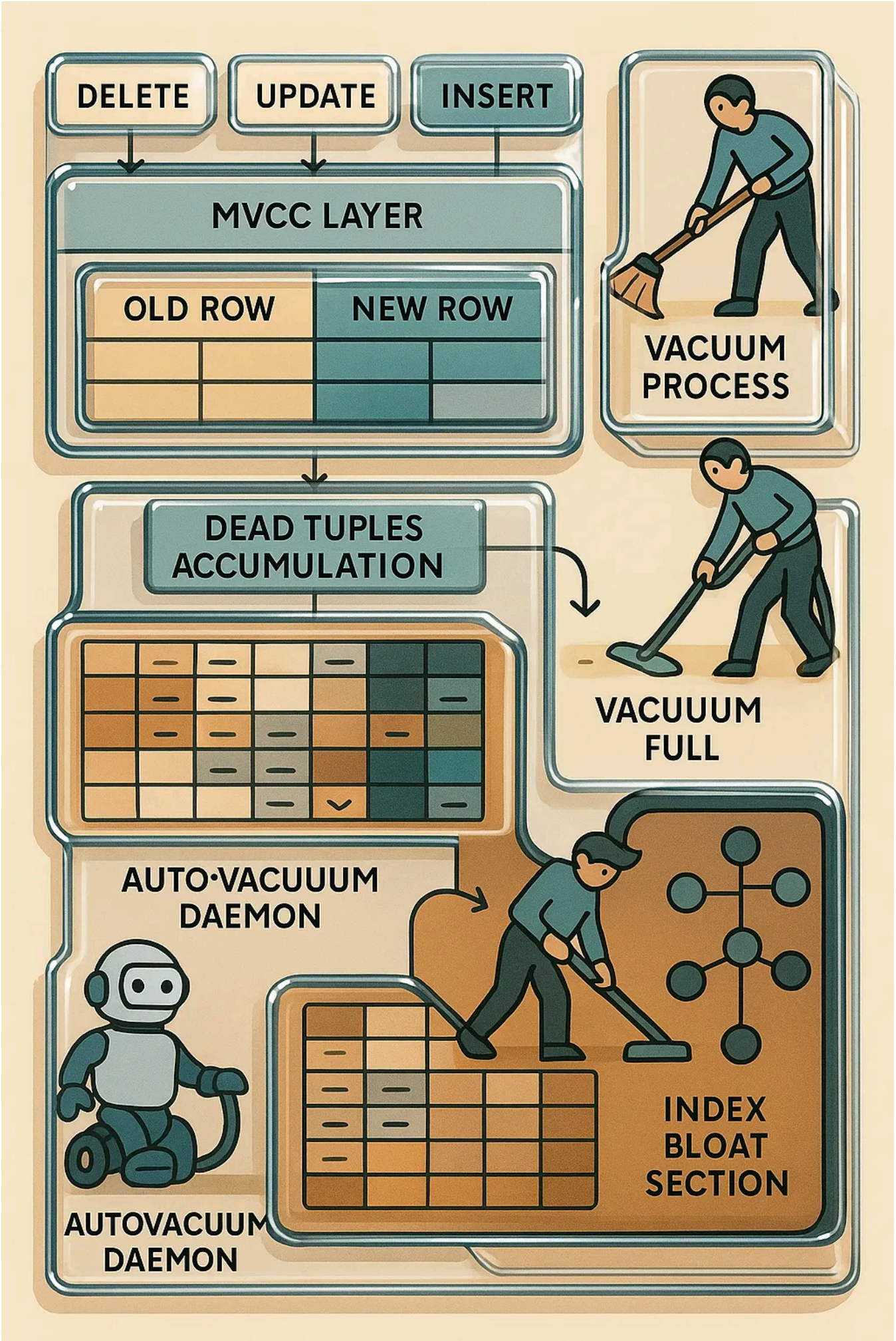
08-PostgreSQL 17: Complete Tuning Guide for VACUUM & AUTOVACUUM

13 min read · Sep 1, 2025

J Jeyaram Ayyalusamy Following


DELETE
UPDATE
INSERT
MVCC LAYER
OLD ROW
NEW ROW
VACUUM PROCESS
DEAD TUPLES ACCUMULATION
VACUUUM FULL
AUTO·VACUUUM DAEMON
INDEX BLOAT SECTION
AUTOVACUUM DAEMON

PostgreSQL's MVCC design creates **dead tuples** during `UPDATE` / `DELETE` . **VACUUM** reclaims them; **AUTOVACUUM** schedules that work. Get these knobs right and you minimize bloat, avoid wraparound, and keep queries fast.

> Tip: Almost every autovacuum knob has a **per-table override** via `ALTER TABLE … SET ( … )` . That's how you tune hot tables aggressively without punishing the whole cluster.

0) Hard Prerequisites for VACUUM in PostgreSQL 17

Before diving into VACUUM tuning itself, there are a few **critical prerequisites** that must be in place. Without these, even the best autovacuum settings won't deliver good performance. Think of them as the foundation of your PostgreSQL house: if the base isn't strong, no amount of decoration will help.

🔹 `track_counts = on` — Required for Autovacuum Decisions

PostgreSQL's autovacuum system depends on statistics about how many rows were inserted, updated, or deleted.
These statistics are tracked only when the parameter `track_counts` is set to **on**.

- If `track_counts = off` , PostgreSQL has **no visibility** into dead tuples.
- That means autovacuum will not know when to run, leading to table bloat, slower queries, and eventually transaction ID wraparound risks.

👉 **Example (postgresql.conf):**

```
track_counts = on
```

✅ **Best Practice:** Always keep `track_counts` enabled. Disabling it is almost never safe in a production system.

📌 **Analogy:** It's like keeping your car's fuel gauge working. Without it, you'll never know when you're about to run out of fuel.

🔹 `shared_buffers` Sized Sanely — Vacuum Relies on Buffer Cache

`shared_buffers` is one of the most important PostgreSQL memory settings. It defines how much RAM PostgreSQL can use for caching table and index data.

- A **too-small value** means vacuum has to constantly fetch data from disk, which is slow.
- A **too-large value** can cause memory pressure and starve the operating system's own disk cache.

👉 **General guideline (PostgreSQL 17):**

- For dedicated database servers: set `shared_buffers` to **25%–40% of total RAM.**
- For mixed workloads (DB + other apps on same server): use a lower value to avoid starving other processes.

👉 **Example (postgresql.conf):**

```
shared_buffers = 8GB    # on a server with 32GB RAM
```

✅ **Best Practice:** Vacuum benefits directly from a well-sized buffer cache because it avoids repeated I/O operations.

📌 **Analogy:** Think of `shared_buffers` as the size of your desk. If your desk is too small, you keep running back to the filing cabinet (disk). If your desk is reasonably large, you can spread out and work faster.

🔹 Enough I/O — Vacuum Is I/O Bound

VACUUM is not primarily a CPU-intensive process — it's mostly **I/O bound**. It reads tables, scans pages, and updates indexes. If your storage system is slow, vacuum performance will suffer.

- On **fast SSDs or NVMe drives**, vacuum can keep up even with high-churn workloads.
- On **slower spinning disks**, vacuum may lag, creating backlog and table bloat.

