

Decoding postgresql.conf

Which parameters to tweak, and when.

Divya Sharma

Customer Reliability Engineer

Supabase is the **foundational platform** of the agentic AI era

Supabase is a fully integrated backend for modern applications. All best of breed. Use one or all.



Postgres Database



Authentication



Storage



Edge Functions



Realtime

Trusted by the leading AI Builders. All offer a seamless, full-stack experience built on Supabase.



Lovable



Figma

40+

AI Builder partners using Supabase as their foundational platform

3M+

Projects created on Supabase by AI Builders to date

100K+

Projects created per week by AI Builders

The journey of a transaction

Seven gates where we'll meet the parameters



The defaults are good. Until they aren't.

Postgres ships with ~350 GUCs. Do you really need to tune all?

01

Don't tune in the dark

Most parameters have a log signature, system view info or metrics for monitoring. Investigate those first.

02

Parameters interact

$\text{max_parallel_workers} \leq \text{max_worker_processes} \times \text{work_mem} \times \text{connections} \times \text{hash operations}.$

Slots vs WAL retention.

03

Wait for evidence

If the log isn't complaining, performance is good/acceptable, the views/metrics looks healthy — leave it alone.

1

STAGE 1 · THE DOORMEN

Connection Management

In this section

```
max_connections  
idle_session_timeout  
transaction_timeout
```

```
idle_in_transaction_session_timeout  
statement_timeout
```

max_connections is not a target. It's a ceiling.

- Each backend $\approx$ a process. Each process $\approx$ memory + file descriptors.
- Raising max_connections does not give you more throughput. It gives you more processes competing for the same CPU and lock manager.
- **Real answer is almost always a pooler (Supavisor, PgBouncer etc).**

WATCH IN LOGS

```
FATAL: sorry, too many clients already
FATAL: remaining connection slots are
reserved for non-replication...
```

MONITOR VIA

```
pg_stat_activity
```

```
pg_stat_database (dataname, numbackends)
```

The four timeouts: what they kill, and when

`statement_timeout`

Caps a single statement.

default: 0 (off)

`transaction_timeout` (PG 17+)

Caps an entire transaction — including idle time between statements.

default: 0 (off)

`idle_in_transaction_session_timeout`

Kills a transaction which is open while doing nothing. Antidote to bloat from BEGIN-then-forget.

default: 0 (off)

`idle_session_timeout`

Kills a connection with no open transaction. Pair with pooler timeouts

default: 0 (off)

WATCH IN LOGS

FATAL: terminating connection
due to idle-in-transaction
timeout

ERROR: canceling statement
due to statement timeout

FATAL: terminating connection due to
transaction timeout

MONITOR VIA

```
pg_stat_activity  
WHERE state =  
  'idle in transaction'
```

2

STAGE 2 · SCRATCH SPACE

Resource Usage · Memory

In this section

```
shared_buffers      maintenance_work_mem  
work_mem            autovacuum_work_mem  
logical_decoding_work_mem
```

shared_buffers — the page cache Postgres actually owns

- Postgres reads through its own buffer cache before touching OS page cache.
- Rule of thumb: 25% of RAM on a dedicated server; OS cache also helps.
- Change requires restart
- *Wrong question: "Is shared_buffers big enough?"*
Right question: "What does my cache hit ratio look like?"

```
SELECT relname, heap_blks_read, heap_blks_hit,  
       round(100.0*heap_blks_hit/  
nullif(heap_blks_hit+heap_blks_read,0),2) AS hit_pct  
FROM pg_statio_user_tables ORDER BY heap_blks_read DESC LIMIT  
10;
```

WATCH IN LOGS

```
LOG: checkpoint complete:  
      buffers_written=...
```

```
auto_explain.log_buffers (if its enabled will  
log plan with buffer information)
```

MONITOR VIA

```
pg_stat_database  
  blks_hit, blks_read
```

```
pg_statio_user_tables
```

```
pg_buffercache (extension)
```

work_mem — the most-multiplied number in your conf

work_mem × N × M

N = concurrent backends · M = sort/hash nodes per plan

4 MB default × 200 conns × 3 hashes/plan ≈ 2.4 GB on the floor.

- Set per-session for analytics: `SET work_mem = '256MB';`
- Set per role: `ALTER ROLE etl SET work_mem = '512MB';`
- `hash_mem_multiplier` (PG 13+) lets hash ops take more without raising the global ceiling.

