PostgreSQL Troubleshooting Runbook Pack

A senior engineer's incident runbook for diagnosing a struggling PostgreSQL database under load. DevOps AI ToolKit · devopsaitoolkit.com

When Postgres misbehaves the symptom is rarely the cause. "Too many clients", a hung query, or a full disk are all downstream of something more specific — a connection leak, a lock chain, a stuck autovacuum, a failing archive command. Work from the live session state outward: see what the backends are actually *doing* right now, then follow the evidence to the subsystem (locks, plans, replication, vacuum, WAL, auth, memory) that explains it. Nearly every check below is read-only and safe to run mid-incident; the few that change state are flagged.

> Examples assume a primary `db-1:5432` with database `appdb`, connecting as a superuser or `pg_monitor` role. Run the read-only queries first. `pg_terminate_backend()` and `VACUUM FULL` are disruptive — treat them as deliberate, recorded actions, not reflexes.

1. First-Response Triage

Start with one question: what are the backends doing? `pg_stat_activity` is the pulse.

```
-- What is running, waiting, and how long has it been going?
SELECT pid, usename, state, wait_event_type, wait_event,
       now() - xact_start  AS xact_age,
       now() - query_start AS query_age,
       left(query, 80) AS query
FROM pg_stat_activity
WHERE state <> 'idle' AND backend_type = 'client backend'
ORDER BY xact_age DESC NULLS LAST
LIMIT 25;

-- Connection headroom and per-state counts
SELECT count(*) AS total,
       count(*) FILTER (WHERE state = 'active')              AS active,
       count(*) FILTER (WHERE state = 'idle in transaction')  AS idle_in_txn,
       (SELECT setting::int FROM pg_settings WHERE name='max_connections') AS max_conn
FROM pg_stat_activity;

-- Commit/rollback/deadlock rates and cache hit ratio for the DB
SELECT numbackends, xact_commit, xact_rollback, deadlocks,
       blks_hit, blks_read,
       round(100.0*blks_hit/nullif(blks_hit+blks_read,0),2) AS cache_hit_pct
FROM pg_stat_database WHERE datname = 'appdb';
```

- Long `xact_age` with `state = 'idle in transaction'` → an app is holding a transaction open. That single fact explains most lock, bloat, and connection pileups.
- `wait_event_type = 'Lock'` on many rows → jump to §3. `'IO'` waits → §8 or §10.
- `deadlocks` climbing or `cache_hit_pct` below ~99% on an OLTP box → §3 / §10.

2. Connection Exhaustion

`FATAL: sorry, too many clients already` means all `max_connections` slots are taken. The real question is *why* — usually a pool sized larger than the DB, or leaked idle-in-transaction sessions.

```
-- Who is holding connections, grouped by app and state
SELECT application_name, usename, state, count(*)
FROM pg_stat_activity
GROUP BY 1,2,3 ORDER BY count(*) DESC;

-- Idle-in-transaction sessions older than 5 min (leak suspects)
SELECT pid, usename, application_name,
       now() - state_change AS idle_for, left(query,60) AS last_query
FROM pg_stat_activity
WHERE state = 'idle in transaction'
  AND now() - state_change > interval '5 minutes'
ORDER BY idle_for DESC;
```

- Raising `max_connections` is almost never the fix — each backend costs memory (§10) and Postgres does not love thousands of them. Put **PgBouncer** in front (transaction pooling) and size the pool to a few × CPU cores.
- Kill a confirmed leaked session only after identifying it (read-only cancel first, then terminate):

```
SELECT pg_cancel_backend(<pid>);      -- polite: cancels the current query
SELECT pg_terminate_backend(<pid>);   -- forceful: drops the whole session
```

- Prevent recurrence with `idle_in_transaction_session_timeout` (e.g. `'60s'`) so Postgres reaps leaks itself. Reserve headroom with `superuser_reserved_connections` so you can still log in during a flood.

3. Locks & Deadlocks

A query that "hangs" is usually blocked, not slow. Find the blocker at the head of the chain — terminating a mid-chain waiter does nothing.

```
-- Blocking → blocked pairs (the classic wait-for graph)
SELECT blocked.pid          AS blocked_pid,
       blocked.query        AS blocked_query,
       blocking.pid         AS blocking_pid,
       blocking.query       AS blocking_query,
       now() - blocked.query_start AS blocked_for
FROM pg_stat_activity blocked
JOIN pg_stat_activity blocking
  ON blocking.pid = ANY(pg_blocking_pids(blocked.pid))
WHERE cardinality(pg_blocking_pids(blocked.pid)) > 0
ORDER BY blocked_for DESC;

-- Raw lock view when you need lock modes / relations
SELECT l.pid, l.mode, l.granted, c.relname
FROM pg_locks l LEFT JOIN pg_class c ON c.oid = l.relation
WHERE NOT l.granted ORDER BY l.pid;
```

- `pg_blocking_pids()` gives the blockers directly — the head of the chain is the one that is *not itself blocked*. That is the session to cancel or terminate.
- Deadlocks are auto-resolved by Postgres (one txn is aborted after `deadlock_timeout`, default `1s`) and logged with both statements — read the log, don't just retry. Recurring deadlocks mean two code paths lock rows in opposite order; fix the ordering.
- Set a defensive `lock_timeout` (e.g. `SET lock_timeout = '3s';`) in migrations and batch jobs so a long lock wait fails fast instead of piling up blockers.