PostgreSQL provides **cost-based throttling parameters** (`vacuum_cost_delay` , `vacuum_cost_limit`) to balance vacuum work with user queries. These settings help control how aggressively vacuum consumes I/O resources.

👉 **Example:**

```
vacuum_cost_limit = 2000
vacuum_cost_delay = 5ms
```

This means: vacuum will pause briefly after doing a chunk of I/O work, reducing the chance it competes heavily with user queries.

✅ **Best Practice:** Ensure your hardware (or cloud storage) can handle sustained I/O loads, and then tune cost-based throttling according to your workload.

📌 **Analogy:** Imagine cleaning a very large warehouse. If your broom (I/O system) is flimsy, you'll clean very slowly. A stronger broom (faster disks) makes the cleanup much more efficient.

✨ Summary

Before tuning autovacuum and vacuum in PostgreSQL 17, make sure the basics are covered:

1. **`track_counts = on`** → Without statistics, autovacuum is blind.

2. **Proper `shared_buffers` sizing** → A balanced buffer cache speeds up vacuum dramatically.
3. **Sufficient I/O capacity** → Vacuum is I/O bound, so invest in fast storage and tune cost-based throttling.

With these prerequisites in place, PostgreSQL's vacuuming can run efficiently, keeping your database healthy and preventing bloat.

1) Core autovacuum controls

Parameter	Default	Scope	What it does	When to change
`autovacuum`	`on`	global	Enables the launcher & workers (wraparound vacuum still forced even if `off`).	Leave `on`.
`autovacuum_max_workers`	`3`	global	Max concurrent autovacuum workers.	Raise to 5–10 if CPU/I/O allow and you have many active tables.
`autovacuum_naptime`	`1min`	global	Minimum time between database scans by the launcher.	Lower (e.g., `30s`) for high-churn DBs.

Example

- `autovacuum = on`

This turns on PostgreSQL's **autovacuum launcher**, which is the background process that automatically schedules vacuuming and analyzing of tables.

- If this is set to `off`, PostgreSQL will not run routine autovacuum jobs.
- However, PostgreSQL will still force emergency vacuums for transaction ID wraparound protection.
  ✅ **Best practice:** Always keep this enabled in production, otherwise tables will bloat and performance will degrade.

🔹 `autovacuum_max_workers = 5`

This defines the **maximum number of autovacuum worker processes** that can run at the same time.

- Default is `3`, which can be too low if you have many busy tables.
- By raising it to `5`, PostgreSQL can clean more tables in parallel.
- Increasing this helps large databases but also means more CPU and I/O usage.
  ✅ **Tip:** Tune this based on server capacity — for example, 5–10 workers on a server with good CPU and fast storage.

🔹 `autovacuum_naptime = 30s`

This sets how often the **autovacuum launcher wakes up** to check all databases for work.

- Default is `1 minute` (60s).
- By lowering it to `30s`, PostgreSQL checks more frequently, which helps in high-churn databases where rows are constantly updated or deleted.
- The trade-off is slightly more background activity, but tables stay cleaner.
  ✅ **Good for OLTP systems** where tables accumulate dead rows quickly.

This setup makes PostgreSQL more proactive in cleaning up dead tuples, especially in workloads with frequent inserts, updates, and deletes.

```
autovacuum = on
autovacuum_max_workers = 5
autovacuum_naptime = 30s
```

2) When VACUUM/ANALYZE triggers (threshold + scale factor)

PostgreSQL decides **when to run autovacuum** on a table by comparing the number of **dead tuples** (rows that have been deleted or updated and are no longer visible) against a threshold.

That threshold is calculated with this formula:

```
dead_tuples >= autovacuum_vacuum_threshold
             + autovacuum_vacuum_scale_factor × reltuples
```

Let's explain each piece:

- `dead_tuples`

This is the number of row versions in a table that are no longer needed. They are invisible to queries but still take up space.
👉 Autovacuum is triggered once this count passes the threshold.

- `autovacuum_vacuum_threshold`

This is a **fixed minimum number of dead tuples** that must exist before autovacuum considers a table for cleanup.

- Default: `50`
- Acts as a "base cost" — even very small tables won't be vacuumed unless they have at least this many dead rows.

🔹 `autovacuum_vacuum_scale_factor`

This is a **fraction of the table size** (`reltuples` = number of rows) added to the threshold.

- Default: `0.2` (20%)
- Larger tables need more dead tuples before autovacuum kicks in.

🔹 `reltuples`

This is PostgreSQL's estimate of the total number of rows in the table, stored in system catalogs.

🔧 Example Calculation

Suppose you have a table `orders` with **1,000,000 rows** (`reltuples = 1,000,000`).

Default settings:

```
autovacuum_vacuum_threshold = 50
autovacuum_vacuum_scale_factor = 0.2
```

Threshold =

```
50 + (0.2 × 1,000,000)
= 50 + 200,000
= 200,050
```

👉 Autovacuum will run once **dead_tuples ≥ 200,050.**

That means **20% of the table** must be dead before vacuum kicks in.

⚡ Why Tuning Matters

For large OLTP tables with constant updates/deletes, waiting until 20% of rows are dead is too late — the table may bloat badly.

So you can **lower the scale factor:**

```
autovacuum_vacuum_scale_factor = 0.02   # 2%
```

Now the threshold is:

```
50 + (0.02 × 1,000,000)
= 50 + 20,000
= 20,050
```

👉 Autovacuum will run much earlier (after ~20k dead rows instead of 200k).

This keeps the table lean and improves query performance.

Tuning these values is critical for **big, frequently updated tables** to prevent bloat and keep performance stable.

Parameter	Default	Scope	Notes
`autovacuum_vacuum_threshold`	`50`	global & per-table	Base trigger for VACUUM.
`autovacuum_vacuum_scale_factor`	`0.2`	global & per-table	Fraction of table size. Lower for hot OLTP tables (e.g., `0.05` or `0.01`).
`autovacuum_vacuum_insert_threshold`	`1000`	global & per-table	Insert-only trigger (helps visibility map/IOS).
`autovacuum_vacuum_insert_scale_factor`	`0.2`	global & per-table	Scale for the insert trigger.
`autovacuum_analyze_threshold`	`50`	global & per-table	Base trigger for ANALYZE.
`autovacuum_analyze_scale_factor`	`0.1`	global & per-table	Planner stats freshness. Lower for skew-prone/rapidly changing data.

Example (global)

```
autovacuum_vacuum_threshold = 50
autovacuum_vacuum_scale_factor = 0.10
autovacuum_vacuum_insert_threshold = 1000
autovacuum_vacuum_insert_scale_factor = 0.05
autovacuum_analyze_threshold = 50
autovacuum_analyze_scale_factor = 0.05
```

Example (per table)

The command changes the **autovacuum settings only for the `orders` table** (it overrides the global defaults).

- **`autovacuum_vacuum_threshold = 100`** → Vacuum will trigger after at least 100 dead rows are found.
- **`autovacuum_vacuum_scale_factor = 0.02`** → In addition, vacuum will trigger when dead rows exceed 2% of the table size.
- **`autovacuum_analyze_scale_factor = 0.03`** → Analyze will refresh statistics when about 3% of the table changes, keeping query plans accurate.

