Version control for transactional data

A git-like protocol for PostgreSQL: the why, the what and the how; the open-source projects already building it, where their design falls short, and a better way built on what Postgres already has.

Abstract

Software teams version their code with git and would not dream of working without it, yet the database underneath that code, the most valuable and least reversible state an organisation holds, is still edited in place with no branch, no diff and no merge. A lively field of open-source projects promises to close that gap and bring git to the database, and this paper sets out the why, the what and the how. We argue that the real prize is not branching but merging, the ability to reconcile divergent lines of transactional data, and we show that the field has split into two camps that each surrender half the problem: Dolt and DoltgreSQL win true git semantics by reimplementing the database engine and leaving native Postgres behind, while Neon and Supabase keep Postgres but offer a branch that can fork and never rejoin. We propose a better way. Not a forked engine and not a forked storage layer, but a version-control protocol built on the write-ahead log, multiversion concurrency and logical decoding that Postgres already ships, with semantic, constraint-aware merge as its first-class operation. Version control for data should separate the version model from the storage, the way git separated the object model from the transport.

Keywords: PostgreSQL; database version control; git for data; data branching; schema migration; three-way merge; conflict resolution; write-ahead log; logical decoding; multiversion concurrency control; MVCC; copy-on-write; CRDT; Dolt; DoltgreSQL; Neon; Supabase; temporal tables; data provenance; digital sovereignty

1. Introduction

Ask a developer to work for a day without git and you will get a flat refusal. Branching, committing, diffing and merging are so woven into how software is built that their absence is unthinkable. Now ask the same developer how they version the production database, the schema and the data that the code exists to serve, and the answer is almost always a sheepish nothing. The database is edited in place. A migration runs forward and, if it goes wrong, a frantic restore from last night's backup is the only way back. Two engineers cannot develop against divergent copies of the data and reconcile them later; they take turns, or they clobber each other. The most valuable and least reversible state an organisation holds is the one piece of the stack with no version control at all.

This is a strange gap, and a growing field of open-source projects has set out to close it under a banner everyone understands: git for your database. The pitch is immediately appealing. Branch the database to try a risky change in isolation. Diff two versions to see exactly what a migration did. Give every analyst their own line of data and merge the good work back. Travel to any past state as easily as `git checkout`. If version control transformed how we write code, the argument runs, it can transform how we manage data.

We think the argument is right, and this is a position paper about how to make it true. We explain why version-controlled data matters (Section 2), what a git-like protocol actually has to provide, which is more and harder than it first looks (Section 3), and where today's projects stand, with credit where it is due and a clear-eyed account of where their designs fall short (Section 4). Then we make our case: that the field has backed itself into a false choice, and that the better path for Postgres is a protocol built on the primitives it already ships rather than a new engine or an unmergeable fork (Section 5). Nashua endorses best-of-breed tools and sound practice over any single product, and that is the spirit here: not a pitch for one project, but an argument about how this should be built.

2. Why version-control the database

The case begins with the work people already do badly by hand. Schema migrations are the obvious one. Every team that runs a database runs migrations, and every team has felt the particular dread of a migration that cannot be cleanly undone. A version-controlled database makes a migration a commit: reversible, inspectable, and testable on a branch before it touches production. The diff is not a guess from the migration script; it is the actual change to schema and data, computed and reviewable like any pull request.

Then there is the development and test loop. Modern teams want a realistic database for every feature branch and every pull-request preview, seeded with production-like data, spun up in seconds and thrown away when the branch merges. Doing that by restoring full copies is slow and expensive; doing it by sharing one database invites collisions. Cheap, instant branches solve a real and daily problem, which is exactly why the branching products have found an eager market.

Beyond the daily loop lie the deeper reasons. Reproducibility: a data scientist who can pin an analysis to an exact committed state of the data can reproduce a result months later, and so can the auditor who questions it. Provenance and audit: a database whose history is a chain of signed commits answers who changed what, and when, by construction rather than by a bolted-on trigger. This is not a nicety in regulated sectors; the European Union's DORA regime and its kin expect financial entities to demonstrate the integrity and traceability of their data, and a versioned history is demonstrable in a way that a mutable table is

not. Recovery: point-in-time restore stops being a backup operation and becomes a checkout. And collaboration at last becomes possible on data the way it is on code, with independent lines of work that converge through review rather than through locking.