WATCH IN LOGS

```
LOG: temporary file: path
      "base/pgsql_tmp/...",
      size 134217728
STATEMENT: SELECT ...

(External merge Disk:
 ... in EXPLAIN ANALYZE)
```

MONITOR VIA

```
log_temp_files = 0

pg_stat_database
  temp_files, temp_bytes

EXPLAIN (ANALYZE, BUFFERS)
→ 'Sort Method:'
```

```
Aggregate (cost=5348342.29..5348342.30 rows=1 width=8) (actual time=77984.568..78001.306 rows=1 loops=1)
-> Unique (cost=1250433.88..5173254.71 rows=14007007 width=17) (actual time=24939.464..77045.024 rows=14448223 loops=1)
    -> Merge Join (cost=1250433.88..4898815.51 rows=54887840 width=17) (actual time=24939.462..69413.044 rows=53255128 loops=1)
        Merge Cond: ((cs_le.cs_company_id)::text = (cs_search.cs_company_id)::text)
        -> Gather Merge (cost=1250432.03..2934134.22 rows=14456539 width=17) (actual time=24932.628..41042.679 rows=14463238 loops=1)
            Workers Planned: 2
            Workers Launched: 2
            -> Sort (cost=1249432.00..1264490.90 rows=6023558 width=17) (actual time=24866.655..29748.967 rows=4821079 loops=3)
                Sort Key: cs_le.cs_company_id, cs_le.rank
                Sort Method: external merge  Disk: 102936kB
                Worker 0: Sort Method: external merge  Disk: 103736kB
                Worker 1: Sort Method: external merge  Disk: 103152kB
            -> Parallel Seq Scan on cs_legal_entities_2024 cs_le (cost=0.00..324048.58 rows=6023558 width=17) (actual time=1.000..1.000 rows=4821079 loops=3)
        -> Index Only Scan using cs_search_2024_cc_int_cs_company_id_idx on cs_search_2024 cs_search (cost=0.56..1204504.77 rows=53181 loops=1)
            Heap Fetches: 0
Planning Time: 0.632 ms
Execution Time: 78018.690 ms
```

[Copy source to clipboard](#)

maintenance_work_mem + autovacuum_work_mem

`maintenance_work_mem`

Used by VACUUM, CREATE INDEX, REINDEX, ALTER TABLE ADD FK.

default: 64MB

`autovacuum_work_mem`

Per autovacuum worker. If -1 (default), falls back to `maintenance_work_mem`

default: -1 (→ `maintenance_work_mem`)

THE INTERACTION

`maintenance_work_mem = 2GB + autovacuum_max_workers = 6 + autovacuum_work_mem = -1` → up to 12 GB of autovacuum RAM alone.

Set `autovacuum_work_mem` to a sane per-worker value (e.g. 512MB), keep `maintenance_work_mem` big for index builds.

WATCH IN LOGS

LOG: automatic vacuum of table "...": index scans: 3 ←
multiple passes = m_w_m too small

MONITOR VIA

`pg_stat_progress_vacuum` (phase = 'vacuuming index'), also
check `max_dead_tuple_bytes`

Till v16 : For the collection of dead tuple identifiers, autovacuum is only able to utilize up to a maximum of 1GB of memory, so setting `autovacuum_work_mem` to a value higher than that has no effect

logical_decoding_work_mem

- Per walsender doing logical decoding. Buffers transaction changes in memory before spilling to disk.
- Default 64 MB. Too small for long-running transactions or bulk loads → spill files on the publisher.
- Symptoms: 'reorderbuffer' spill files in `pg_replslot/<slot>/`; replication lag despite fast network.
- Streaming logical replication (PG 14+) eases this — but only if the subscriber supports it.

