# Write-Ahead Logging:-

**--How Databases Stay Fast and Reliable**

1.Most developers don't think about what happens under the hood when they call INSERT or UPDATE.

But the way databases write data is the difference between speed, reliability, and data loss.

2. That's where Write-Ahead Logging (WAL) comes in.

Instead of writing directly to disk every time, databases like PostgreSQL first write changes to a log (the WAL).

Only after this log is safely stored does the DB update its in-memory structures (page cache).

**--Why?**

- Speed: Writing to the WAL is a sequential disk operation, much faster than random writes to the data files.

- Reliability: If the system crashes, the WAL ensures all committed transactions can be replayed and no data is lost.

Meanwhile, a background process called a checkpoint runs quietly in the background.

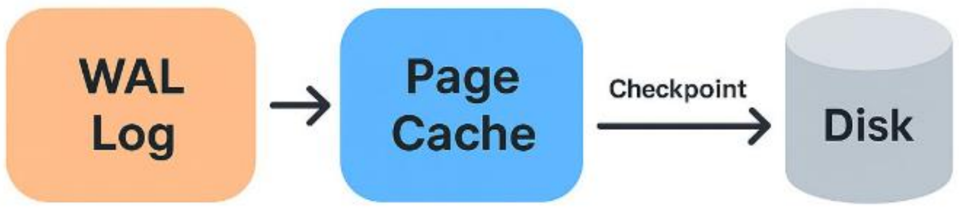
Its job is to flush updated pages from memory to the actual database files on disk.

**This two-step approach gives us the best of both worlds:-**

- Fast commits.

- Durability and consistency.

Without WAL, databases would either be painfully slow or constantly at risk of corruption.



The **key difference** between wal_buffers, max_wal_size, and min_wal_size in PostgreSQL in **simple language**:

---

- **Parameters and Their Roles**

| Parameter | What it Controls | Unit | Scope | Key Purpose |
|---|---|---|---|---|
| wal_buffers | Amount of **shared memory** used to temporarily store WAL (Write Ahead Log) records before writing them to disk. | Memory (KB/MB) | Per cluster | Optimizes write performance by buffering WAL records in memory. |
| max_wal_size | The maximum size of WAL files that can exist before PostgreSQL triggers an **automatic checkpoint**. | Disk space (MB/GB) | Per cluster | Prevents WAL files from growing indefinitely; controls checkpoint frequency. |
| min_wal_size | The minimum size of WAL files that PostgreSQL will keep, even after a checkpoint. | Disk space (MB/GB) | Per cluster | Avoids frequent creation/deletion of WAL segments (stability & performance). |

❖ **How They Work Together**

- **wal_buffers** → First stop in memory (small, fast, temporary).

Think of it like a small RAM queue for WAL records.

Default: -1 (auto-tuned: usually 1/32 of shared_buffers, up to 16MB).

- **max_wal_size** → WAL growth limit on disk.

If WAL grows beyond this, PostgreSQL forces a checkpoint.

Larger max_wal_size = fewer checkpoints = better throughput, but more recovery time after crash.

- **min_wal_size** → WAL files reserved on disk even when not needed.

Prevents constant recycle/delete/recreate of WAL files.

Acts as a floor limit for WAL retention.

❖ **Quick Example**

- Suppose min_wal_size = 1GB, max_wal_size = 4GB.
- During activity:

1. WAL fills up → Checkpoints run.
2. Files won't shrink below 1GB (min_wal_size).
3. If workload is high, WAL may grow up to 4GB (max_wal_size) before checkpoint is forced.

- In memory, WAL records are first staged in wal_buffers before being written to disk.

✅**In short:**

- wal_buffers = memory buffer for WAL before hitting disk.
- max_wal_size = upper limit of WAL files before checkpoint.
- min_wal_size = lower limit of WAL files kept to avoid churn.