None of this is exotic. It is the ordinary machinery of software engineering, applied to the one asset that has stubbornly resisted it. The question is not whether the database deserves version control. It plainly does. The question is what it would take to give it the real thing.

3. What a git-like protocol means

It is worth being precise about what git actually provides, because the easy half is routinely mistaken for the whole. Git is a content-addressed object store: every version of every file is a blob named by the hash of its contents, trees compose blobs into a snapshot, and a commit names a tree together with its parents, forming an immutable directed acyclic graph of history. On top of that store sit two layers of convenience: refs, which are human names for points in the graph, so that `main` and `feature` are just labels on commits; and operations, of which `diff` and `merge` are the ones that matter.

The decisive operation, the one that makes git a collaboration tool rather than a fancy backup, is the three-way merge. When two branches diverge, git does not ask which one wins. It finds their common ancestor, computes what each side changed relative to that ancestor, and combines the changes, flagging only the places where both sides touched the same thing in incompatible ways. That is what lets a hundred people work in parallel and converge. Branching without merging is just a cheap copy; branching with merging is version control.

Carry this to a database and the mapping is suggestive but not trivial, as Figure 1 lays out. A commit should be an immutable snapshot of the whole database at a logical instant. A branch should be a named line of such snapshots. A diff should report the rows and schema objects that changed between two commits. And a merge should reconcile two divergent lines of data into one. The first three are, in essence, a storage and naming problem. The fourth is the hard one, and it is where the analogy to git stops being comfortable.

Text merges because a source file is a line-ordered sequence with no rules about its contents; git can splice two sets of line changes and the result is still a valid file by definition. A database is not like that. Its contents are governed by invariants: primary keys must stay unique, foreign keys must resolve, check constraints must hold, and, above all of those, the data must still mean something. Two branches that each insert a customer with the same natural key have not produced a merge conflict in the textual sense; they have produced a state that the schema forbids. Two branches that each decrement an account balance by a different amount cannot both simply win. A merge for data has to respect the rules of the data, and sometimes the intent behind it. That is the real content of the phrase git for the database, and it is the test against which the existing projects should be measured.

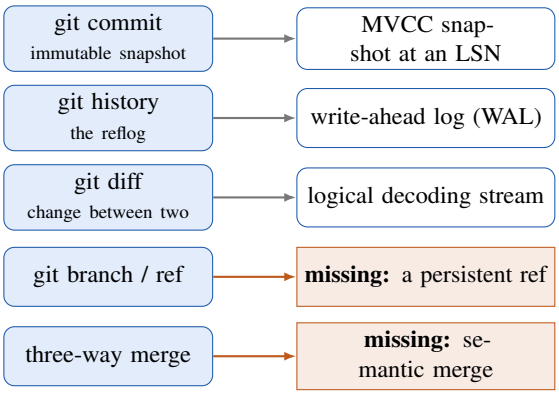


Figure 1. The git object model mapped onto Postgres. Three of git's four building blocks already exist inside Postgres as the MVCC snapshot, the write-ahead log and logical decoding; what is missing is a persistent branch ref and, above all, a merge that understands the data. A protocol has only to supply the two warm boxes, not to rebuild the three blue ones.

4. The state of the art, and where it falls short

The field divides cleanly into three groups, and the division is instructive because each group is defined by what it gives up. Table 1 sets them side by side.

4.1. Reimplementing the engine: Dolt and DoltgreSQL

Dolt is the most complete realisation of the idea, and it deserves real credit. It is an open-source SQL database built from the ground up to be versioned: its storage is a content-addressed Merkle directed acyclic graph, structured with probabilistic trees, often called prolly trees, that give it git's structural sharing and fast diffs while remaining an indexed, queryable database. Branch, diff, merge, clone, push and pull are first-class SQL and command-line operations. Dolt is, genuinely, git for data, and DoltgreSQL extends the same engine to speak the PostgreSQL wire protocol so that Postgres clients can connect.