- **`autovacuum_vacuum_insert_threshold = 500`** → Vacuum will also trigger after 500 new rows are inserted (helps visibility for index-only scans).
- **`autovacuum_vacuum_insert_scale_factor = 0.03`** → Vacuum can also be triggered if inserts exceed 3% of the table size.

```
ALTER TABLE orders
  SET (autovacuum_vacuum_threshold = 100,
       autovacuum_vacuum_scale_factor = 0.02,
       autovacuum_analyze_scale_factor = 0.03,
       autovacuum_vacuum_insert_threshold = 500,
       autovacuum_vacuum_insert_scale_factor = 0.03);
```

3) Cost-based I/O throttling (pacing)

Parameter	Default	Scope	What it means
`vacuum_cost_page_hit`	`1`	global	Cost for cached page.
`vacuum_cost_page_miss`	`10`	global	Cost for read miss.
`vacuum_cost_page_dirty`	`20`	global	Cost when dirtying page.
`vacuum_cost_limit`	`200`	global	Budget per cycle before pausing.
`vacuum_cost_delay`	`0ms`	global	Sleep after hitting the budget.
`autovacuum_vacuum_cost_limit`	`-1`	global & per-table	Per-worker override (`-1` uses global).
`autovacuum_vacuum_cost_delay`	`-1`	global & per-table	Per-worker override.

Throughput-friendly pacing

PostgreSQL's `VACUUM` is **I/O bound**. If it runs too aggressively, it can slow down user queries. To balance this, PostgreSQL uses a **cost-based throttling system**:

- Each page access (hit, miss, dirty) has a "cost."

- When the cost reaches `vacuum_cost_limit`, vacuum pauses for `vacuum_cost_delay` before continuing.
- This spreads out the work and avoids overwhelming disks.

Your configuration looks like this:

```
vacuum_cost_limit = 2000
vacuum_cost_delay = 5ms
autovacuum_vacuum_cost_limit = 4000
autovacuum_vacuum_cost_delay = 2ms
```

🔹 vacuum_cost_page_hit = 1

This is the cost charged when VACUUM touches a page already in shared buffers (cache hit).
Default is **1**. Raising it makes cached work "spend" the budget faster, leading to more frequent pauses.
Best for smoothing I/O on very latency-sensitive OLTP systems where even cached scans should throttle.

👉 **Example:** With `vacuum_cost_limit = 200`, VACUUM can touch about **200 ÷ 1 = 200** cached pages before pausing.

🔹 vacuum_cost_page_miss = 10

Cost when VACUUM must read a page from disk (cache miss).
Default is **10**. Increasing it penalizes reads more, forcing more frequent sleeps when the workload causes many misses.
Best for HDD/slow storage or shared environments where reads spike latency.

👉 **Example:** With `vacuum_cost_limit = 200`, about **200 ÷ 10 = 20** disk reads happen before a pause.

🔹 vacuum_cost_page_dirty = 20

Cost when VACUUM dirties a previously clean page (implies a write-back later).
Default is **20**. Raising it heavily discourages bursts of dirtying and spreads write

pressure over time.
Best when checkpoints or background writes are causing jitter and you want gentler write patterns.

👉 **Example:** With `vacuum_cost_limit = 200`, VACUUM can dirty **200 ÷ 20 = 10** pages before pausing. With limit **2000**, that's **100** pages.

🔹 `vacuum_cost_limit = 2000`

This means a **manual VACUUM command** can perform work that accumulates a cost of **2000 units** before it must pause.

- Default is much lower (`200`).
- Raising it allows VACUUM to clean more pages in one cycle, finishing faster.
- Best for when you can afford heavier maintenance windows (e.g., off-peak hours).

👉 Example: If scanning dirty pages (cost=20 each), VACUUM can process about `2000 ÷ 20 = 100 pages` before pausing.

🔹 `vacuum_cost_delay = 5ms`

When the limit is hit (2000 units), VACUUM will pause for **5 milliseconds** before continuing.

- This short break reduces I/O pressure on the system.
- Default is `0ms` (no pause).
- Setting a non-zero delay is safer for production, so user queries are less impacted.

🔹 `autovacuum_vacuum_cost_limit = 4000`

This applies to **autovacuum workers** (the background vacuum processes).

- They get a **higher cost limit** (4000 vs. 2000), which means they can do **twice as much work** before pausing.
- This makes autovacuum run more efficiently, since background jobs are often expected to handle large workloads continuously.

🔹 `autovacuum_vacuum_cost_delay = 2ms`

This sets the pause duration for autovacuum workers.

- At 2ms, it's shorter than the 5ms used for manual VACUUM.
- That means autovacuum will pause less and move faster through large tables.
- This helps prevent bloat from growing too fast, especially on high-traffic tables.

✅ Putting It All Together

With these settings:

- **Manual VACUUM** runs at a steady pace (2000 cost units, 5ms pause) → safer for on-demand use when you don't want to overload the system.
- **Autovacuum** runs more aggressively (4000 cost units, 2ms pause) → ensures background cleanup keeps up with heavy workloads and prevents table bloat.

📌 **Analogy:** Imagine vacuuming your house.

- Manual vacuuming (you doing it) → you take breaks more often (every 2000 units, 5ms rest).
- Autovacuum (robot vacuum) → it works harder and rests less (every 4000 units, 2ms rest), because you expect it to keep cleaning without you watching.

```
vacuum_cost_limit = 2000
vacuum_cost_delay = 5ms
autovacuum_vacuum_cost_limit = 4000
autovacuum_vacuum_cost_delay = 2ms
```

> Rule of thumb: If autovacuum hurts foreground latency, increase delays or reduce limits; if tables bloat, decrease delays / increase limits and/or increase workers.

4) Memory & buffer usage

Parameter	Default	Scope	Notes
`autovacuum_work_mem`	`-1`	global	Memory per autovacuum worker; `-1` = use `maintenance_work_mem`. Increase for large indexes (e.g., `512MB–1GB`).
`maintenance_work_mem`	`64MB`	global	Used by manual `VACUUM`, `CREATE INDEX`, etc.
`vacuum_buffer_usage_limit`	`2MB`	global & per-VACUUM option	Caps how much of `shared_buffers` VACUUM can occupy; reduce interference with queries.

Example

```
autovacuum_work_mem = 1GB
vacuum_buffer_usage_limit = 8MB
```

5) Freezing & wraparound protection (critical)

These protect you from transaction ID wraparound (catastrophic if ignored).

Parameter	Typical Default	Scope	What it does
`autovacuum_freeze_max_age`	~2,000,000,000	global	Forces anti-wraparound VACUUM when reached.
`autovacuum_multixact_freeze_max_age`	hundreds of millions	global	Same for multixact IDs (shared locks).
`vacuum_freeze_min_age`	50,000,000	global & per-table	Don't freeze very new tuples.
`vacuum_freeze_table_age`	150,000,000	global & per-table	Aggressive freeze scan threshold.
`vacuum_failsafe_age` / `vacuum_multixact_failsafe_age`	implementation-dependent	global	Emergency mode to avoid wraparound (prioritizes freezing over niceness).

