

DIY Postgres Branching

Radim Marek

Skeptic, Turned Believer

boringSQL

Why

- Wrote **Instant database clones with PostgreSQL 18**
- My most-read article
- I knew about ZFS, but never really tried it for branching at scale
- This talk is about what comes next
- And comes as prep for the new article

PostgreSQL 18 made one level easy.

There's a second level most people skip.

This talk covers both, and tells you which one you actually need.

What does branching actually mean?

```
git:      checkout -b feat/foo   →  2ms    ✓ branch   ✓ merge
Docker:   FROM base              →  fast   ✓ branch   ✗ merge
Database: ???                    →  ???    ✗ branch   ✗ merge
```

Git taught us to think in branches. Docker made infrastructure forkable.

Databases are the one place it took a decade and a SaaS subscription.

The SaaS answer

- **DbLab / Postgres.ai**: "Thin clones for developers" · self-hosted · ~2019
- **Neon**: "Instant branching in milliseconds" · $19–69/month · 2022
- **Xata**: "Branch your database like you branch your code" · $20+/month · 2022

Same pitch, different implementations. All three chose **cluster-level**. Once you understand why, the differences are obvious.

The shared staging database is everyone’s problem

- Migrations tested against empty fixtures, not real data
- One destructive test blocks everyone’s PR
- "Don’t run that on staging" is a real sentence people say
- PR review without data is just reading code

And now agents run your migrations. Autonomously. In parallel. On the same database.

Agents can’t be afraid to touch staging. They just touch it.

When Databricks bought Neon in 2025, **over 80%** of Neon databases were created by AI agents.

→ github.com/boringsql/regresql: SQL query regression testing

- Empty fixtures hide plan regressions: the planner chooses differently on 100 rows vs. 10 million
- You can’t tell from reading SQL which index it will use, or whether it will seq-scan

boringSQL

Two ways to copy a database: clone or branch

Database-level → clone: copy one database inside a running cluster

- Fast, same Postgres process, no new instance
- Shares the cluster's history
- Limited option for live databases

Cluster-level → branch: snapshot the whole data directory, start a new Postgres

- Gets its own `pg_xact`, its own WAL, its own lifecycle
- Works on a live source

Part 1: Why you can’t clone a live database

Why does the rule exist?

I tried to remove a safety check from PostgreSQL. I was wrong.

```
CREATE DATABASE test_run TEMPLATE myapp STRATEGY = FILE_COPY;
```

```
ERROR: source database "myapp" is being accessed by other users
DETAIL: There is 1 other session using the database.
```

Not "no active queries." No **sessions**. One idle pool connection is enough.

Finding out why changed how I see every database branching product out there.

pg_xact: PostgreSQL's commit log

Every row knows which transaction wrote it. `pg_xact` answers one question: **did that transaction commit?**

(each row stores `xmin`: inserted by, `xmax`: deleted by)

The critical detail:

```
$PGDATA/
  base/16384/…    ← one database's files
  pg_xact/        ← commit log. Cluster-wide. Shared by every database.
  pg_wal/         ← write-ahead log. Also cluster-wide.
```

One database's files live in `base/16384/`. `pg_xact` does not.

→ boringsql.com/posts/postgresql-mvcc-byte-by-byte

Why FILE_COPY can’t escape the cluster

`FILE_COPY` copies one database’s files. The clone still reads visibility from the **shared** `pg_xact`.

A transaction in-flight at copy time can commit *after* the copy. The clone sees the row appear. No write of its own.

Include `pg_xact` and it stops being a clone. It’s a branch.

That means the **whole cluster**, not just one database. That’s why every branching vendor chose cluster-level.

What a clone actually needs

Database-level (FILE_COPY):

- Zero sessions on source
- Any in-flight transaction can corrupt the clone via shared `pg_xact`
- A truly consistent clone would require freezing the whole cluster
- No escape within a single database

Cluster-level:

- Snapshot the whole data directory (`pg_xact` + WAL included)
- Start a new Postgres on that snapshot
- Consistency via crash recovery; works on a live source

Make the snapshot instant → instant branching.

Part 2: Cloning, inside the cluster

You're already doing this

Every Postgres installation ships with `template1`. Every `CREATE DATABASE` copies it.

```
CREATE DATABASE myapp;
-- is the same as:
CREATE DATABASE myapp TEMPLATE template1;
```

You've been cloning databases since day one. You just haven't been using it deliberately.