================================================================================
=======

**Scenario: PostgreSQL Database with Heavy Inserts**

❖ **Settings:**

- wal_buffers = 16MB
- min_wal_size = 1GB
- max_wal_size = 4GB

**1. WAL Buffers in Action (Memory Stage)**

- You run a COPY command that inserts 500,000 rows into a table.

PostgreSQL doesn't write each row immediately to disk → instead, it puts the WAL changes into wal_buffers (16MB).

- Once buffers are full or a transaction commits, WAL is flushed from memory → WAL segment files (on disk).

? **Analogy:** Think of wal_buffers as a temporary basket in RAM. Rows go into the basket before being packed into WAL files.

**2. min_wal_size (Floor on Disk WAL Files)**

- After a checkpoint, PostgreSQL could delete/recycle unused WAL files.
- But with min_wal_size = 1GB, it always keeps at least 1GB worth of WAL files ready on disk.
- This avoids creating/deleting WAL files too often if your workload is bursty.

? **Example:**

- After your COPY, you generated 1.5GB WAL.
- Checkpoint occurs → database could shrink WAL usage.
- But since min_wal_size=1GB, it won't drop below 1GB of WAL segments.

**3. max_wal_size (Checkpoint Trigger)**

- Suppose your app suddenly runs huge batch jobs generating WAL continuously.
- WAL keeps accumulating on disk.
- When WAL files exceed 4GB (max_wal_size), PostgreSQL forces a checkpoint (flush dirty pages to disk, mark old WAL reusable).
- This ensures WAL files don't grow unbounded and eat up your storage.

� **Example:**

- Insert workload generates 5GB of WAL.
- At 4GB, PostgreSQL forces a checkpoint.
- After checkpoint, some WAL files are recycled (but not below min_wal_size=1GB).

� **Summarized Flow in Scenario:**

- WAL records → wal_buffers (16MB RAM).
- Flushed to WAL files on disk.
- WAL files accumulate →

    Always keep at least min_wal_size = 1GB.

    If growth exceeds max_wal_size = 4GB → trigger checkpoint.

✅ **In Plain Words:**

- wal_buffers = short-term memory basket.
- min_wal_size = how much WAL stock you always keep.
- max_wal_size = how much WAL you allow before forcing cleanup (checkpoint).

---

**you're noticing there are three different WAL–related size parameters (max_wal_size, min_wal_size, wal_keep_size) and wondering why wal_keep_size exists if min/max already control WAL size.**

Let's break it down with roles & scenarios:

� **Purpose of Each**

- **min_wal_size & max_wal_size** → Control checkpoints and WAL recycling for local durability & crash recovery.

- **wal_keep_size** → Controls WAL retention for replication (streaming replicas need past WAL files).

▯ Scenario Example (Replication in Play)
**Setup:**

**Primary server with:**

1. min_wal_size = 1GB
2. max_wal_size = 4GB
3. wal_keep_size = 512MB

One standby server replicating via streaming replication.

**▯ What Happens:**

- Normal Operation (min/max in play):
- Application generates WAL → stored in wal_buffers → written to disk.
- WAL files grow up to 4GB. At that point, checkpoint occurs → files recycled but at least 1GB is kept (min_wal_size).
- This ensures primary works efficiently.

**Replication Delay Introduced:**

- Suppose the standby is lagging (network issue, slow disk, etc.).
- Primary must keep old WAL files until standby catches up.

**This is where wal_keep_size matters:**

- PostgreSQL guarantees at least 512MB of WAL files are retained for standby use, even if they are older than what's required for crash recovery.

**Without wal_keep_size:**

- If primary recycles WAL files too soon (due to checkpoint + min/max policy), the standby might need a WAL that no longer exists.
- Result: standby falls out of sync and you must rebuild it from base backup. ❌

**With wal_keep_size:**

- Primary holds on to at least 512MB of WAL files specifically for replication, regardless of min_wal_size/max_wal_size.
- This gives replicas breathing room to lag safely without breaking replication. ✅

**▯ Key Difference in One Line**

- min_wal_size / max_wal_size → WAL retention for checkpoints & crash recovery (local node).