Aggressive one-off

```
VACUUM (FREEZE, VERBOSE) my_large_table;
```

Monitor risk

```
SELECT
  relname,
  age(relfrozenxid) AS xid_age
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE relkind = 'r'
ORDER BY xid_age DESC
LIMIT 20;
```

6) Table truncation & index cleanup

Setting / Option	Default	Where	What it does
`vacuum_truncate`	`on`	global & per-table	Returns empty tail pages to OS at end of VACUUM (brief `ACCESS EXCLUSIVE` lock).
`VACUUM … INDEX_CLEANUP`	`AUTO`	command option	`AUTO` lets PG skip index cleanup if it's not worthwhile; `ON` forces it; `OFF` skips it.
`VACUUM … PROCESS_TOAST`	`AUTO`	command option	Control whether to process TOAST table explicitly.
`VACUUM … SKIP_LOCKED`	off	command option	Skip relations that cannot be locked (keep maintenance moving).
`VACUUM … PARALLEL n`	off	command option	Parallel index cleanup on large tables (good on multi-core systems).
`VACUUM … BUFFER_USAGE_LIMIT 'N MB'`	—	command option	Per-run override of `vacuum_buffer_usage_limit`.
`VACUUM … DISABLE_PAGE_SKIPPING`	off	command option	Scan all pages even if visibility map says they're all-visible (useful for correctness checks).

Example 1:

```
-- Force thorough index cleanup and shrink buffers used
VACUUM (INDEX_CLEANUP ON, BUFFER_USAGE_LIMIT '16MB') orders;
```

🔹 What it means:

1. **`INDEX_CLEANUP ON`**

- Normally, PostgreSQL may skip cleaning indexes during vacuum if it thinks the benefit is small (default = `AUTO`).
- With `ON`, you force VACUUM to remove **all dead index entries.**
- This helps avoid **index bloat**, especially on large fact tables (`orders`) that get many updates or deletes.

2. **`BUFFER_USAGE_LIMIT '16MB'`**

- This limits how much of `shared_buffers` VACUUM is allowed to use during the operation.

- Default is very small (2MB).
- By setting it to `16MB`, you let VACUUM use more memory to process pages, but not so much that it starves user queries.
- It keeps the maintenance predictable and avoids cache pollution.

👉 This command runs a vacuum on the `big_fact` table, forcing a **full index cleanup** and allowing VACUUM to use **up to 16MB of buffer cache** for its work. It's useful when you know indexes are bloated and want to reclaim space more aggressively.

Example 2:

```
-- Keep maintenance moving under lock contention
VACUUM (SKIP_LOCKED, PARALLEL 4) orders;
```

🔹 What it means:

1. **`SKIP_LOCKED`**

- If VACUUM can't immediately lock a page (because another transaction is holding it), it will **skip that page** and continue with the rest.
- Without this, VACUUM might wait (or block), slowing down the process.
- This is useful in **highly concurrent systems** where you don't want vacuum jobs to stall.

2. **`PARALLEL 4`**

- Allows VACUUM to use **4 parallel workers** for index cleanup.
- Great for very large partitioned or heavily indexed tables like `orders`.
- It speeds up vacuuming on multi-core servers by splitting the work.

👉 This command vacuums the `orders` table but tells PostgreSQL **not to wait on locked pages** (`SKIP_LOCKED`) and to use **4 parallel workers** for faster cleanup. It keeps the system responsive even under heavy load.

📌 **Analogy:**

- The first is like deep-cleaning your house but only using a fixed number of cleaning supplies (16MB buffer) to stay organized.
- The second is like sending four cleaners into different rooms at once and telling them to **skip locked rooms** and come back later, so the cleaning doesn't stall.

7) Logging & live monitoring

Parameter / View	What you get
`log_autovacuum_min_duration = 0`	Logs every autovacuum — essential while tuning. Use `-1` to disable after.
`VACUUM (VERBOSE)`	Per-run detail in logs (dead/live tuples, index cleanup).
`pg_stat_progress_vacuum`	Live progress: phase, heap_blks_total/scanned/vacuumed, index info, etc.
`pg_stat_user_tables`	`n_live_tup` , `n_dead_tup` , last vacuum/analyze timestamps.
`pg_stat_all_tables`	Same across all schemas; useful for wraparound monitoring with `age(relfrozenxid)` .

Quick bloat watch

```
SELECT relname, n_dead_tup, n_live_tup,
       round(100.0*n_dead_tup/GREATEST(n_live_tup,1),2) AS dead_pct
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC
LIMIT 20;
```

8) Per-table storage parameters (fine-grained control)

Most autovacuum controls can be applied per table (and per TOAST table using the `toast.` prefix).

- `autovacuum_enabled` (bool)
- `autovacuum_vacuum_threshold` (int)
- `autovacuum_vacuum_scale_factor` (float)
- `autovacuum_vacuum_insert_threshold` (int)
- `autovacuum_vacuum_insert_scale_factor` (float)
- `autovacuum_analyze_threshold` (int)
- `autovacuum_analyze_scale_factor` (float)
- `autovacuum_vacuum_cost_limit` (int)
- `autovacuum_vacuum_cost_delay` (time)

Examples

```
-- Aggressive on a hot OLTP table
ALTER TABLE payments
  SET (autovacuum_vacuum_scale_factor = 0.01,
       autovacuum_analyze_scale_factor = 0.02,
       autovacuum_vacuum_cost_limit = 8000,
       autovacuum_vacuum_cost_delay = 1ms);
```

```
-- Insert-only log table (favor visibility)
ALTER TABLE http_logs
  SET (autovacuum_vacuum_insert_threshold = 500,
       autovacuum_vacuum_insert_scale_factor = 0.02,
       autovacuum_analyze_scale_factor = 0.02);
```

```
-- Temporarily disable (bulk backfill); wraparound vacuums still occur
ALTER TABLE staging_load SET (autovacuum_enabled = off);
```

9) Practical presets (drop-in profiles)

A) OLTP (many small updates/deletes)

```
autovacuum = on
autovacuum_max_workers = 6
autovacuum_naptime = 30s
```

```
autovacuum_vacuum_threshold = 50
autovacuum_vacuum_scale_factor = 0.02
autovacuum_analyze_scale_factor = 0.03
autovacuum_vacuum_insert_threshold = 500
autovacuum_vacuum_insert_scale_factor = 0.03
vacuum_cost_limit = 3000
vacuum_cost_delay = 3ms
autovacuum_vacuum_cost_limit = 6000
autovacuum_vacuum_cost_delay = 1ms
autovacuum_work_mem = 1GB
vacuum_buffer_usage_limit = 8MB
log_autovacuum_min_duration = 0
```

B) Insert-heavy (append-only logs, time-series)

```
autovacuum_vacuum_insert_threshold = 500
autovacuum_vacuum_insert_scale_factor = 0.02
autovacuum_vacuum_scale_factor = 0.20     # normal dead-tuple trigger can stay
autovacuum_analyze_scale_factor = 0.05
```

C) Mixed workload, cautious pacing

```
autovacuum_max_workers = 5
autovacuum_naptime = 45s
vacuum_cost_limit = 2000
vacuum_cost_delay = 5ms
autovacuum_vacuum_scale_factor = 0.05
autovacuum_analyze_scale_factor = 0.05
log_autovacuum_min_duration = 200ms
```

10) VACUUM command options (quick reference)

```
VACUUM                          -- plain vacuum (no lock, space reusable)
VACUUM (VERBOSE)                -- detailed log
VACUUM (FULL)                   -- rewrites table, exclusive lock, shrinks file
VACUUM (FREEZE)                 -- aggressive freezing (xid safety)
VACUUM (INDEX_CLEANUP ON|OFF|AUTO)
VACUUM (DISABLE_PAGE_SKIPPING)  -- scan all pages regardless of VM
VACUUM (PROCESS_TOAST ON|OFF)
VACUUM (SKIP_LOCKED)
VACUUM (PARALLEL 4)
VACUUM (BUFFER_USAGE_LIMIT '16MB')
```

> **`VACUUM FULL`** : use sparingly; it takes an **exclusive lock** and is **I/O heavy**. Prefer normal VACUUM + `REINDEX` for index bloat; consider `CLUSTER` / `pg_repack` for online compaction if needed.