One config line. 316× faster.

```
# postgresql.conf
file_copy_method = clone
```

```
CREATE DATABASE test_run TEMPLATE app_template STRATEGY=FILE_COPY;
```

METHOD	6 GB DATABASE
WAL_LOG (default)	67 seconds
FILE_COPY + clone	**212 milliseconds**

The ZFS trap

```
$ cat /sys/module/zfs/parameters/zfs_bclone_enabled
0   ← Ubuntu 24.04 (ZFS 2.2.x): block cloning is OFF
1   ← Ubuntu 26.04 (ZFS 2.4.1): finally on by default
```

ZFS_BCLONE_ENABLED	TIME	POOL SPACE
0 (default on ZFS 2.2.x)	3.33 s	+468 MiB
1	**0.56 s**	**0**

Enable it:

```
echo 1 | sudo tee /sys/module/zfs/parameters/zfs_bclone_enabled
echo "options zfs zfs_bclone_enabled=1" | sudo tee /etc/modprobe.d/zfs.conf
```

The template pattern

```
CREATE DATABASE app_template;
-- load nightly: pg_dump | pg_restore + run masking SQL here
ALTER DATABASE app_template IS_TEMPLATE true;
ALTER DATABASE app_template ALLOW_CONNECTIONS false;
```

`ALLOW_CONNECTIONS false` closes the door. It doesn't clear the room. Sessions already attached stay attached. **Masking goes in the refresh job. This data lands on every CI runner.**

Per-test isolation: the fixture

```python
@pytest.fixture(scope="function")
def db():
    name = f"test_{uuid4().hex[:8]}"
    with psycopg.connect(ADMIN_DSN, autocommit=True) as admin:
        admin.execute(
            f'CREATE DATABASE "{name}" TEMPLATE app_template STRATEGY=FILE_COPY'
        )
        try:
            with psycopg.connect(f"postgresql://localhost/{name}") as conn:
                yield conn
        finally:
            admin.execute(f'DROP DATABASE IF EXISTS "{name}" WITH (FORCE)')
```

500 tests × 212ms ≈ **2 minutes** overhead for complete isolation with real data.

Part 3: Branching, snapshot the whole cluster

Snapshot. Let crash recovery fix it.

The branch wakes up as if after a power loss. Crash recovery brings it to a consistent state.

```
zfs snapshot tank/pgdata@before-migration-42
zfs clone tank/pgdata@before-migration-42 tank/pgdata-branch
rm -f /tank/pgdata-branch/postmaster.pid
sudo -u postgres pg_ctl start -D /tank/pgdata-branch \
  -o "-p 5433 -c archive_mode=off" -w
```

```
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  redo starts at 0/9D000028
LOG:  redo done at 0/9D000080
LOG:  database system is ready to accept connections
```

Any PostgreSQL version. Any running database. **Since 2005.** Branch of a 1.94 GB cluster: **414 KiB**. After deleting 2M rows: 892 MB. You pay for divergence.

`archive_mode=off` is not optional

The branch inherits `archive_command` **and** the source's timeline:

```
next WAL file, source: 0000000100000000000009D
next WAL file, branch: 0000000100000000000009D
```

Same name, same archive destination. The branch **silently overwrites production's WAL archive**, and your PITR window with it.

```sql
-- branch-sanitize.sql: run before anything else
ALTER SYSTEM SET archive_mode = off;
ALTER SYSTEM SET synchronous_standby_names = '';
SELECT pg_reload_conf();
```

A branch believes it is production

INHERITED	WHAT THE BRANCH DOES
`archive_command`	overwrites production's WAL archive
logical replication subscriptions	advances the **production** slot: silent data loss
`synchronous_standby_names`	first `COMMIT` hangs forever
`pg_cron` jobs	runs your nightly billing job. **Twice.**
`postgres_fdw` / `dblink`	still points at production: writes escape

Sanitize on first boot. Don't rely on remembering five flags.

No ZFS? Reflink a paused replica.

1. **Pause** replay on the replica (seconds, production never knows)
2. **Reflink** the data directory (`cp --reflink=always`)
3. **Resume** replay, then promote the branch (new timeline, safe WAL names)

```
psql -h replica -c "SELECT pg_wal_replay_pause();"
psql -h replica -c "CHECKPOINT;"

cp -a --reflink=always /mnt/xfs/replica /mnt/xfs/branch

psql -h replica -c "SELECT pg_wal_replay_resume();"

rm -f /mnt/xfs/branch/postmaster.pid
pg_ctl start -D /mnt/xfs/branch -o "-p 5436 -c archive_mode=off" -w
pg_ctl promote -D /mnt/xfs/branch -w
```

Measured: **1 MB** extra disk to branch a 1,025 MB data directory.

CI: set up ZFS once

```
- name: Install ZFS (file-backed, no spare device needed)
  run: |
    sudo apt-get install -y zfsutils-linux
    sudo truncate -s 20G /zfs.img
    sudo zpool create -f tank /zfs.img
    sudo zfs set recordsize=8K tank
    echo 1 | sudo tee /sys/module/zfs/parameters/zfs_bclone_enabled
```

File-backed pool, 2-vCPU runner, no spare device needed.