WATCH IN

```
(spill to disk — look at)
$PGDATA/pg_replslot/
  <slot_name>/xid-*.spill
```

MONITOR VIA

```
pg_stat_replication_slots
  spill_txns, spill_count,
  spill_bytes, stream_txns
pg_replication_slots
  confirmed_flush_lsn
```

3

STAGE 3 · THE CREW

Background Workers & Parallelism

In this section

```
max_worker_processes  
max_parallel_workers  
max_parallel_workers_per_gather    max_parallel_maintenance_workers
```

Three concentric limits — get the order right

`max_worker_processes`

All background workers in the cluster: parallel, logical replication apply, extensions.

`max_parallel_workers`

Workers available to parallel queries + maintenance (subset of the above).

`max_parallel_workers_per_gather`

Per query Gather node. 2 default.
work_mem correlation - per worker

`max_parallel_maintenance_workers`

CREATE INDEX, VACUUM. 2 default.
maintenance_work_mem correlation - per utility command

Order matters

Raising the inner two without raising `max_parallel_workers` does nothing — queries silently degrade to fewer workers than planned.

And neither limit matters if `max_worker_processes` is too low — that's the ceiling that also feeds logical replication apply workers.

WATCH IN LOGS

```
LOG: planned vs launched  
workers: 4 / 2 (in  
EXPLAIN ANALYZE)
```

MONITOR VIA

```
pg_stat_activity  
backend_type =  
'parallel worker'
```

4

STAGE 4 · THE DIARY

WAL · Checkpoints · Archiving

In this section

```
checkpoint_timeout  
archive_mode
```

```
max_wal_size  
archive_command
```

Two ways to trigger a checkpoint — pick the polite one

TIMED

checkpoint_timeout

Default 5 min → raise to 15–30 min for write-heavy (full page writes reduction)
Smooths I/O, but increases recovery time.

VOLUME

max_wal_size

Default 1GB → raise so timeout fires first.
If volume forces checkpoints, investigate the I/O load

The principle: checkpoints should be timed, not forced.

WATCH IN LOGS

LOG: checkpoints are occurring too frequently (12 seconds apart)
HINT: Consider increasing max_wal_size.

MONITOR VIA

```
pg_stat_checkpointer (PG 17+)
pg_stat_bgwriter      (pre-17)
  checkpoints_timed vs checkpoints_req
→ ratio >> 90% timed = healthy
```

```

postgres=# select * from pg_stat_checkpointer;
-[ RECORD 1 ]-----+-----
num_timed            | 717  ←
num_requested        | 11   ←
restartpoints_timed  | 0
restartpoints_req    | 0
restartpoints_done   | 0
write_time           | 421709
sync_time            | 1739
buffers_written      | 127799
stats_reset          | 2026-09-04 12:39:39.106882+00

```

Prior to v17 this information is in pg_stat_bgwriter view

5

STAGE 5 · TELLING THE REPLICA

Replication

In this section

```
max_wal_senders          max_replication_slots
wal_keep_size            max_slot_wal_keep_size
idle_replication_slot_timeout
max_standby_archive_delay max_standby_streaming_delay
hot_standby_feedback
```

Senders, slots, and the idle-slot timer

`max_wal_senders`

How many concurrent walsender processes. Each streaming replica, logical subscriber, or pg_basebackup uses one.

default: 10

`max_replication_slots`

How many slots can exist. Slots are the persistent bookmark a consumer leaves on the WAL.

default: 10

`idle_replication_slot_timeout` (PG 18+)

Auto-invalidates a slot whose consumer has been gone too long.

default: 0 (off)

Slots are persistent. Replicas may not be.

A dropped replica with a still-defined slot keeps holding WAL — until your disk says no. `idle_replication_slot_timeout` finally lets you bound that.