The cost is in plain sight in that sentence: built from the ground up. Dolt is not Postgres. It is a new engine that reimplements a database and, in DoltgreSQL's case, the Postgres protocol on top of it. An organisation adopting it does not add versioning to the Postgres it runs; it replaces Postgres with something else and inherits that something's performance characteristics and the long tail of Postgres extensions it does not reproduce, PostGIS and pgvector and the rest that production systems lean on. DoltgreSQL reached a 1.0 release in 2026 and is impressively compatible, but compatible with the Postgres dialect is not the same as being Postgres. For a greenfield project that values versioning above all, that can be a fair trade. For the vast installed base of production Postgres, replacing the engine to gain version control is a very large tail wagging a very important dog.

There is a subtler limitation, and it matters for the argument to come. Dolt's merge is modelled on git's: it works a cell at a time, taking each side's changes relative to the common ancestor and combining them, and where both sides changed the same cell it records a conflict for a human to resolve. This is structural merge, and it is the right default, but it is semantically blind. It does not know that two concurrent increments to a stock level should sum rather than collide, or that two inserts sharing a natural key are not a

Table 1. Where today's approaches stand. Native PG means the real PostgreSQL engine, with its extensions and ecosystem, rather than a reimplementation of its wire protocol. The pattern is stark: the only project with true data merge is the one that leaves Postgres, and every tool that keeps Postgres gives merge up.

Project	Approach	Native PG	Branch	Data diff	Data merge	Conflict model
Dolt	new engine, Merkle DAG	no	yes	yes	yes	cell-wise, structural
DoltgreSQL	new engine, PG wire	no	yes	yes	yes	cell-wise, structural
Neon	copy-on-write storage	yes	yes	no	no	none
Supabase branching	per-branch instance	yes	yes	schema only	no	none
Temporal / audit tools	linear history	yes	no	over time	no	none
A protocol (proposed)	*over WAL + MVCC*	*yes*	*yes*	*yes*	*yes*	*semantic, constraint-aware*

coincidence but a contradiction. Dolt does detect constraint violations that a merge introduces and surfaces them for resolution rather than silently committing a broken state, which is to its credit, but detection after the fact is not the same as a merge that understands the data's meaning. The structural model is the ceiling of what cell-wise merge can do.

4.2. Branching the storage: Neon and Supabase

The second group keeps Postgres, exactly the thing Dolt gives up, and buys its branching lower in the stack. Neon re-architects Postgres storage, separating compute from a multiversion, log-structured storage service, and branching falls out of that design almost for free: a branch is a copy-on-write fork of the storage at a chosen point in the log, created in seconds no matter how large the database, because nothing is copied until it is written. Supabase offers branching in a similar spirit, standing up a separate Postgres instance per branch and wiring it into a Git-driven preview workflow, and the same copy-on-write pattern runs right through the ecosystem, from Xata's data-carrying branches to the ZFS thin clones of Postgres.ai. All are genuinely useful, and for the development and test loop of Section 2 they are close to ideal.

But look at the table again. These are branches that can fork and never rejoin. Neon can create a branch at an instant and diverge from it; it cannot merge that branch's data changes back into its parent, because nothing in a copy-on-write storage fork knows how to reconcile two sets of divergent page writes into one coherent, constraint-respecting state. That is not an oversight to be patched; it is the consequence of solving the problem at the storage layer, where the unit is an opaque page, rather than at the logical layer, where the unit is a row with meaning. Supabase's branches are oriented at schema migrations for preview environments, not at merging data at all. The result is powerful copy-on-write cloning, which is a real and valuable thing, wearing the vocabulary of version control while delivering only its easy half. Calling a fork without a merge a branch borrows a promise the design cannot keep.

4.3. Logging history: temporal tables and audit trails

A third group is sometimes swept into the same conversation and should be distinguished. System-versioned temporal tables, as standardised in SQL:2011 and approximated in Postgres by extensions such as temporal_tables, and audit frameworks such as pgMemento, give a table a history: every row carries the span of time for which it was current, and you can query the database as it stood at any past instant. This is valuable, and it supplies the provenance and point-in-time reading that Section 2 asked for. But it is a single, linear history. There is no branch and no merge, no parallel lines of work to reconcile. Time-travel is not version control; it is one axis of it, the one that was always the easiest to add.

