


Open main menu 

Articles
[Blog](#) [Phoenix](#) [Files](#) [Laravel](#) [Bytes](#) [Ruby](#) [Dispatch](#) [Django](#) [Beats](#) [JavaScript](#) [Journal](#)

[Security](#) [Infra](#) [Log](#) [Docs](#) [Community](#) [Status](#) [Pricing](#)
[Sign In](#) [Get Started](#)  [RSS Feed](#)
[Blog](#) [Phoenix](#) [Files](#) [Laravel](#) [Bytes](#) [Ruby](#) [Dispatch](#) [Django](#) [Beats](#) [JavaScript](#) [Journal](#) [Security](#) [Infra](#) [Log](#) [Docs](#) [Community](#) [\(opens an external site\)](#)  [Status](#) [\(opens an external site\)](#)  [Pricing](#)
[Sign In](#) [Get Started](#)  [RSS Feed](#)

Reading time • 8 min  [Share this post on Twitter](#)  [Share this post on Hacker News](#)  [Share this post on Reddit](#)

Litestream v0.5.0 is Here



Name
Ben Johnson
@benbjohnson
[@benbjohnson](#)

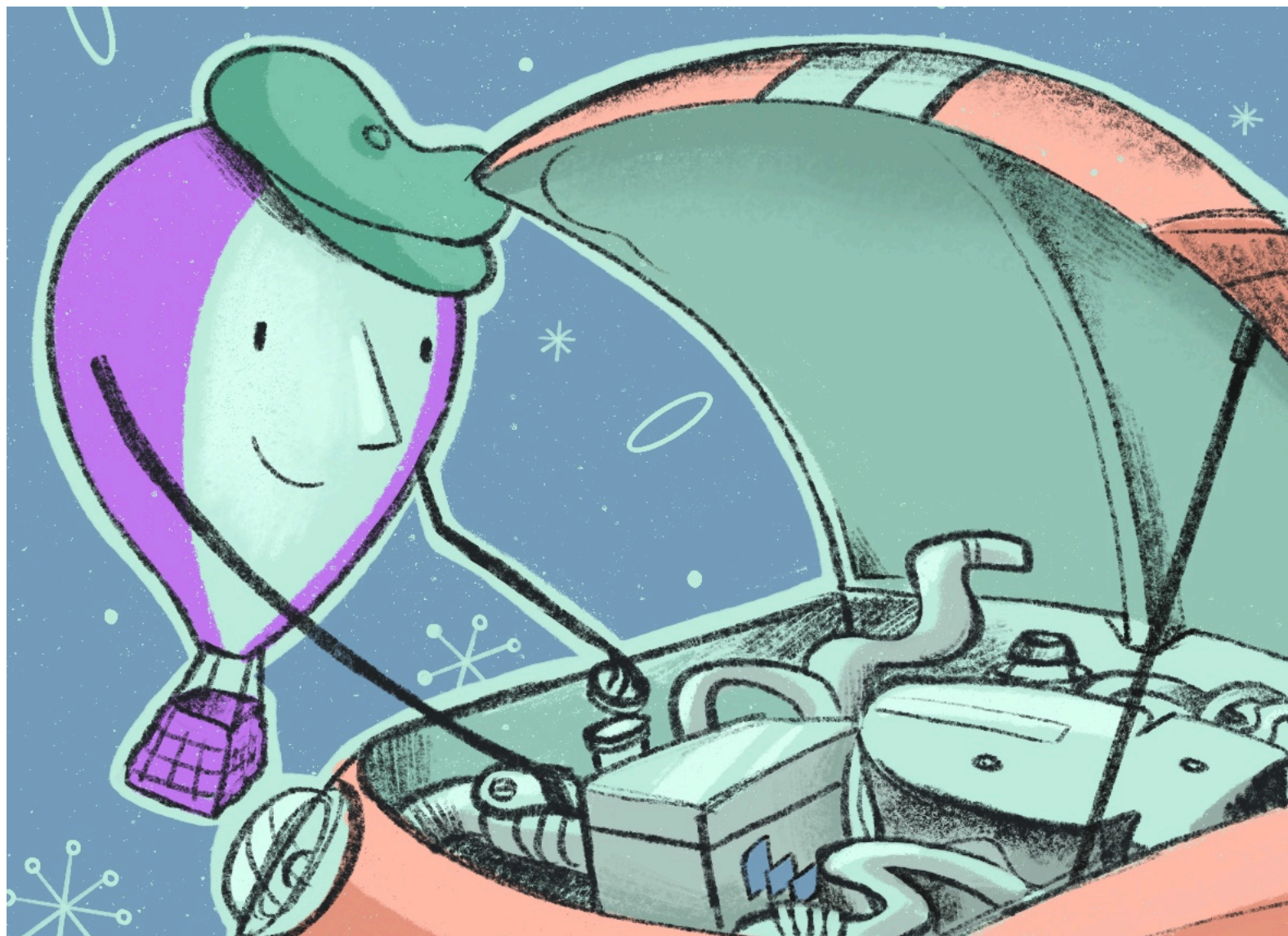


Image by  [Annie Ruygt](#)

