Deep Dive into PostgreSQL WAL Parameters: `wal_writer_delay`, `wal_writer_flush_after`, and `wal_skip_threshold`

- Efficient Write-Ahead Logging (WAL) is crucial for PostgreSQL performance, durability, and crash recovery. In this post, we explore three WAL-related parameters that affect how and when data is flushed to disk: `wal_writer_delay`, `wal_writer_flush_after`, and `wal_skip_threshold`.

Background: What is WAL?

WAL (Write-Ahead Logging) is the mechanism PostgreSQL uses to ensure data durability and crash recovery. Every change made to the database is first recorded in WAL before being written to the actual data files.

Key components involved in WAL handling:

- **Backend processes**: Write WAL to shared memory.
- **WAL Writer process**: Flushes WAL from memory to disk.
- **Checkpointer**: Periodically writes dirty pages to data files.
- **Archiver** (optional): Archives WAL segments for PITR.

□ 1. `wal_writer_delay`

□ Definition:

```
wal_writer_delay = 200ms  # Default: 200ms
```

□ Explanation:

This setting controls **how often the WAL writer wakes up** to flush WAL from the **WAL buffers in shared memory** to disk.

If `wal_writer_flush_after` is not triggered first, WAL will be flushed after this delay.

□ Behavior:

Condition WAL is written to disk? 200ms passed ✓
Yes `wal_writer_flush_after` triggered earlier ✓ Yes (immediate flush)

□ Risk:

- Longer delay → Higher chance of WAL loss in crash (though committed transactions are still safe).
- Shorter delay → Increased I/O, but safer.

✓Best Practice:

Keep the default (`200ms`) unless you're tuning for ultra-low-latency systems.

❓ 2. `wal_writer_flush_after`

📌 Definition:

```
wal_writer_flush_after = 1MB  # Default: 1MB
```

🔍 Explanation:

This setting specifies how much WAL data the writer should accumulate **before forcing a flush to disk** (calling `fsync()` or similar).

- Measured in bytes (even though the doc says "pages").
- If 1MB of WAL is written, PostgreSQL will **flush immediately**, without waiting for the full `wal_writer_delay`.

🧠 Behavior:

Data written >= 1MB Flush immediately Yes ✅ Yes No ❌ Wait for `wal_writer_delay`

🔧 What if set to `0`?

- Flushing is **disabled**.
- WAL is still **written**, but not **fsync'ed** — relies on the OS page cache.
- ❌ Dangerous in crashes — WAL not guaranteed to reach disk.

✅Best Practice:

Keep default (1MB) unless you have specific storage behavior you're optimizing for.

❓ 3. `wal_skip_threshold`

🔹 Definition:

```
wal_skip_threshold = 10MB  # Default: 2MB (in newer versions)
```

🔹 Explanation:

This controls whether PostgreSQL **skips WAL logging** during bulk operations like:

- `COPY`
- `CREATE TABLE AS SELECT`
- `pg_restore` with `--disable-triggers`

If the size of data written in such operations exceeds this threshold **and the table is new and not yet visible to others**, WAL logging **is skipped**.

✅Example:

```
COPY big_table FROM '/path/large.csv' WITH CSV;
-- If big_table is new and file > 10MB → WAL is skipped
```

❌But not for:

- Normal `INSERT`/`UPDATE`
- Existing tables
- Small loads (e.g., <10MB)

⚠ Risk:

- Skipping WAL = **no crash recovery** for that data.
- **PITR is not possible** until next base backup.
- Not replicated to streaming replicas.

✅Best Practice:

- Keep it in sync with your typical load size.
- Always take a **base backup** after bulk loads if WAL is skipped.

🧠 Combined Scenario: What Happens?

Let's assume the following settings:

```
wal_writer_delay = 200ms
wal_writer_flush_after = 1MB
wal_skip_threshold = 10MB
```

🔍 Scenario:

1. A `COPY` command loads 20MB of data into a **new table**.
2. Since 20MB > 10MB → WAL is skipped.
3. Meanwhile, other backend activity triggers WAL.
4. As soon as 1MB of WAL is accumulated → WAL writer flushes to disk (even if 200ms hasn't passed).
5. If `wal_writer_flush_after` was `0`, it would wait the full 200ms (or possibly longer) before flushing.

🔄 What If I Want to Avoid Dropping the Database During Restore?

We also covered a common related issue during restore:

❌Problem:

```
pg_restore --clean -d mydb dumpfile.dump
# ERROR: cannot drop database because it is being accessed
```

✅Solution:

Instead of dropping the DB, use:

```
pg dump --clean -Fc -f mydb.dump mydb
pg_restore --no-owner --no-privileges -d mydb mydb.dump
```

This will:

- Drop individual objects
- Not drop the database
- Work even if other users are connected (as long as they're not using specific objects)

Or better yet — create a **wrapper script** that:

- Revokes new connections
- Terminates existing sessions
- Performs restore
- Grants connections back

✅Summary Table

Parameter Purpose Risk Best Practice `wal_writer_delay` Controls flush timing (ms) Longer delay = more WAL in memory Default (200ms) fine `wal_writer_flush_after` Immediate flush after X MB `0` disables safe flushing 1MB good default `wal_skip_threshold` Skip WAL during bulk loads No PITR; not replicated Tune per bulk size, always back up

□ Final Thoughts

Tuning these WAL parameters is an **advanced but powerful way** to balance:

- **Performance** (less fsync, better I/O)
- **Durability** (WAL integrity)
- **Backup & recovery** strategy (avoid data loss)

These should always be adjusted in coordination with:

- Your **workload profile** (OLTP vs OLAP)
- Your **hardware** (SSD vs HDD, write cache)
- Your **recovery requirements** (PITR, HA, streaming)