11) Tuning workflow (step-by-step)

1. Measure

- Identify hot/bloated tables: `pg_stat_user_tables` (`n_dead_tup` , `last_autovacuum`).
- Watch live runs: `pg_stat_progress_vacuum` .
- Turn on logs: `log_autovacuum_min_duration = 0` (temporarily).

2. **Adjust globally** (small moves)

- Lower `autovacuum_vacuum_scale_factor` (e.g., 0.2 → 0.1).
- Raise worker/memory/cost limits if hardware allows.

3. **Target per-table outliers**

- Set aggressive per-table storage params for the top offenders.
- Add insert-trigger knobs for append-only tables.

4. **Stabilize**

- Reduce `log_autovacuum_min_duration` logging once happy.
- Keep an eye on `age(relfrozenxid)` for wraparound safety.

12) Your snippet + minimal must-adds

You shared:

```
autovacuum = on
autovacuum_naptime = 60
autovacuum_vacuum_scale_factor = 0.2
autovacuum_analyze_scale_factor = 0.1
```

Add these **minimum essentials**:

```
autovacuum_vacuum_threshold = 50
autovacuum_vacuum_insert_threshold = 1000
autovacuum_vacuum_insert_scale_factor = 0.2
log_autovacuum_min_duration = 0      # enable during tuning, disable later
```

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 Oracle Open Source AWS MySQL

Following

Written by Jeyaram Ayyalusamy

158 followers · 2 following

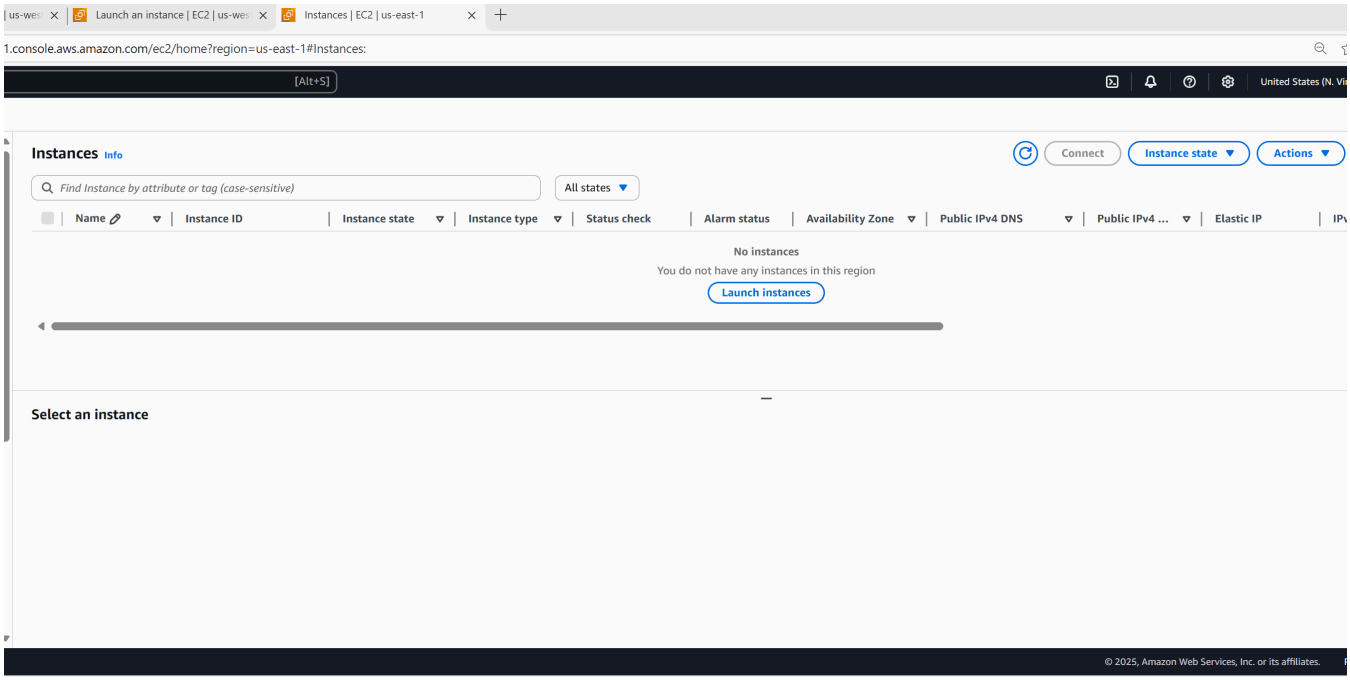
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