I'm Ben Johnson, and I work on Litestream at Fly.io. Litestream makes it easy to build SQLite-backed full-stack applications with resilience to server failure. It's open source, runs anywhere, and [it's easy to get started](#).

Litestream is the missing backup/restore system for SQLite. It runs as a sidecar process in the background, alongside unmodified SQLite applications, intercepting WAL checkpoints and streaming them to object storage in real time. Your application doesn't even know it's there. But if your server crashes, Litestream lets you quickly restore the database to your new hardware.

The result: you can safely build whole full-stack applications on top of SQLite.

A few months back, we announced [plans for a major update to Litestream](#). I'm psyched to announce that the first batch of those changes are now "shipping". Litestream is faster and now supports efficient point-in-time recovery (PITR).

I'm going to take a beat to recap Litestream and how we got here, then talk about how these changes work and what you can expect to see with them.

Litestream to LiteFS to Litestream

Litestream is one of two big SQLite things I've built. The other one, originally intended as a sort of sequel to Litestream, is LiteFS.

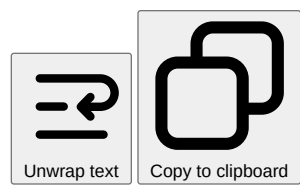
Boiled down to a sentence: LiteFS uses a FUSE filesystem to crawl further up into SQLite's innards, using that access to perform live replication, for unmodified SQLite-backed apps.

the big deal about LiteFS for us is that it lets you do the multiregion primary/read-replica deployment people love Postgres for: reads are fast everywhere, and writes are sane and predictable. We were excited to make this possible for SQLite, too.

But the market has spoken! Users prefer Litestream. And honestly, we get it: Litestream is easier to run and to reason about. So we’ve shifted our focus back to it. First order of business: [take what we learned building LiteFS and stick as much of it as we can back into Litestream](#).

The LTX File Format

Consider this basic SQL table:



```
CREATE TABLE sandwiches (  
  id INTEGER PRIMARY KEY AUTOINCREMENT,  
  description TEXT NOT NULL,  
  star_rating INTEGER,  
  reviewer_id INTEGER NOT NULL  
);
```

In our hypothetical, this table backs a wildly popular sandwich-reviewing app that we keep trying to get someone to write. People eat a lot of sandwiches and this table gets a lot of writes. Because it makes my point even better and it’s funny, assume people dither a lot about their sandwich review for the first couple minutes after they leave it. This Quiznos sub... is it ★ or ★★?

Underneath SQLite is a B-tree. Like databases everywhere, SQLite divides storage up into disk-aligned pages, working hard to read as few pages as possible for any task while treating work done within a page as more or less free. SQLite always reads and writes in page-sized chunks.

Our sandwiches table includes a feature that’s really painful for a tool like Litestream that thinks in pages: an automatically updating primary key. That key dictates that every insert into the table hits the rightmost leaf page in the underlying table B-tree. For SQLite itself, that’s no problem. But Litestream has less information to go on: it sees only a feed of whole pages it needs to archive.

Worse still, when it comes time to restore the database – something you tend to want to happen quickly – you have to individually apply those small changes, as whole pages. Your app is down, PagerDuty is freaking out, and you’re sitting there watching Litestream reconstruct your Quiznos uncertainty a page (and an S3 fetch) at a time.

So, LTX. Let me explain. We needed LiteFS to be transaction-aware. It relies on finer-grained information than just raw dirty pages (that’s why it needs the FUSE filesystem). To ship transactions, rather than pages, we invented a [file format we call LTX](#).

LTX was designed as an interchange format for transactions, but for our purposes in Litestream, all we care about is that LTX files represent ordered ranges of pages, and that it supports compaction.

Compaction is straightforward. You’ve stored a bunch of LTX files that collect numbered pages. Now you want to restore a coherent picture of the database. Just replay them newest to oldest, skipping duplicate pages (newer wins), until all changed pages are accounted for.

