Best Practices in PostgreSQL

8 rules for a database that never loses a commit. One data directory. One log. 8 KB pages.

1 WAL first, data file later

The log hits disk before the commit returns. The data page follows minutes later, at checkpoint.

Deep dive ↓ Level 0 · WAL first, commit second

ONE COMMIT, IN ORDER

`INSERT row 42` → `pg_wal/000000010000000000000003 · fsync` → `COMMIT returns ✓` → `checkpoint → base/16384/16397`

Cost of durability: **1 sequential fsync** of a few hundred bytes, not 4+ random 8 KB page writes. That trade is the entire design.

2 An index only helps if the planner picks it

No index, or stale statistics, and the same query reads 1,250,000 pages instead of 4.

Deep dive ↓ Reading EXPLAIN, line by line

EXPLAIN (ANALYZE, BUFFERS): THE TWO SHAPES

PLAN	PAGES READ	TIME AT 10M ROWS	VERDICT
Index Scan · Buffers: shared hit=4	4	0.05 ms	✓ healthy
Seq Scan · rows est. 1, actual 48,000	~1,250,000	3-8 s	✗ stale stats, run ANALYZE

3 Updates never overwrite: vacuum cleans up

Every UPDATE writes a new row version and leaves the old one behind. Until VACUUM removes it, every scan walks past the corpse.

Deep dive ↓ Level 3 · Vacuum, freeze, wraparound

ONE PAGE, BEFORE AND AFTER VACUUM

BEFORE: DEAD TUPLES PILE UP

`row v1 ✗ dead`

`row v2 ✗ dead`

`row v3 · live`

Table "grows" with zero new rows. This is bloat.

AFTER, VACUUM RECLAIMS THE SPACE

`row v3 · live`

`free space · reusable`

Check it: **n_dead_tup** in pg_stat_user_tables.

4 Checkpoint distance bounds crash recovery

On restart, PostgreSQL replays every WAL byte since the last checkpoint. That distance is a setting, not luck.

Deep dive ↓ Level 5 · max_wal_size is a checkpoint budget

MAX_WAL_SIZE BUDGETS THE REPLAY

`checkpoint ✓` → `WAL written since: this is what replays` → `crash · replay all of it`



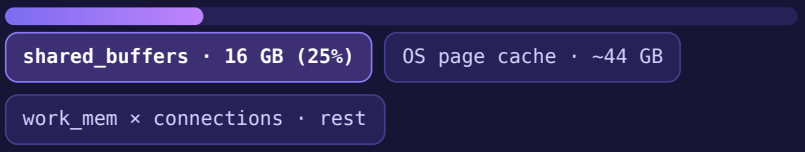
Default budget: **1 GB**. At 50 MB/s of WAL it fills in about 20 seconds, so a checkpoint fires about every 20 seconds and at most 1 GB ever replays. Replay time is that 1 GB divided by your replay speed. Measure the speed from the server log: the "redo starts at" line and the "redo done at" line carry timestamps, and their LSNs give you the bytes. Bytes divided by seconds is your replay speed. In pg_stat_checkpointer, num_requested outnumbering num_timed means the budget is too small for your write rate.

5 shared_buffers is ~25% of RAM: on purpose

The OS page cache holds the second copy. Setting shared_buffers to 80% double-buffers everything and evicts the working set.

Deep dive ↓ Level 1 · Shared buffers & checkpoints

64 GB OF RAM, SPLIT THE WAY PRODUCTION SPLITS IT



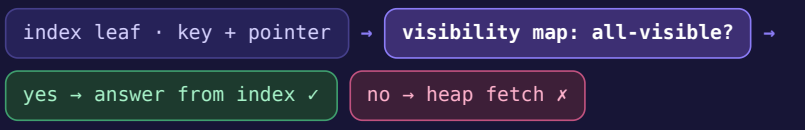
Proof it is working: **Buffers: shared hit** dominates **shared read** in EXPLAIN output.

6 "Index-only" is only true if the visibility map is fresh

The scan checks the visibility map before skipping the heap. A stale map turns it into an index scan plus one heap fetch per row.

Deep dive ↓ Level 2 · Index-only scans & the visibility map

THE VISIBILITY-MAP CHECK, PER ROW



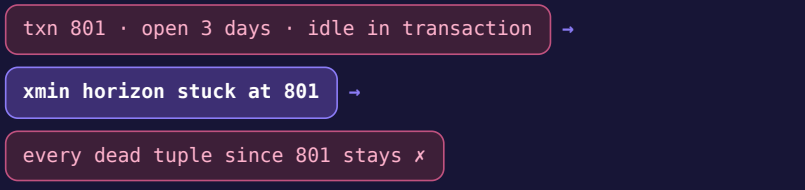

```
EXPLAIN (ANALYZE, BUFFERS) …  ← watch this line
  Heap Fetches: 0             ← healthy · in the hundreds = stale map,
```

7 One long transaction blocks vacuum for everyone

Vacuum cannot remove any version newer than the oldest open snapshot. One idle-in-transaction session from Friday freezes cleanup all weekend.

Deep dive ↓ Level 3 · The cleanup horizon

THE HORIZON IS SET BY THE OLDEST TRANSACTION




```
SELECT pid, now() - xact_start AS age, state
FROM pg_stat_activity ORDER BY age DESC;  ← the top row owns your
```

8 A synchronous standby makes commit wait: price it

synchronous_commit = on + synchronous_standby_names naming a sync standby means zero data loss on failover, paid for in commit latency on every single write.

Deep dive ↓ Level 4 · WAL streaming & sync standbys

WHAT THE COMMIT WAITS FOR

SETTING	COMMIT RETURNS AFTER	LOST ON PRIMARY CRASH
synchronous_commit = on, no sync standby	local WAL fsync	0 committed rows
+ synchronous_standby_names set	standby confirms WAL received	0, and the standby has it
synchronous_commit = off	nothing · ~3× WAL-writer delay	**last ~600 ms of commits ✗**