- wal_keep_size → WAL retention for replication safety (standbys).

 So, even if min/max_wal_size manage WAL size for the primary, you still need wal_keep_size when you have replicas — otherwise lagging replicas risk falling behind and breaking.

| Parameter | Too Low → Problem | Too High → Problem | DBA Notes / Best Practices |
|---|---|---|---|
| wal_keep_size (WAL reserved for replicas) | Standby falls behind → primary recycles needed WAL → **replication breaks** (standby must be reinitialized with base backup). | Wastes disk space on primary (old WAL files pile up even if replicas don't need them). | Always size for **max expected replication lag**. Eg: If standby can lag 30 min at ~100 MB/min WAL → set wal_keep_size ≥ 3GB. |
| min_wal_size (WAL floor on primary) | Too small = frequent WAL file create/delete cycles → unnecessary I/O overhead. | Holds more WAL files than needed, wasting space. | Use moderate value (1–2GB typical). Doesn't help replicas, only local efficiency. |
| max_wal_size (WAL ceiling before forced checkpoint) | Too small = frequent checkpoints → higher I/O, performance degradation. | Too large = fewer checkpoints → longer crash recovery time. | Balance between **checkpoint frequency** and **crash recovery tolerances**. Larger DBs often use 8–32GB. |

**Real-World Example**

- You have primary + DR standby across a WAN link.
- Workload generates ~200MB WAL/minute.
- Network hiccup causes standby to lag 15 minutes.

**Case 1: wal_keep_size = 256MB**

Primary recycles WAL every ~1–2 minutes (because standby is lagging).

After 15 min lag → standby requests WAL file that primary no longer has → replication breaks ❌

**Case 2: wal_keep_size = 4GB**

Primary retains at least 4GB WAL.

That's enough for 20 minutes lag at 200MB/min.

Standby can safely catch up once network recovers → replication continues ✅

✅**Golden Rule for DBAs:**

wal_keep_size = Replication safety net

min_wal_size = Don’t churn WAL files too much

max_wal_size = Keep checkpoints under control

**how max_wal_size and min_wal_size directly influence checkpoint triggering in PostgreSQL.**

- **Recap:** What a Checkpoint Is

A checkpoint flushes dirty pages from shared buffers to data files.

It also allows recycling/removing old WAL files.

- **Checkpoints can be triggered by:**

1. Time (checkpoint_timeout)
2. WAL growth (max_wal_size / min_wal_size)
3. Manual/other events (e.g., CHECKPOINT command, fast shutdown).

**? Role of max_wal_size and min_wal_size**

1. max_wal_size → Upper Bound (Triggers Checkpoint)

PostgreSQL keeps generating WAL.

If the total WAL generated since the last checkpoint exceeds max_wal_size,
→ it forces an immediate checkpoint (even if checkpoint_timeout hasn’t expired).

**? Example:**

- max_wal_size = 4GB
- DB workload writes 200MB WAL/minute.
- After ~20 minutes, 4GB WAL is generated → checkpoint is triggered.
- This ensures WAL does not grow indefinitely.

**2. min_wal_size → Lower Bound (Affects Recycling, Not Trigger)**

- After a checkpoint, PostgreSQL may recycle (reuse) or remove old WAL files.
- But it never shrinks below min_wal_size, even if workload is light.
- This avoids constant create/delete cycles for WAL segments.

**▯ Example:**

- min_wal_size = 1GB, max_wal_size = 4GB.
- Workload suddenly stops (low WAL generation).
- After checkpoint, PostgreSQL would normally shrink WAL usage down very low.
- But here it keeps 1GB worth of WAL files on disk, ready for reuse.