4.4. Three concerns, wrongly fused

Step back and the pattern is clear. Every project answers the same three questions at once, and bundles the answers into a single product: how history is modelled, how a version is physically stored, and how conflicts are resolved. Dolt fuses a git-like version model to a bespoke storage engine and a cell-wise conflict model, and to get the first you must take all three. Neon fuses a thin version model to a copy-on-write storage engine and no conflict model at all. Because the three are welded together, you cannot keep Postgres and still have merge, and you cannot have semantic merge from anyone, because the conflict model is whatever the storage engine happened to make easy. The false choice that the field presents, Postgres without merge or merge without Postgres, is an artefact of fusing concerns that are logically separate. Pull them apart and a better design appears.

5. A better way: a protocol, not a fork

Our proposal is to stop building version-controlled databases and start defining a version-control protocol for the database we already have. The move is the same one git itself made: git's genius was to separate the object model, commits and trees and refs, from the transport that moves them around, so that the model could outlive any particular implementation. Database version control should separate the version model, commits and branches and merges, from the physical storage that materialises them, so that it can run on stock Postgres instead of requiring a new engine. Figure 2 shows the shape of it.

5.1. The substrate Postgres already ships

The striking thing, and the heart of our argument, is how much of the machinery Postgres already has. Its multiversion concurrency control keeps multiple versions of every row and hands each transaction a consistent snapshot of the database at a logical instant: that is a commit in all but name. Its write-ahead log is an ordered, durable record of every change ever applied, which is to say a reflog. And logical decoding, the mature mechanism that already powers logical replication, turns that log into a clean, ordered stream

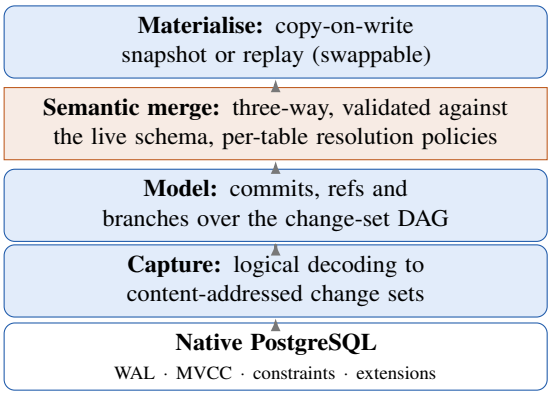


Figure 2. A version-control protocol over native Postgres. The engine at the base is unchanged, stock Postgres with its constraints and extensions intact. Above it, logical decoding captures changes as content-addressed commits, a thin model layer supplies branches and refs, and semantic merge, the one genuinely hard part, reconciles divergence by validating against the live schema and applying per-table policies. How a branch is physically materialised is a swappable optimisation, not part of the contract.

of row-level changes that any client can consume. Three of git's four primitives, immutable snapshots, a history log and a change stream, are sitting inside Postgres already, as Figure 1 showed. A protocol does not need to rebuild them. It needs only to add the two things Postgres lacks: persistent named refs over that history, and a merge that understands the data.

Capturing commits, then, is a matter of consuming the logical-decoding stream and recording each transaction's changes as a content-addressed change set, hashed by its contents so that identical changes share storage and any change set can be named and fetched by its hash. A slot is created once, and the changes are drained and stored:

```
SELECT pg_create_logical_replication_slot(
  'vc', 'pgoutput');

-- each transaction's changes, in order,
-- hashed and stored as a commit object
SELECT * FROM pg_logical_slot_get_changes(
  'vc', NULL, NULL);
```

Nothing here forks the engine or reaches beneath the row. The protocol reads what Postgres already emits and builds the version graph beside it.

5.2. Commits, refs and branches

With change sets as commits, the rest of the easy half is bookkeeping. A commit records its parent or parents and the content hash of the change set it introduces, forming the same immutable directed acyclic graph git uses. A ref is a name pointing at a commit, and a branch is a ref that moves as new commits are added to it. This metadata is small, it is ordinary relational data, and it can live in a schema inside the very database it versions or in a companion store. Because a commit is defined by content-addressed logical changes rather than by physical pages, it is portable: it means the same thing on any Postgres, on-premises or in any cloud, which is precisely the property a copy-on-write storage fork cannot offer. The version model is pure protocol, and it carries no lock-in.