Importantly, LTX isn’t limited to whole database backups. We can use LTX compaction to compress a bunch of LTX files into a single file with no duplicated pages. And Litestream now uses this capability to create a hierarchy of compactions:

- at Level 1, we compact all the changes in a 30-second time window
- at Level 2, all the Level 1 files in a 5-minute window
- at Level 3, all the Level 2’s over an hour.

Net result: we can restore a SQLite database to any point in time, *using only a dozen or so files on average*.

Litestream performs this compaction itself. It doesn’t rely on SQLite to process the WAL file. Performance is limited only by I/O throughput.

No More Generations

What people like about Litestream is that it’s just an ordinary Unix program. But like any Unix program, Litestream can crash. It’s not supernatural, so when it’s not running, it’s not seeing database pages change. When it misses changes, it falls out of sync with the database.

Lucky for us, that’s easy to detect. When it notices a gap between the database and our running “shadow-WAL” backup, Litestream resynchronizes from scratch.

The only time this gets complicated is if you have multiple Litestreams backing up to the same destination. To keep multiple Litestreams from stepping on each other, Litestream divides backups into “generations”, creating a new one any time it resyncs. You can think of generations as Marvel Cinematic Universe parallel dimensions in which your database might be simultaneously living in.

Yeah, we didn’t like those movies much either.

LTX-backed Litestream does away with the concept entirely. Instead, when we detect a break in WAL file continuity, we re-snapshot with the next LTX file. Now we have a monotonically incrementing transaction ID. We can use it look up database state at any point in time, without searching across generations.

Upgrading to Litestream v0.5.0

Due to the file format changes, the new version of Litestream can’t restore from old v0.3.x WAL segment files.

That’s OK though! The upgrade process is simple: just start using the new version. It’ll leave your old WAL files intact, in case you ever need to revert to the older version. The new LTX files are stored cleanly in an ltx directory on your replica.

The configuration file is fully backwards compatible.

There’s one small catch. We added a new constraint. You only get a single replica destination per database. This probably won’t affect you, since it’s how most people use Litestream already. We’ve made it official.

The rationale: having a single source of truth simplifies development for us, and makes the tool easier to reason about. Multiple replicas can diverge and are sensitive to network availability. Conflict resolution is brain surgery.

Litestream commands still work the same. But you'll see references to "transaction IDs" (TXID) for LTX files, rather than the generation/index/offset we used previously with WAL segments.

We've also changed `litestream wal` to `litestream ltx`.

Other Stuff v0.5.0 Does Better

We've beefed up the [underlying LTX file format library](#). It used to be an LTX file was just a sorted list of pages, all compressed together. Now we compress per-page, and keep an index at the end of the LTX file to pluck individual pages out.

You're not seeing it yet, but we're excited about this change: we can operate page-granularly even dealing with large LTX files. This allows for more features. A good example: we can build features that query from any point in time, without downloading the whole database.

We've also gone back through old issues & PRs to improve quality-of-life. CGO is now gone. We've settled the age-old contest between `mattn/go-sqlite3` and `modernc.org/sqlite` in favor of `modernc.org`. This is super handy for people with automated build systems that want to run from a MacBook but deploy on an x64 server, since it lets the cross-compiler work.

We've also added a replica type for NATS JetStream. Users that already have JetStream running can get Litestream going without adding an object storage dependency.

And finally, we've upgraded all our clients (S3, Google Storage, & Azure Blob Storage) to their latest versions. We've also moved our code to support newer S3 APIs.

What's next?

The next major feature we're building out is a Litestream VFS for read replicas. This will let you instantly spin up a copy of the database and immediately read pages from S3 while the rest of the database is hydrating in the background.

We already have a proof of concept working and we're excited to show it off when it's ready!

Last updated

- Oct 2, 2025

[🐦 Share this post on Twitter](#) [📰 Share this post on Hacker News](#) [👤 Share this post on Reddit](#)

Author



Name

Ben Johnson
@benbjohnson
[@benbjohnson](#)

Previous post ↓

[Build Better Agents With MorphLLM](#)

Previous post ↓

[Build Better Agents With MorphLLM](#)



Company

[About Pricing Jobs](#)

Articles

[Blog](#) [Phoenix Files](#) [Laravel Bytes](#) [Ruby Dispatch](#) [Django Beats](#) [JavaScript Journal](#)

Resources

[Docs](#) [Support](#) [Support Metrics](#) [Status](#)

Contact

[GitHub](#) [Twitter](#) [Community](#)

Legal

[Security](#) [Privacy policy](#) [Terms of service](#) [Acceptable Use Policy](#)

Copyright © 2025 Fly.io