**▯ Note: min_wal_size does not trigger checkpoints; it just defines the floor of WAL recycling.**

**✅In Simple Words**

- max_wal_size: The ceiling → If WAL grows this much since last checkpoint → trigger a checkpoint.
- min_wal_size: The floor → After checkpoint, PostgreSQL always keeps at least this much WAL around (no matter what).

**▯ How They Work Together**

Let's simulate:

| WAL Activity | WAL Usage Growth | Effect |
|---|---|---|
| Start workload | WAL grows in size | Buffered → written to WAL files |
| WAL < max_wal_size | Checkpoint only by timeout (checkpoint_timeout) | Normal case |
| WAL ≥ max_wal_size | **Forces a checkpoint** | Prevents unbounded WAL growth |
| After checkpoint | WAL files recycled, but never less than min_wal_size | Efficient reuse |

**▯ DBA Tip:**

- If max_wal_size is too small → frequent checkpoints → performance drops.
- If max_wal_size is too large → fewer checkpoints, but crash recovery takes longer (because more WAL must be replayed).

- min_wal_size should be set high enough to avoid WAL churn, but not waste too much disk.

**WAL Flush vs Checkpoint in PostgreSQL**

| Feature | WAL Flush | Checkpoint |
|---|---|---|
| **When it happens** | On every commit, or when wal_buffers fills up. | On checkpoint_timeout, or if WAL since last checkpoint > max_wal_size, or manual (CHECKPOINT command). |
| **What is written** | WAL records (redo logs) → **WAL segment files** in pg_wal/. | Dirty pages from **shared_buffers → heap/index files** (main data files). |
| **Guarantee provided** | **Durability** → once WAL is on disk, committed data survives crash (can be replayed). | **Consistency & space control** → ensures data files are in sync with WAL, and WAL files can be recycled. |
| **Where it writes** | Always to pg_wal/ (disk). | To heap & index files (table data) in $PGDATA/base/…, and may recycle WAL in pg_wal/. |
| **Triggers** | Transaction commit, or synchronous_commit settings. | checkpoint_timeout, max_wal_size, manual trigger, or database shutdown. |
| **Effect on WAL files** | Creates/extends WAL segment files as needed. | Marks old WAL files as reusable → prevents unbounded growth (but keeps ≥ min_wal_size). |
| **Crash recovery role** | WAL flush ensures replay is possible after crash. | Checkpoint defines the **starting point** for crash recovery (don't need to replay from the beginning of WAL). |

✓Simple Analogy

- WAL Flush = You write every expense immediately in your notebook (WAL). Even before updating your bank ledger.
- Checkpoint = Every once in a while, you total up expenses and update your bank ledger (data files), then you can throw away some old notebook pages (recycle WAL).

- **DBA Implication**
- If checkpoints are happening too frequently because of WAL growth:

1. Either workload is generating too much WAL (heavy updates, large transactions, vacuum, etc.).
2. Or max_wal_size is set too low for that workload.

**Why Frequent Checkpoints Hurt Performance**

- When PostgreSQL hits a checkpoint too often (because max_wal_size is small or workload is WAL-heavy), these problems occur:

**1. I/O Spikes ("Checkpoint Storms")**

- At checkpoint, PostgreSQL flushes all dirty pages from shared_buffers to disk.
- If this happens too often, you get large bursts of random I/O → slowing down queries.
- Instead of a smooth trickle of writes, you get frequent write storms.

**2. WAL Churn (File Create/Delete Overhead)**

- Each checkpoint allows WAL recycling, but if it happens too often, PostgreSQL must constantly recycle or create WAL segment files.
- This causes extra disk I/O and CPU overhead.

**3. Increased Background Processes Workload**

- Background writer and checkpointer work harder, waking up more often.
- Causes higher system load (CPU + I/O).

**4. User Query Latency**

- If too much flushing happens at once, user queries may stall waiting for I/O.
- Latency-sensitive applications (e.g., OLTP systems) can feel this badly.