5.3. Semantic, constraint-aware merge

Here is the hard half, and the place where a protocol can do what no current tool does. Because commits are logical change sets rather than opaque pages, a three-way merge can operate where meaning lives. Given two branches and their common ancestor, the protocol computes each side's row-level changes and combines them, and then does the two things that distinguish a database merge from a text merge. First, it validates the proposed result against the live schema, every constraint, key and foreign-key relationship, so that a merge can never commit a state the database would reject; a violation is raised as a conflict, not written as corruption. Second, and this is the real advance, it resolves overlaps using policies declared per table and per column by the people who own the data's meaning, rather than a single blind cell-wise rule.

Those policies are where domain intent enters. A counter column can be declared to merge by summing the deltas each side applied, the behaviour of a convergent replicated data type, so that two concurrent increments correctly sum instead of colliding. A last-updated timestamp can merge by taking the later value. A set-valued relationship can merge by union. A natural-key collision can be declared an unconditional conflict, because the schema's meaning says the two rows are the same entity. Where no policy applies and both sides touched the same value, the protocol does exactly what git does and stops for a human, presenting the conflict in terms of rows and values rather than pages. A declaration might read as plainly as this:

```
MERGE POLICY ON inventory (on_hand)
  USING sum_of_deltas;
MERGE POLICY ON inventory (sku)
  USING conflict;  -- same SKU, same item
```

This is the piece that prior art shows is possible. Content-addressed systems with user-defined merge functions already exist outside the relational world: Irmin, the git-like store behind parts of the MirageOS project, is built around exactly this idea of mergeable data with custom merge logic, and the literature on convergent and commutative replicated data types gives a rigorous account of when concurrent changes can be merged without conflict. The theory of invariant confluence, from Bailis and colleagues, goes one step further and tells us precisely which invariants a merge can preserve without coordination and which, global uniqueness and foreign keys among them, provably cannot. That is not a counsel of despair; it is the reason a natural-key collision has to be declared a conflict rather than wished away, and it tells a protocol exactly where an automatic policy is safe and where a human must decide. What has not been done is to bring that semantic merge to native, transactional Postgres, governed by the schema's own constraints. That is the open problem worth solving, and the one the branching products have stepped around rather than through.

5.4. Materialisation as an optimisation

A fair objection is that replaying logical changes to reconstruct a branch could be slow for a large database. The answer is the decoupling itself: how a branch is physically materialised is not part of the version model, so it can be

chosen freely and changed later. A branch can be replayed from change sets when it is small or rarely used, or stood up as a copy-on-write snapshot when it must be fast, using filesystem snapshots such as ZFS or the very copy-on-write storage that Neon and its kind have built. In other words, the excellent fork-fast engines are not competitors to this protocol; they are one possible backend for its materialisation, with the merge and the portable version model supplied above them. The protocol says what a branch and a merge mean; the storage layer is free to make them fast.

5.5. Honest limits

We are not claiming this is easy, and a position paper that pretended otherwise would deserve suspicion. Semantic merge is genuinely hard: writing correct per-table policies takes care, long-diverged branches accumulate conflicts that no policy can resolve without judgement, and very high-volume change streams raise real questions of storage and throughput for the commit log. Validating a large merge against every constraint has a cost, and some invariants, as invariant-confluence theory proves, cannot be reconciled without coordination at all, so an honest protocol surfaces those as conflicts rather than pretending to resolve them. These are the reasons the problem is still open, not reasons to declare it unsolvable. What the protocol approach buys is that the hard part, merge, is tackled in the one place it can be tackled well, at the logical layer with the schema in hand, instead of being abandoned at the storage layer or flattened into a cell-wise rule. It also means the work compounds: because it is a protocol over stock Postgres, every improvement in the engine, every extension, every operational tool keeps working underneath it.

6. What this means in practice

For an organisation choosing today, the honest guidance follows the table. If what you need is cheap, instant databases for development, testing and pull-request previews, the copy-on-write branching tools are excellent and you should use them, with clear eyes that a branch there is a disposable fork, not a line of work you will merge back. If you are starting fresh and versioning is the point of the system, Dolt is a serious and capable tool, provided you accept that you are choosing a new engine rather than extending Postgres. And if what you want is the full promise, branch and diff and merge over the production Postgres you already run and trust, the honest answer is that no product delivers it yet, and the right thing to bet on is a protocol built on Postgres's own primitives rather than another fork of the engine or the storage.