---

4. Slow Queries & Plans

Enable `pg_stat_statements` (shared_preload_libraries) once — it is the single best diagnostic Postgres offers.

```
-- Top time consumers since last stats reset
SELECT round(total_exec_time)::int AS total_ms, calls,
       round(mean_exec_time)::int  AS mean_ms, rows,
       left(query, 90) AS query
FROM pg_stat_statements
ORDER BY total_exec_time DESC LIMIT 15;
```

Then inspect the actual plan of the worst offender with real timings and buffers:

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE)
SELECT * FROM orders WHERE customer_id = 42 AND status = 'open';
```

- `Seq Scan` on a large table with a selective `WHERE` → a missing or unusable index. Confirm with `pg_stat_user_tables.seq_scan` vs `idx_scan`.
- `EXPLAIN ANALYZE` estimated rows wildly off actual rows → stale statistics; run `ANALYZE orders;` (cheap, safe).
- High `shared read` in `BUFFERS` → data isn't cached (§10). Find never-used and duplicate indexes via `pg_stat_user_indexes.idx_scan = 0` before adding more — every index also slows writes and vacuum.

---

5. Replication Lag

A lagging standby risks stale reads and a longer RPO. Diagnose from the primary first.

```
-- On the PRIMARY: per-standby send/flush/replay lag
SELECT client_addr, application_name, state, sync_state,
       write_lag, flush_lag, replay_lag,
       pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), replay_lsn)) AS replay_bytes
FROM pg_stat_replication;

-- Replication slots — a stuck/inactive slot pins WAL forever
SELECT slot_name, active, wal_status,
       pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)) AS retained_wal
FROM pg_replication_slots;
```

```
-- On the STANDBY: how far behind in time is replay?
SELECT now() - pg_last_xact_replay_timestamp() AS replay_delay;
```

- `replay_lag` high but `flush_lag` low → the standby *received* WAL but replay is stalled, often by a query on the standby conflicting with replay. `hot_standby_feedback = on` reduces cancellations at the cost of some primary bloat.
- A slot with `active = false` and growing `retained_wal` will fill the primary's disk (§8). Drop truly dead slots with `SELECT pg_drop_replication_slot('name');` — but only once you're sure that standby is gone for good.
- Sustained lag with no conflict → the standby is I/O- or CPU-bound, or the network is saturated. Check its `pg_stat_activity` for `wait_event = 'Recovery*'`.

---

6. Autovacuum & Bloat

Dead tuples from updates/deletes aren't reclaimed until vacuum runs. When autovacuum falls behind, tables and indexes bloat and plans degrade.

```
-- Dead-tuple ratio and last (auto)vacuum/analyze per table
SELECT relname, n_live_tup, n_dead_tup,
       round(100.0*n_dead_tup/nullif(n_live_tup+n_dead_tup,0),1) AS dead_pct,
       last_autovacuum, last_autoanalyze, autovacuum_count
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC LIMIT 15;

-- Is an autovacuum running right now, and is it blocked?
SELECT pid, now() - query_start AS running_for, wait_event, query
FROM pg_stat_activity WHERE query ILIKE 'autovacuum:%';
```

- High `dead_pct` with an old `last_autovacuum` → autovacuum can't keep up. It is often throttled by conservative `autovacuum_vacuum_cost_delay` / `cost_limit`, or starved by too few `autovacuum_max_workers`.
- A long-running transaction or idle-in-transaction session (§1) holds back the vacuum horizon so dead rows *cannot* be removed anywhere. Clear that first — tuning autovacuum won't help while an old snapshot is pinned.
- `VACUUM (VERBOSE, ANALYZE) tablename;` is online and safe. `VACUUM FULL` rewrites the table and takes an `ACCESS EXCLUSIVE` lock (full outage on that table) — schedule it in a window, or use `pg_repack` for online compaction.

---

7. Transaction ID Wraparound

The most dangerous Postgres failure mode: if the oldest unfrozen XID ages past ~2 billion, Postgres shuts down writes to protect data. It always announces itself in the logs first — never ignore a wraparound warning.