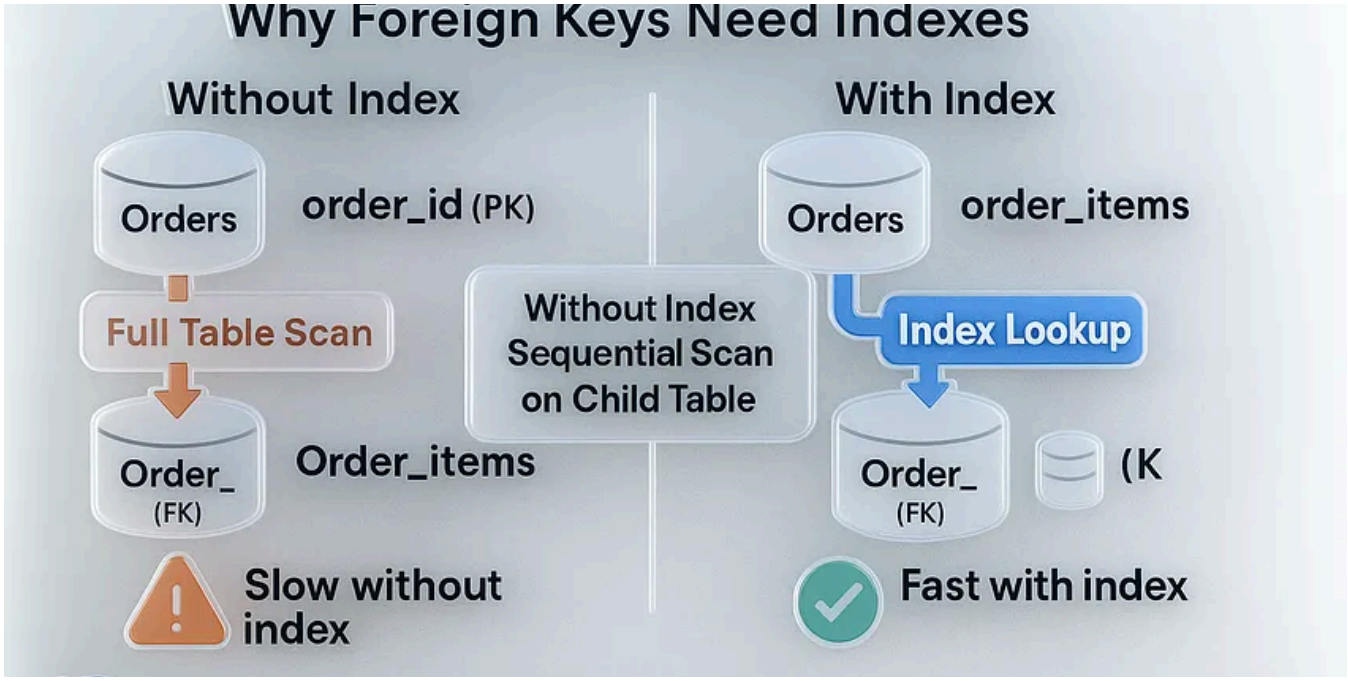


Jeyaram Ayyalusamy

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

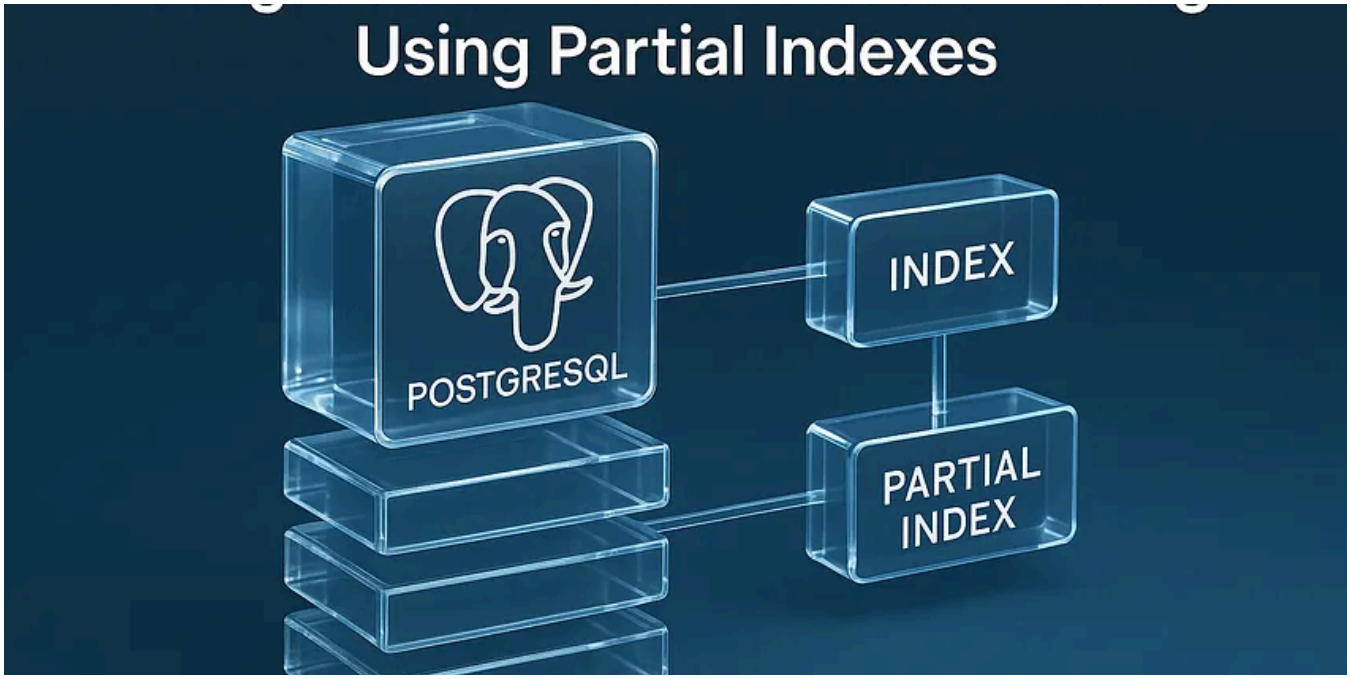


Jeyaram Ayyalusamy

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

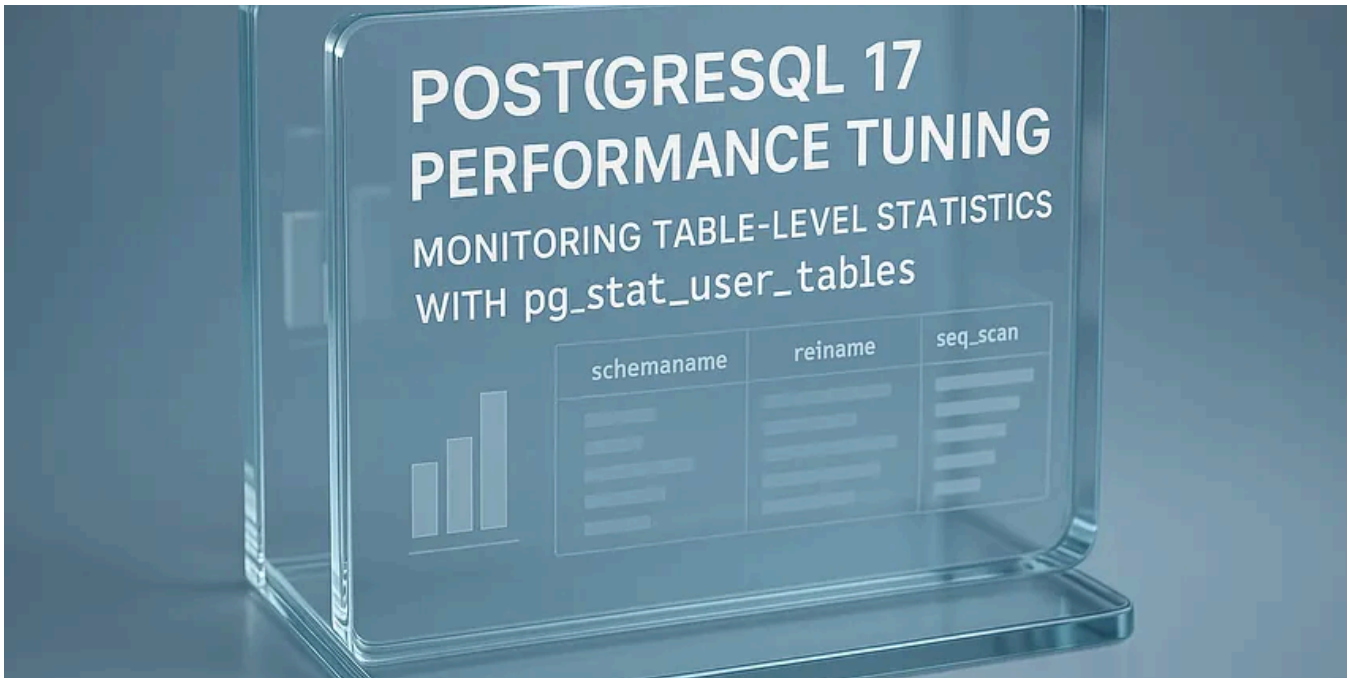


Jeyaram Ayyalusamy

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

24 - PostgreSQL 17 Performance Tuning: Monitoring Table-Level Statistics with pg_stat_user_tables

When tuning PostgreSQL, one of the most important steps is to observe table-level statistics. You cannot optimize what you cannot measure...