WATCH IN LOGS

ERROR: requested WAL segment 000... has already been removed

MONITOR VIA

`pg_replication_slots` · `pg_stat_replication`

wal_keep_size vs max_slot_wal_keep_size

wal_keep_size

Minimum WAL to retain for non-slot replicas

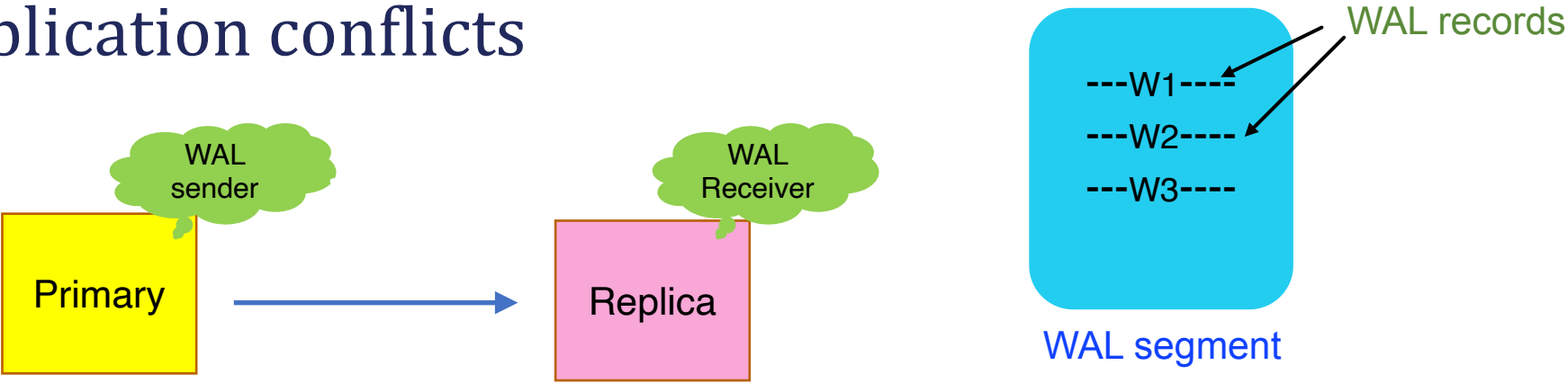
- Unconditional retention. WAL stays even if no one's reading it.
- Used when replicas don't have slots (rare in modern setups).
- Default 0 — slots are the modern path.

max_slot_wal_keep_size

Maximum WAL a slot is allowed to keep before being invalidated

- The safety valve. Above this, slot becomes invalidated and replica must re-sync.
- **Better to lose a replica than the primary.**
- Set to a fraction of free disk on pg_wal partition.

Understanding replication conflicts



Time	Primary	Primary WAL	Replica	Replica WAL
t0	-	Sending WAL record W1	Reading from "A"	Received WAL record W1
t1	WAL generating operation on "A"	Sending WAL record W2	Reading from "A"	Received W1 and W2, waiting to apply
t2	WAL generating operation completed	Sending WAL record W3	Still Reading from "A"	Still waiting to apply W1 and W2
t4	-	Sending WAL record W4	Cancelling statement due to conflict with recovery	Cancel query (based on conditions and after a threshold), and apply WAL

RECOVERY_CONFLICT_SNAPSHOT



Time	Primary	Replica
t0	-	Reading from "A"
t1	Update/Delete on "A"	Still Reading from "A"
t2	Vacuum triggered	Still Reading from "A"
t4	-	Cancelling statement due to conflict with recovery, as the rows need to be removed

Understanding replication conflicts

Long-running query on the replica vs WAL replay - Lag vs query cancellation - trade-off to make

max_standby_streaming_delay

How long replay will wait for queries to finish before canceling them, when WAL arrives via streaming. -1 = wait forever (lag grows).

default: 30s

max_standby_archive_delay

Same idea, but for WAL applied from the archive

default: 30s

hot_standby_feedback

Tells the primary the replica's oldest xmin so vacuum on the primary won't remove rows the replica still needs. Drawback = primary bloat

default: off

WATCH IN LOGS

ERROR: canceling statement due to conflict with recovery
(note the error detail carefully to find the fix)