```
-- Databases closest to wraparound (watch the frozen-XID age)
SELECT datname, age(datfrozenxid) AS xid_age,
       round(100.0*age(datfrozenxid)/2.0e9,1) AS pct_toward_wrap
FROM pg_database ORDER BY xid_age DESC;

-- Tables driving a database's age
SELECT relname, age(relfrozenxid) AS xid_age
FROM pg_class WHERE relkind IN ('r','m','t')
ORDER BY xid_age DESC LIMIT 15;
```

- `autovacuum_freeze_max_age` (default 200M) triggers an anti-wraparound autovacuum automatically; ages far above it mean freezing isn't completing — usually the same culprits as §6 (throttling, or a long-held snapshot blocking cleanup).
- Approaching the limit: run a manual freeze on the worst tables — `VACUUM (FREEZE, VERBOSE) big_table;`. This is online but I/O-heavy; do the largest tables first.
- If Postgres has already stopped accepting writes, restart in single-user mode and `VACUUM` per the wraparound message. This is a genuine emergency — page the DBA/on-call lead.

---

8. Disk Full & WAL Growth

`could not extend file ... No space left on device` or `PANIC: could not write to file "pg_wal/..."` mean the data or WAL volume is full. WAL that can't recycle is the usual runaway.

```
# Sizes: data dir, and WAL specifically
df -h /var/lib/postgresql
du -sh /var/lib/postgresql/*/main/pg_wal

# Is archiving failing? failed_count climbing = WAL can't be recycled
psql -h db-1 -d appdb -c \
  "SELECT archived_count, failed_count, last_failed_wal, last_failed_time FROM pg_stat_archiver;"
```

- Growing `pg_wal` almost always has one of three causes: (a) a failing `archive_command` (fix the archive target — Postgres will *not* delete un-archived WAL), (b) an inactive replication slot pinning WAL (§5), or (c) `wal_keep_size` set very high.
- **Never** `rm` files in `pg_wal/` — that corrupts the cluster. Fix the archive/slot cause and Postgres recycles WAL on the next checkpoint.
- To buy emergency space, delete OS-level junk (old logs, archives already copied) — not database files. Then address the root cause before the volume refills.

---

9. Authentication & Connection

`FATAL: password authentication failed`, `no pg_hba.conf entry for host`, or SSL errors are config, not load. They're deterministic — reproduce with a precise `psql`.

```
# Reproduce with an explicit host/user/db so the FATAL is unambiguous
psql "host=db-1 port=5432 dbname=appdb user=appuser sslmode=require"

# On the server: inspect the rules and reload after edits (no restart needed)
psql -h db-1 -c "SELECT type, database, user_name, address, auth_method FROM pg_hba_file_rules WHERE error IS NULL;"
psql -h db-1 -c "SELECT pg_reload_conf();"
```

- `no pg_hba.conf entry` → rules are matched **top-to-bottom, first match wins**. A broad `reject` above your rule silently blocks it; order matters as much as content.
- `password authentication failed` with correct credentials → often a `scram-sha-256` vs `md5` mismatch after an upgrade, or a role whose password predates the SCRAM switch. Reset with `ALTER ROLE appuser PASSWORD '...';`.
- SSL errors → check `ssl = on`, cert file permissions (key must be `0600`, owned by `postgres`), and the client's `sslmode`. `pg_hba_file_rules` surfaces parse errors without a reload.

---

10. Memory / OOM & Checkpoints

If the OOM killer takes out a backend (`server process was terminated by signal 9: Killed`), Postgres restarts into crash recovery and every connection drops. The cause is usually over-committed `work_mem` × concurrency, not `shared_buffers`.

```
# Was it the kernel OOM killer?
dmesg -T | grep -i "out of memory\|oom-killer\|Killed process"
journalctl -u postgresql --since "30 min ago" | grep -iE "signal 9|recovery|checkpoint"
```

```
-- Key memory + checkpoint settings
SELECT name, setting, unit FROM pg_settings
WHERE name IN ('shared_buffers','work_mem','maintenance_work_mem',
               'effective_cache_size','max_connections',
               'checkpoint_timeout','max_wal_size','checkpoint_completion_target');

