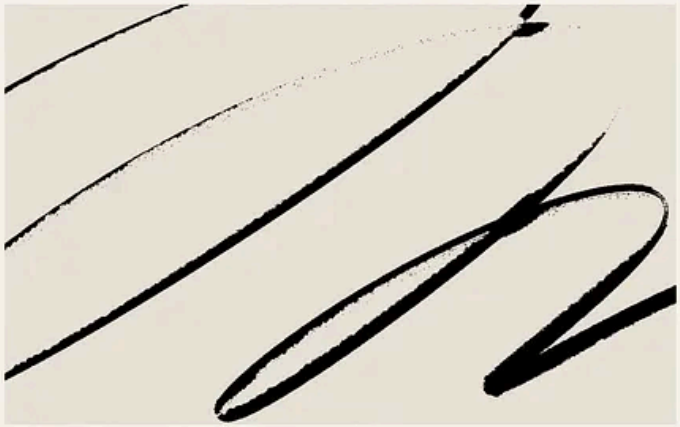


Rizqi Mulki

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

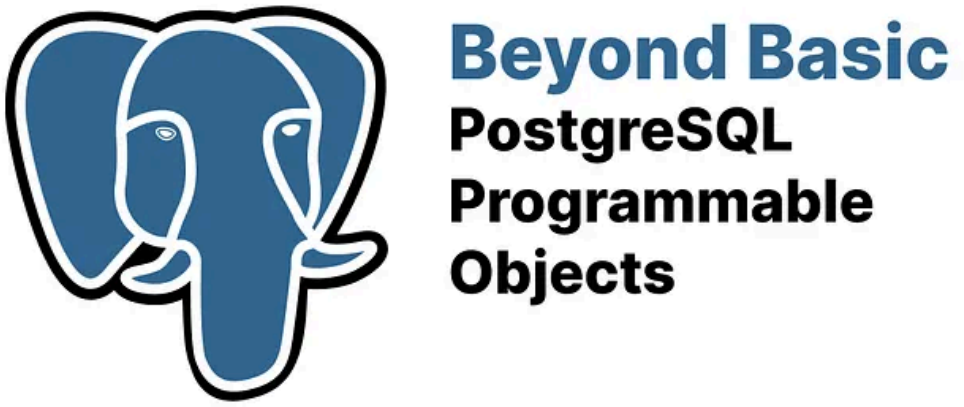


Vahid Yousefzadeh

Real-Time Statistics in Oracle 19c

Until Oracle version 12c, executing DML statements on a table (either using the conventional method or direct-path) would not update the...

Aug 13

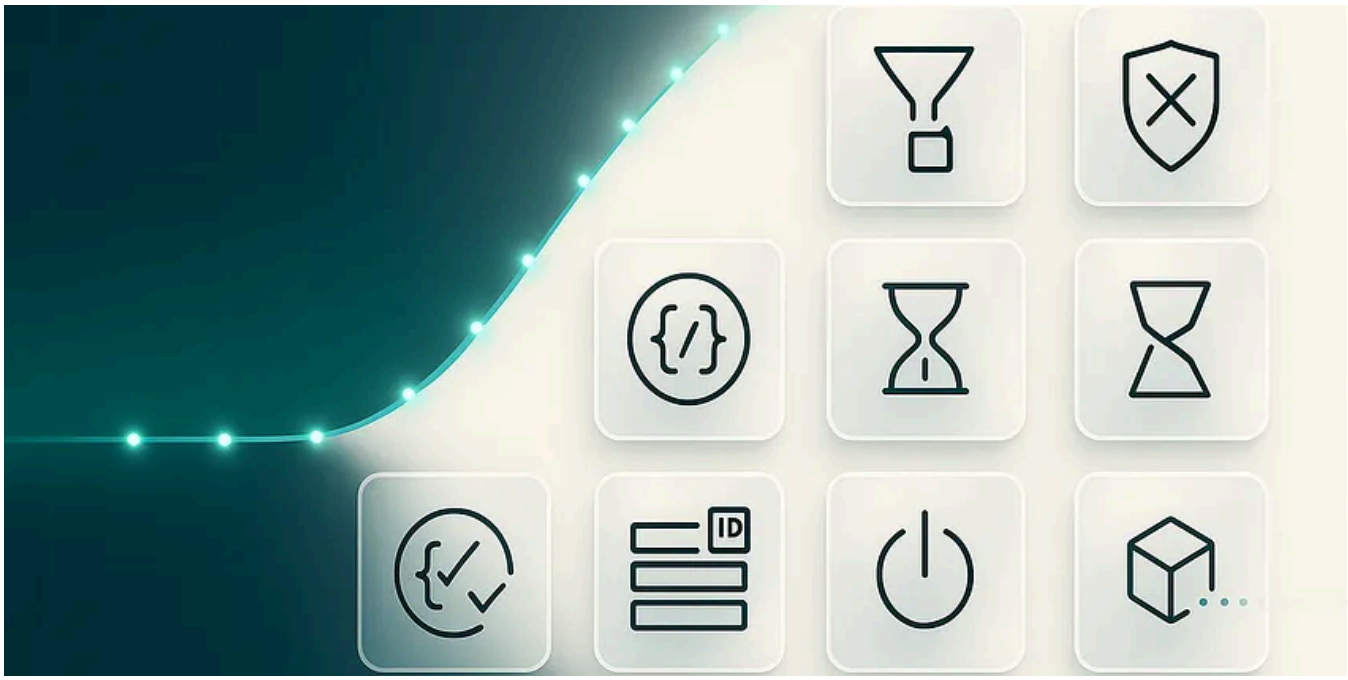


In Stackademic by bektiaw

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



Hash Block

10 Node.js Error-Handling Tricks That Saved Me

Practical patterns to catch bugs early, keep services alive, and turn chaos into readable logs.

4d ago 17

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

See more recommendations