MONITOR VIA

pg_stat_database_conflicts

Code

Blame



Raw



```
2553     errdetail_recovery_conflict(RecoveryConflictReason reason)
2554 {
2555     switch (reason)
2556     {
2557         case RECOVERY_CONFLICT_BUFFERPIN:
2558             errdetail("User was holding shared buffer pin for too long.");
2559             break;
2560         case RECOVERY_CONFLICT_LOCK:
2561             errdetail("User was holding a relation lock for too long.");
2562             break;
2563         case RECOVERY_CONFLICT_TABLESPACE:
2564             errdetail("User was or might have been using tablespace that must be dropped.");
2565             break;
2566         case RECOVERY_CONFLICT_SNAPSHOT:|
2567             errdetail("User query might have needed to see row versions that must be removed.");
2568             break;
2569         case RECOVERY_CONFLICT_LOGICALSLOT:
2570             errdetail("User was using a logical replication slot that must be invalidated.");
2571             break;
2572         case RECOVERY_CONFLICT_STARTUP_DEADLOCK:
2573             errdetail("User transaction caused deadlock with recovery.");
2574             break;
2575         case RECOVERY_CONFLICT_BUFFERPIN_DEADLOCK:
2576             errdetail("User transaction caused buffer deadlock with recovery.");
2577             break;
2578         case RECOVERY_CONFLICT_DATABASE:
2579             errdetail("User was connected to a database that must be dropped.");
2580             break;
2581     }
```

hot_standby_feedback

Lately, we've been getting issues on our replica database logs, with errors such as these:

```
Canceling statement due to conflict with recovery
```

```
2022-09-23 01:27:10 UTC:54.240.197.112(9953):[24188]:DETAIL:  
User query might have needed to see row versions that must be removed.
```

hot_standby_feedback will not cause these errors, but will cause primary bloat

6

STAGE 6 · CLEANING CREW

Autovacuum

In this section

```
autovacuum                                autovacuum_freeze_max_age
_worker_slots _max_workers _max_parallel_workers
_vacuum_threshold _scale_factor _vacuum_max_threshold
_analyze_threshold _analyze_scale_factor
```

The formula — when does autovacuum fire?

```
trigger = LEAST(
    autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor × reltuples,
    autovacuum_vacuum_max_threshold      -- PG 18+: hard cap on the trigger
)
```

_vacuum_threshold

Baseline dead tuples before considering. Tiny tables benefit from a small value.

default: 50

_vacuum_scale_factor

Fraction of reltuples added on top. 0.2 = 20% of table.

default: 0.2

_vacuum_max_threshold

PG 18+. Caps the trigger absolutely. Huge tables no longer wait until 20% to clean.

default: 100,000,000

_analyze_threshold / _scale_factor

Same math, for ANALYZE. Stats freshness matters as much as bloat.

default: 50 / 0.1

For busy tables, per-table override are a good direction:

```
ALTER TABLE orders SET (autovacuum_vacuum_scale_factor = 0.05,
    autovacuum_vacuum_threshold = 1000);
```

The formula — when does autovacuum fire?

```
trigger = LEAST(  
    autovacuum_vacuum_threshold + autovacuum_vacuum_scale_factor × reltuples,  
    autovacuum_vacuum_max_threshold      -- PG 18+: hard cap on the trigger  
)
```

For a 1 billion row table with defaults:

```
- `threshold + scale_factor × reltuples`  
= 50      + 0.2      × 1,000,000,000  
= 200,000,050 dead tuples before autovacuum fires
```

```
- With PG 18, default autovacuum_vacuum_max_threshold = 100,000,000` caps it = autovacuum fires at 100M dead instead of 200M
```

autovacuum_freeze_max_age — the clock you can't ignore

- Postgres XIDs are 32 bits. They wrap. To keep history visible, old tuples must be "frozen".
- When a table's oldest XID is older than `autovacuum_freeze_max_age`, an anti-wraparound vacuum kicks in
- **If they fall behind, the cluster will eventually refuse new XIDs and shut down to protect itself.**

WATCH IN LOGS

WARNING: database "db"
must be vacuumed within
N transactions

HINT: To avoid a database
shutdown, execute a
database-wide VACUUM

ERROR: database is not
accepting commands to
avoid wraparound...

MONITOR VIA

```
SELECT datname,  
       age(datfrozenxid)  
FROM pg_database  
ORDER BY 2 DESC;
```

Slots, workers, and the parallel sub-crew

`autovacuum_worker_slots` (PG 18+)

How many worker slots are reserved at startup. The pool the launcher draws from. Decouples capacity planning from runtime concurrency.

default: 16

`autovacuum_max_workers`

How many slots may be active at once. Pre-PG18 this was both the pool and the limit. Now: tune for concurrency.

default: 3

`autovacuum_max_parallel_workers` (PG 19+)

Across all autovacuum workers, how many parallel workers may help index cleanup. Bounded by `max_parallel_workers`.

default: 0

THE NEW MODEL

Reserve plenty of slots (`worker_slots`), keep runtime concurrency modest (`max_workers`), and turn on parallel index cleanup on huge tables. Workers borrow from `max_parallel_workers` — see Stage 3, the limits compose.