CI: branch per PR

```
- name: Branch per PR
  run: |
    BRANCH=pgdata-${{ github.run_id }}
    sudo zfs clone tank/pgdata@nightly tank/$BRANCH
    sudo rm -f /tank/$BRANCH/postmaster.pid
    docker run -d --name $BRANCH -p 5433:5432 \
      -v /tank/$BRANCH:/var/lib/postgresql/18/docker \
      postgres:18 -c archive_mode=off
```

Every PR gets its own Postgres. Branch is instant. (ZFS calls it a clone; Postgres-wise, it's a branch.) `archive_mode=off` is mandatory.

Docker branches the binaries. ZFS branches the data. Teardown: `docker rm -f` + `zfs destroy`.

Two levels. Pick one.

	DATABASE-LEVEL (CLONE)	CLUSTER-LEVEL (BRANCH)
How	`CREATE DATABASE … TEMPLATE … FILE_COPY`	snapshot PGDATA, start new Postgres
Scope	one database inside the cluster	entire cluster (incl. `pg_xact`, WAL)
Session constraint	zero sessions on source	none: snapshot is below Postgres
Live source?	✗	✓
Cost	~200ms, same instance	new Postgres per branch
Examples	pg18 template pattern, your CI fixture	ZFS snapshot, reflink replica, **all three vendors**

Database-level is cheaper. Cluster-level is the only option when the source is live.

Part 4: What the market built

The snapshot is the easy part

You've seen it: three commands. Everything around them is the actual product.

- **Anonymization** on the way in: production data masked before it reaches any branch
- **The pipeline**: keeping the baseline fresh, nightly refresh, shipping it to runners
- **Lifecycle**: a branch per PR, a connection string for CI, cleanup when the PR closes
- **Safety**: sanitize on boot, every time (`archive_mode`, subscriptions, cron)
- **A GitHub app**, so nobody has to maintain `branch-create.sh`

Every line here, you've now seen as a script. **That's the DIY price.**

Three companies. Three layers.

	DBLAB (POSTGRES.AI)	XATA	NEON
Layer	block device (ZFS / LVM)	block device (NVMe-oF CoW)	storage engine replacement
Postgres	unmodified	unmodified	patched
Branch speed	instant	instant	instant
Branch past LSN	snapshots only	✕	✓ up to 30 days
Deployment	self-hosted	managed cloud	managed cloud

None of them found a way around the zero-sessions rule. All three stepped **below** the database, where the rule doesn’t apply.

How Neon solved historical branching

```
Postgres (patched)
  ↓ GetPage@LSN            ← pages fetched by LSN, not from disk
safekeepers (Paxos quorum) ← WAL durable once majority flushed
  ↓
pageserver
  per-timeline LSM: delta layers + image layers → S3
  pg_xact + pg_multixact versioned alongside relation pages
branch = timeline(ancestor_id, ancestor_lsn) // no copy, no replay
```

A branch at LSN X gets the **commit log as of X**. Historical branching is consistent and instant.

Same compute-storage split as AlloyDB, but with LSN-addressable storage. That's what makes branching possible.

Where DIY stops

	PG18 REFLINKS	ZFS SNAPSHOT	WAL ARCHIVE	XATA / DBLAB	NEON
Copy current state	✓ 200ms	✓ instant	~minutes	✓ instant	✓ instant
Copy a live source	✗	✓	✓	✓	✓
Branch at past LSN	✗	snapshots only	~minutes	✗	✓ instant
Vanilla PostgreSQL	✓	✓	✓	✓	✗
No filesystem access	✗	✗	~PITR	✓	✓
Per-PR automation	DIY	DIY	DIY	built-in	built-in

Pick your layer

- **CI test isolation, you own the box**: clone (database-level). `file_copy_method = clone`, frozen template, nightly masked refresh.
- **Branch something in use**: branch (cluster-level). ZFS snapshot. `postmaster.pid` and `archive_mode=off`.
- **Branch the past**: WAL archive + `recovery.signal`. Minutes OK: DIY. Milliseconds: Neon.
- **On RDS / Cloud SQL**: no filesystem. Neon or Xata.
- **On Kubernetes**: CSI `VolumeSnapshot` (OpenEBS ZFS, LVM). Branch = PVC from snapshot.
- **Self-hosted, want the platform**: DbLab. Open source, REST API, ~3 days.

The shared staging database survived this long

...because branching was expensive.

Your filesystem made it cheap in **2005**.

Your CI runner has ZFS available **today**.

PostgreSQL 18 made it a config option.

The only thing that changed tonight is that you know **which layer you're standing on.**

Pick your layer.

github.com/boringsql/pg-branch-recipes