-- Are checkpoints firing too often? (req >> timed = under-sized max_wal_size)
SELECT num_timed, num_requested, write_time, sync_time FROM pg_stat_checkpointer;
```

- `work_mem` is **per sort/hash node, per connection** — a 256MB `work_mem` with 200 active backends and multi-node plans can request tens of GB. Keep it modest and raise it per-session for known-heavy jobs.
- `shared_buffers` ~25% of RAM and `effective_cache_size` ~50–75% is the usual OLTP starting point. Growing `shared_buffers` past that rarely helps and steals RAM from the OS cache.
- `num_requested` far exceeding `num_timed` → checkpoints are triggered by WAL pressure; raise `max_wal_size` and keep `checkpoint_completion_target = 0.9` to smooth I/O spikes.

---

11. Escalation Workflow

1. **0–10 min:** Snapshot `pg_stat_activity` (§1) and post the longest-running / blocking sessions in the incident channel. Note connection count vs `max_connections`.
2. **10–20 min:** Localize the subsystem — locks (§3), a slow plan (§4), WAL/disk (§8), or auth (§9). Cancel (never blind-terminate) the narrowest blocking session with a change note; re-test.
3. **20–30 min:** If it's replication (§5), wraparound (§7), or an OOM/crash loop (§10), escalate to the DB owner. Do **not** `rm` WAL, `VACUUM FULL` a hot table, or bump `max_connections` blindly under pressure.
4. **30+ min or data-facing:** Page the on-call lead, open a Sev-2, and start the incident notes template below. Capture `EXPLAIN ANALYZE` and the relevant log lines before anything is restarted.
5. **Stuck?** Bring in a second set of senior eyes: devopsaitoolkit.com/work-with-me, or paste the errors into the free AI Incident Response assistant at devopsaitoolkit.com/dashboard/incident-response.

---

12. Diagnostic Decision Tree

```
Postgres incident?
 ├─ "too many clients" / connections at max? ── yes → §2: pool w/ PgBouncer; reap idle-in-transaction; DON'T just raise max_connections
 └─ no
    ├─ queries hanging, not slow? ── yes → §3: pg_blocking_pids → cancel the HEAD of the chain
    ├─ specific query slow? ──────── yes → §4: pg_stat_statements → EXPLAIN (ANALYZE, BUFFERS) → index/ANALYZE
    ├─ standby behind? ───────────── yes → §5: pg_stat_replication + slots; check hot_standby_feedback / inactive slot
    ├─ writes refused, wraparound log? yes → §7: age(datfrozenxid) → VACUUM FREEZE; EMERGENCY, page DBA
    ├─ "no space left" / pg_wal huge?  yes → §8: pg_stat_archiver + slots; fix archive_command; NEVER rm pg_wal
    ├─ FATAL auth / no pg_hba entry? ── yes → §9: pg_hba_file_rules (first match wins) → pg_reload_conf()
    ├─ backend killed / crash recovery? yes → §10: dmesg OOM → work_mem × concurrency; tune checkpoints
    └─ bloat / stale plans / dead tups? yes → §6: dead_pct + last_autovacuum; clear old snapshot first
```

---

13. Copy/Paste One-Shot Triage

```
# Read-only. Run against the primary. Substitute your host/db.
psql -h db-1 -d appdb -X <<'SQL'
\echo == connections / headroom ==
SELECT count(*) total,
       count(*) FILTER (WHERE state='active') active,
       count(*) FILTER (WHERE state='idle in transaction') idle_in_txn,
       (SELECT setting::int FROM pg_settings WHERE name='max_connections') max_conn
FROM pg_stat_activity;
\echo == longest running (non-idle) ==
SELECT pid, state, now()-xact_start AS xact_age, wait_event, left(query,60)
FROM pg_stat_activity WHERE state<>'idle' AND backend_type='client backend'
ORDER BY xact_age DESC NULLS LAST LIMIT 5;
\echo == blocking chains ==
SELECT pid, pg_blocking_pids(pid) AS blocked_by, left(query,50)
FROM pg_stat_activity WHERE cardinality(pg_blocking_pids(pid))>0;
\echo == xid wraparound risk ==
SELECT datname, age(datfrozenxid) FROM pg_database ORDER BY 2 DESC LIMIT 3;
\echo == WAL archiver ==
SELECT archived_count, failed_count, last_failed_time FROM pg_stat_archiver;
\echo == inactive replication slots ==
SELECT slot_name, active, wal_status FROM pg_replication_slots WHERE NOT active;
SQL
```

14. Incident Notes Template

```
INCIDENT: PostgreSQL degradation / outage
Started (UTC):          ____   Detected by: ____   Sev: ____
Host / DB:              db-1:5432 / appdb
Impact:                 (which app paths / % of queries / read or write)
Connections:            __/__ max   idle_in_txn: ____
Symptom localized to:   [ ] connections [ ] locks [ ] slow query [ ] replication
                        [ ] autovacuum/bloat [ ] wraparound [ ] disk/WAL [ ] auth [ ] OOM/checkpoint
Key evidence:           blocking pid ____  slow query ____  replay_lag ____
                        xid_age ____  failed archives ____  OOM in dmesg? Y/N
Action taken:           (cancel/terminate pid / index / freeze / config + reload + who + when)
Result:                 [ ] recovered @ ____   [ ] escalated to ____
Root cause:             ____
Follow-ups:             (pooling, indexes, autovacuum tuning, slot cleanup, alerts) ____
```