**5. Replication Lag**

- Frequent checkpoints create more WAL recycling pressure.
- If replicas lag even slightly, primary may need to retain more WAL segments → risk of bloat or replication break if wal_keep_size isn't tuned.

✅**What DBAs Usually Do**

1. Increase max_wal_size

- Lets PostgreSQL accumulate more WAL before triggering a checkpoint.
- Reduces checkpoint frequency → smoother performance.
- Trade-off: Crash recovery takes longer (more WAL to replay).

2. Tune checkpoint_timeout

- Default 5min (older versions) or 5–15min; DBAs often increase it (15–30min typical).
- Balances recovery time vs performance stability.

3. Spread Out Writes (Checkpoint Completion Target)

- Parameter: checkpoint_completion_target (default = 0.5).
- Controls how evenly the checkpoint I/O is spread over time.
- Example: Set to 0.7–0.9 → smoother, less spiky I/O.

4. Monitor WAL Generation

- Check with:

```
SELECT pg_current_wal_lsn(), pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn)
FROM pg_stat_replication;
```

- Know your workload's WAL generation rate (MB/min).

5. Size WAL with Workload in Mind

- Rule of thumb:
- max_wal_size ≈ (WAL generation per minute × desired checkpoint interval in minutes)

- Example: If DB generates 500MB WAL/min, and you want 30min between checkpoints →

max_wal_size ≈ 15GB.

✅Summary for DBAs:

- Frequent checkpoints = bad (spiky I/O, query stalls, WAL churn).
- Fix = increase max_wal_size, spread I/O with checkpoint_completion_target, and monitor WAL generation.

**Why WAL Piles Up in pg_wal/**

**1⃞ Checkpoints Too Infrequent**

⍰ How it works:

- WAL is written to disk continuously on every commit.
- Checkpoint = the moment PostgreSQL can recycle (reuse) old WAL files.
- If checkpoints don't happen often enough (because checkpoint_timeout is high or max_wal_size is large), WAL segments keep accumulating.

❖ ⍰ Example:

- Workload: 500MB WAL/min.
- checkpoint_timeout = 30min, max_wal_size = 10GB.
- In 30 minutes → 15GB WAL is generated.
- But PostgreSQL won't recycle until either 30min timeout or 10GB max_wal_size hits.
- Result → pg_wal directory keeps growing with dozens/hundreds of WAL files.

✅**Reason:**
Old WAL files are still needed for crash recovery until checkpoint occurs → so PostgreSQL can't recycle them yet.

**2⃞ Standby (Replica) Lagging**

**⍰ How it works:**

- In replication, primary must keep WAL files until all standbys confirm they have received/replayed them.
- If a standby is slow (network lag, apply lag) → primary cannot recycle old WAL yet.

- Even if checkpoints happen, PostgreSQL holds onto extra WAL files to serve the lagging standby.

❖ ▯ Example:

- Primary generates 200MB WAL/min.
- Standby falls behind by 30min.
- Primary must retain at least 6GB of WAL (30 × 200MB) so the standby can catch up.
- So WAL piles up in pg_wal/, even if checkpoints run on time.

✅**Reason:**
Old WAL files are still needed for replication catch-up → PostgreSQL can't recycle them.

**Summary**

| Cause | Why WAL Piles Up | DBA Fix |
|---|---|---|
| **Checkpoints infrequent** | WAL needed for crash recovery until next checkpoint → files not recycled. | Tune max_wal_size, checkpoint_timeout, checkpoint_completion_target. |
| **Standby lagging** | WAL needed by replica → primary can't recycle until replica catches up. | Tune wal_keep_size, fix replication lag (network/IO tuning). |

**Golden Rule:**

- If you see WAL piling up but no replicas exist → checkpoint tuning issue.
- If you see WAL piling up and replicas exist → standby lag issue.