Score weights — autovacuum learns to prioritize

PG 19 replaces "first table whose trigger fired" with a weighted score. The launcher picks the most urgent work, not the next.

`autovacuum_vacuum_score_weight`

Weight on dead-tuple urgency (the bloat dimension).

default: 1

`autovacuum_analyze_score_weight`

Weight on stale-stats urgency (the plan dimension).

default: 1

`autovacuum_freeze_score_weight`

Weight on freeze-age urgency (the wraparound dimension).

default: 1

`autovacuum_multixact_freeze_score_weight`

Weight on multixact-age urgency (the multixact-wraparound dimension).

default: 1

The `pg_stat_autovacuum_scores` view shows the current scores of all tables in the current database.

What to watch — autovacuum at a glance

WATCH IN LOGS

```
LOG: automatic vacuum of
      table "x":
      index scans: 3
      pages: 0 removed, ...
tuples: N removed,
      M remain, K dead but
      not yet removable

      ← dead but not removable
        = a held xmin somewhere
          (replica? slot? long
            transaction?)

WARNING: database "db"
        must be vacuumed within
        N transactions
```

MONITOR VIA

```
pg_stat_user_tables
  n_dead_tup, n_live_tup,
  last_autovacuum,
  last_autoanalyze

pg_stat_progress_vacuum
  phase, heap_blks_scanned,
  num_dead_tuples

pg_stat_progress_analyze

log_autovacuum_min_duration = 0
  → every autovacuum run
    leaves a logged summary
```

7

STAGE 7 · PUT ON GLASSES

Reporting & Logging

In this section

```
log_min_duration_statement  
log_autovacuum_min_duration  
log_statement  
log_temp_files
```

The four parameters that make the rest of this talk visible

`log_min_duration_statement`

Log any statement that ran longer than N. -1 = off, 0 = log everything. Most people start at 1000 (ms) and walk it down.

default: -1

`log_autovacuum_min_duration`

Log any autovacuum that ran longer than N. 0 = log every run (recommended on busy systems).

default: 10min

`log_statement`

Categorical: 'none' | 'ddl' | 'mod' | 'all'. DDL on a production system is almost always cheap to log and invaluable for forensics.

default: none

`log_temp_files`

Log temp file creation over N bytes. 0 = log all. Companion to `work_mem` tuning.

default: -1

CHEAT SHEET

Error / log line	Look at parameter(s)	Open this view
FATAL: sorry, too many clients already	max_connections (and use a pooler)	pg_stat_activity
canceling statement due to statement timeout	statement_timeout / transaction_timeout	pg_stat_activity, pg_stat_statements
temporary file ... size NNN	work_mem, hash_mem_multiplier	pg_stat_database (temp_*)
checkpoints are occurring too frequently	max_wal_size, checkpoint_timeout	pg_stat_checkpoint / pg_stat_bgwriter
requested WAL segment ... has already been removed	max_slot_wal_keep_size, wal_keep_size	pg_replication_slots, pg_stat_replication
canceling statement due to conflict with recovery	max_standby_streaming_delay, hot_standby_feedback	pg_stat_database_conflicts
must be vacuumed within N transactions	autovacuum_freeze_max_age, autovacuum_max_workers	pg_database (age(datfrozenxid)), pg_stat_progress_vacuum

Key Takeaways

- **UPGRADE !!!** to the latest major version
- **idle_in_transaction_session_timeout** - to timeout sessions which can hold xid horizon
- Setup effective **monitoring** and **logging** for your database
- Maximum used transaction IDs should be monitored in **every** PostgreSQL database
- Do not tune in the dark

Thank you.

Questions, war stories, corrections — all welcome.

<https://ie.linkedin.com/in/divyasharma95>