Sep 7 19 1

See all from Jeyaram Ayyalusamy

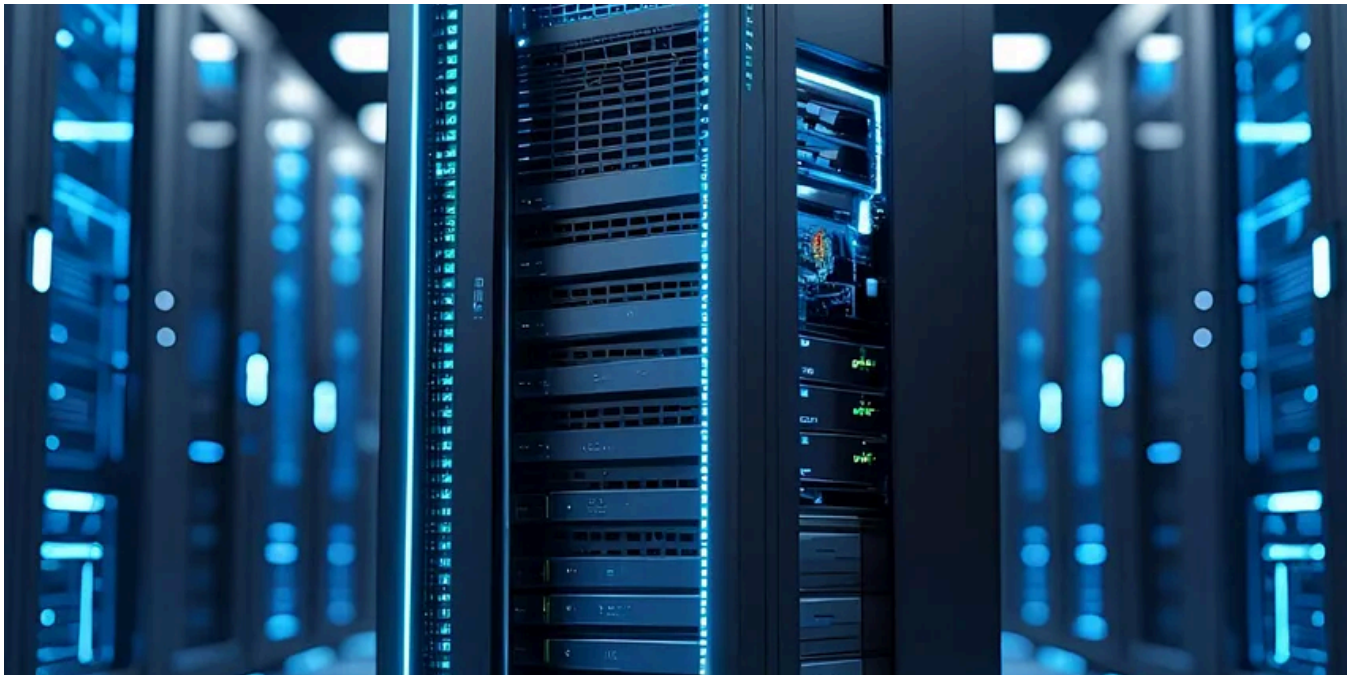
Recommended from Medium

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

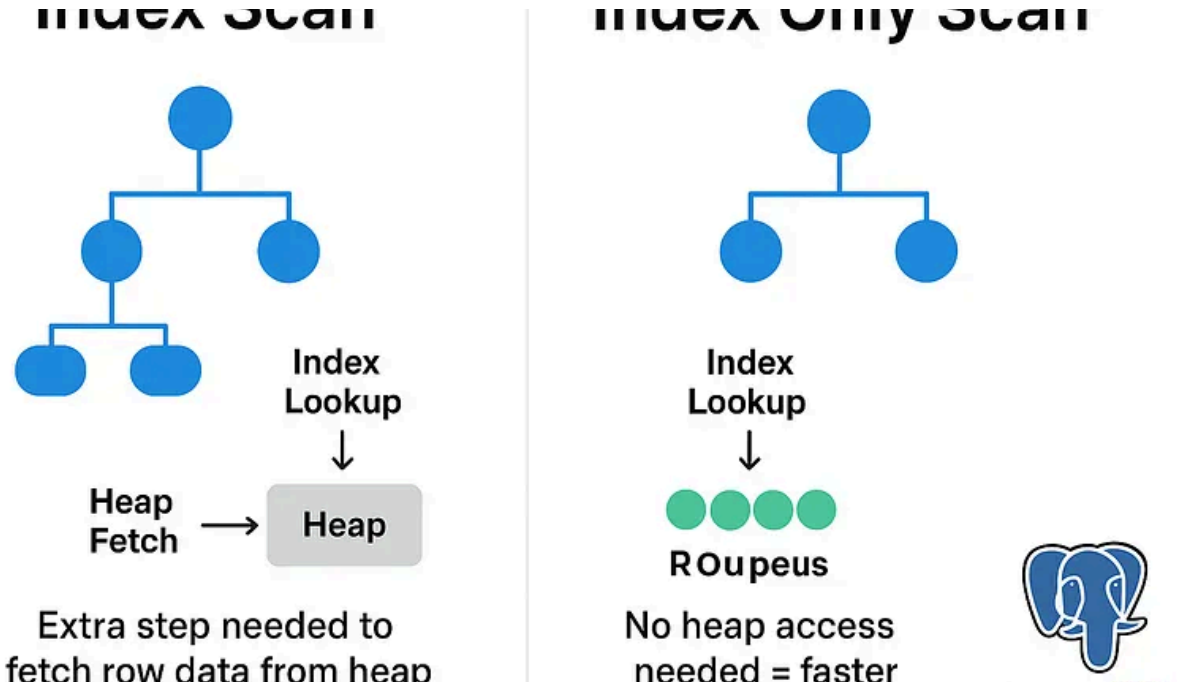


Rizqi Mulki

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

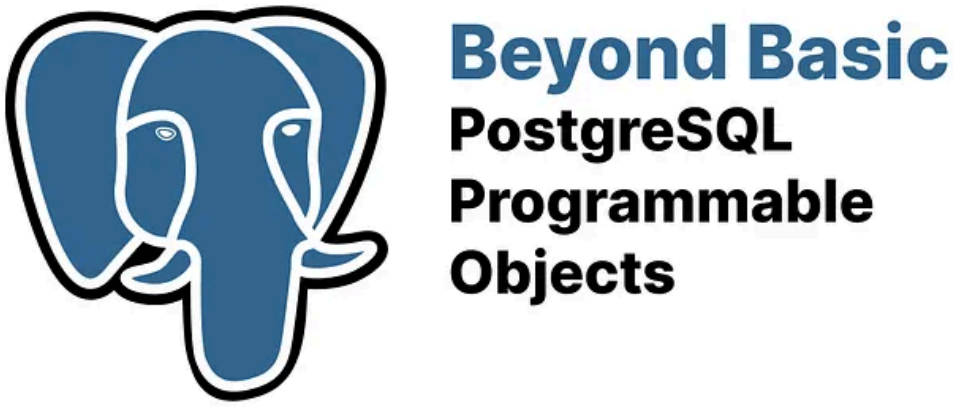


Eshemiedomi Samuel Oghenevwede

PostgreSQL Covering Indexes: Beyond Regular Indexes for Faster Queries

Go beyond regular indexes. Learn how PostgreSQL covering indexes cut heap access and deliver ultra-fast query performance.