That last position is the one Nashua holds, and it is of a piece with how we read this whole landscape. We favour best-of-breed tools and open standards over single-vendor platforms, and we are wary of lock-in wherever it hides, including inside a storage layer that can branch but never let your data leave on equal terms. A version model that is pure protocol, portable across any Postgres on any infrastructure, on-premises or cloud, keeps the choice of where and how to run firmly with the organisation. Cloud and on-premises are both good homes for data; a protocol that does not care which is exactly the point.

7. Conclusion

The database deserves the version control that transformed every other part of software engineering, and the projects chasing that goal are right to chase it. But the prize is merge, not just branch, and the field has split into tools that merge by leaving Postgres and tools that keep Postgres by giving up merge. That split is not inevitable. It comes from fusing three separable concerns, the version model, the physical storage and the conflict resolution, into one product. Separate them, build the version model as a protocol on the write-ahead log, multiversion storage and logical decoding that Postgres already ships, and put a semantic, constraint-aware merge at its centre, and the false choice dissolves. Git earned its place by separating the model from the machine. Data deserves the same, and Postgres, almost uniquely, already has the parts to make it real. Nashua would rather help build that than sell a fork of it.

References

[1] DoltHub (2026). *Dolt: Git for Data*. https://github.com/dolthub/dolt

[2] DoltHub (2026). *Prolly Trees: the data structure behind Dolt*. https://docs.dolthub.com/architecture/storage-engine/prolly-tree

[3] DoltHub (2026). *DoltgreSQL: a Postgres-flavoured Dolt*. https://github.com/dolthub/doltgresql

[4] DoltHub (2026). *Merge and conflict resolution in Dolt*. https://docs.dolthub.com/sql-reference/version-control/merges

[5] Neon (2026). *Branching in Neon: copy-on-write Postgres*. https://neon.tech/docs/introduction/branching

[6] Supabase (2026). *Branching documentation*. https://supabase.com/docs/guides/deployment/branching

[7] The PostgreSQL Global Development Group (2026). *PostgreSQL documentation: Logical Decoding*. https://www.postgresql.org/docs/current/logicaldecoding.html

[8] The PostgreSQL Global Development Group (2026). *PostgreSQL documentation: Concurrency Control (MVCC)*. https://www.postgresql.org/docs/current/mvcc.html

[9] The PostgreSQL Global Development Group (2026). *PostgreSQL documentation: Write-Ahead Logging (WAL)*. https://www.postgresql.org/docs/current/wal-intro.html

[10] Al-Kateb, M. et al.; ISO/IEC 9075:2011 (SQL:2011) *system-versioned tables*; and *temporal_tables* for PostgreSQL. https://github.com/arkhipov/temporal_tables

[11] Bitterlich, F. (2026). *pgMemento: audit trail and versioning for PostgreSQL*. https://github.com/pgMemento/pgMemento

[12] Chacon, S. and Straub, B. (2014). *Pro Git*, 2nd edition, Apress. https://git-scm.com/book

[13] Shapiro, M., Preguiça, N., Baquero, C. and Zawirski, M. (2011). *Conflict-free Replicated Data Types*. SSS 2011. https://hal.inria.fr/inria-00609399

[14] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M. and Stoica, I. (2015). *Coordination Avoidance in Database Systems*, which introduces invariant confluence. VLDB. https://arxiv.org/abs/1402.2237

[15] Xata and ClickHouse (2026). *Managed Postgres with instant branching*. https://clickhouse.com/blog/managed-postgres-ga-xata-acquisition

[16] Postgres.ai (2026). *Database Lab Engine: instant thin clones of PostgreSQL*. https://postgres.ai/

[17] Gazagnaire, T. et al. (2026). *Irmin: a distributed, git-like, mergeable store*. https://irmin.org/

[18] TerminusDB (2026). *TerminusDB: a git-like graph database*. https://terminusdb.com/

[19] PlanetScale (2026). *Branching and deploy requests*. https://planetscale.com/docs/concepts/branching