Sep 6 2

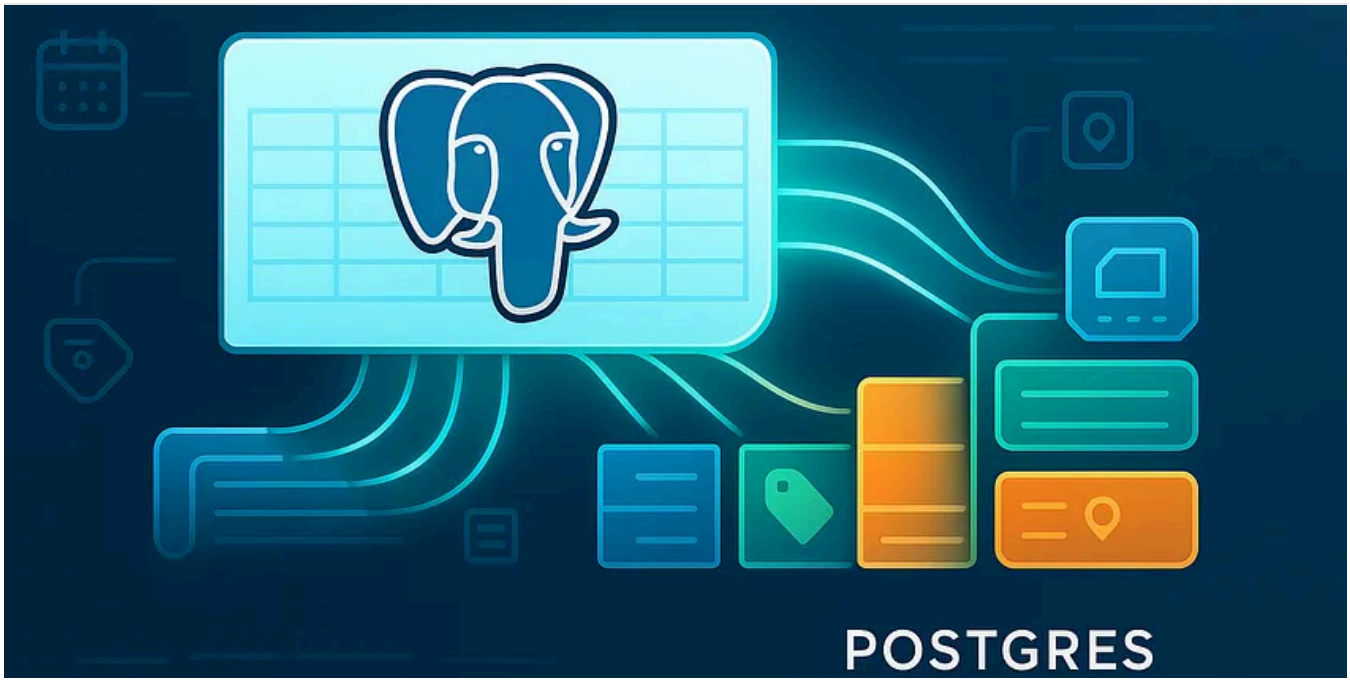


In Stackademic by bektiaw

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

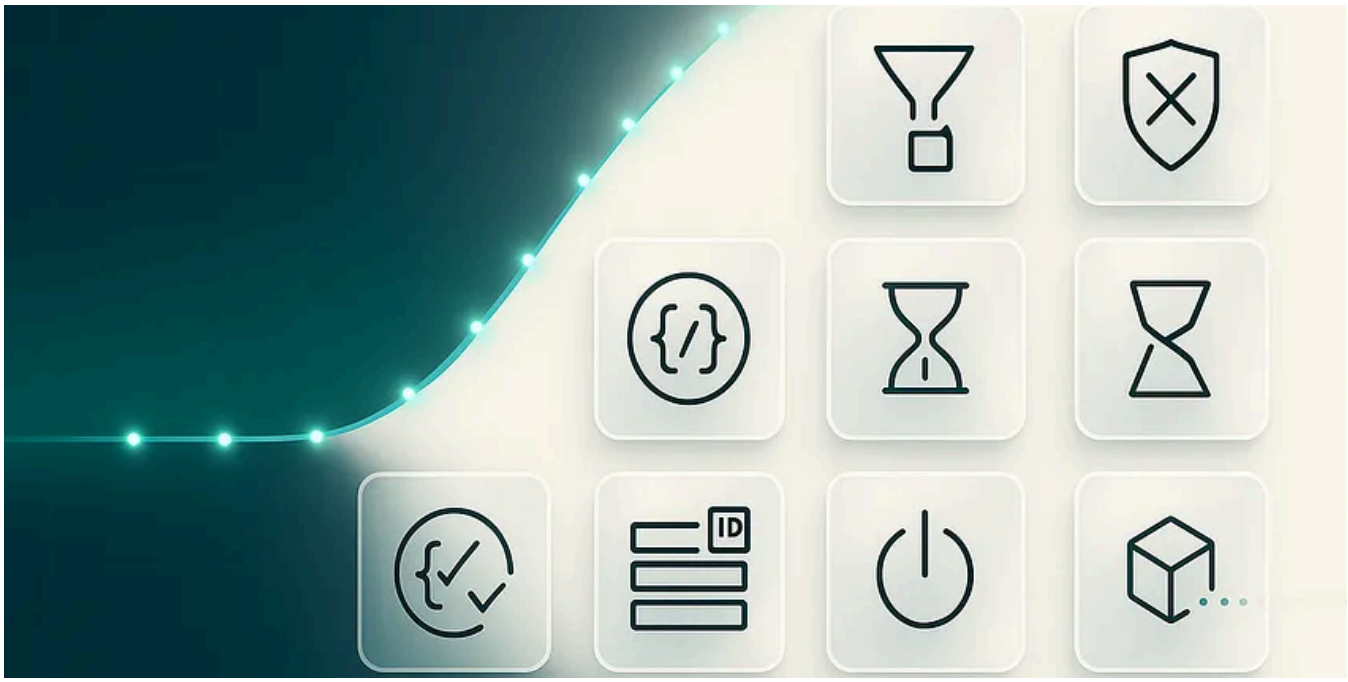


Thinking Loop

10 Postgres Partitioning Patterns for Internet-Scale Apps

Keep Postgres fast past a billion rows with real patterns and ready-to-run SQL.

Aug 13 88 2



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