READING TIME • 14 MIN

I'm All-In on Server-Side SQLite

Annie Ruygt

I'm Ben Johnson. I wrote BoltDB, an embedded database that is the backend for systems like etcd. Now I work at Fly.io, on Litestream. Litestream is an open-source project that makes SQLite tenable for full-stack applications through the power of ✨replication✨. If you can set up a SQLite database, you can get Litestream working in less than 10 minutes.

The conventional wisdom of full-stack applications is the n-tier architecture, which is now so common that it's easy to forget it even has a name. It's what you're doing when you run an "application server" like Rails, Django, or Remix alongside a "database server" like Postgres. According to the conventional wisdom, SQLite has a place in this architecture: as a place to run unit tests.

The conventional wisdom could use some updating. I think that for many applications – production applications, with large numbers of users and high availability requirements – SQLite has a better place, in the center of the stack, as the core of your data and persistence layer.

It's a big claim. It may not hold for your application. But you should consider it, and I'm here to tell you why.

A Brief History of Application Databases

50 years is not a long time. In that time, we've seen a staggering amount of change in how our software manages data.

In the beginning of our story, back in the '70s, there were Codd's rules, defining what we now call "relational databases", also known today as "databases". You know them, even if you don't: all data lives in tables; tables have columns, and rows are addressable with keys; C.R.U.D.; schemas; a textual language to convey these concepts. The language, of course, is SQL, which prompted a Cambrian explosion of SQL databases, from Oracle to DB2 to Postgres to MySQL, throughout the '80s and '90s.

It hasn't all been good. The 2000s got us XML databases. But our industry atoned by building some great columnar databases during the same time. By the 2010s, we saw dozens of large-scale, open-source distributed database projects come to market. Now anyone can spin up a cluster and query terabytes of data.

As databases evolved, so too did the strategies we use to plug them in to our applications. Almost since Codd, we've divided those apps into tiers. First came the database tier. Later, with memcached and Redis, we got the caching tier. We've got background job tiers and we've got routing tiers

and distribution tiers. The tutorials pretend that there are 3 tiers, but we all know it's called "n-tier" because nobody can predict how many tiers we're going to end up with.

You know where we're going with this. Our scientists were so preoccupied with whether or not they could, and so on.

See, over these same five decades, we've also seen CPUs, memory, & disks become hundreds of times faster and cheaper. A term that practically defines database innovation in the 2010s is "big data". But hardware improvements have made that concept slippery in the 2020s. Managing a 1 GB database in 1996? A big deal. In 2022? Run it on your laptop, or a t3.micro.

When we think about new database architectures, we're hypnotized by scaling limits. If it can't handle petabytes, or at least terabytes, it's not in the conversation. But most applications will never see a terabyte of data, even if they're successful. We're using jackhammers to drive finish nails.

The Sweet Release of SQLite

There's a database that bucks a lot of these trends. It's one of the most popular SQL databases in the world, so standardized it's an official archival format of the Library of Congress, it's renowned for its reliability and its unfathomably encompassing test suite, and its performance is so good that citing its metrics on a message board invariably starts an argument about whether it should be disqualified. I probably don't have to name it for you, but, for the one person in the back with their hand raised, I'm talking about SQLite.

SQLite is an embedded database. It doesn't live in a conventional architectural tier; it's just a library, linked into your application server's process. It's the standard bearer of the "single process application": the server that runs on its

own, without relying on nine other sidecar servers to function.

I got interested in these kinds of applications because I build databases. I wrote BoltDB, which is a popular embedded K/V store in the Go ecosystem. BoltDB is reliable and, as you'd expect from an in-process database, it performs like a nitro-burning funny car. But BoltDB has limitations: its schema is defined in Go code, and so it's hard to migrate databases. You have to build your own tooling for it; there isn't even a REPL.

If you're careful, using this kind of database can get you a lot of performance. But for general-purpose use, you don't want to run your database off the open headers like a funny car. I thought about the kind of work I'd have to do to make BoltDB viable for more applications, and the conclusion I quickly reached was: that's what SQLite is for.

SQLite, as you are no doubt already typing into the message board comment, is not without its own limitations. The biggest of them is that a single-process application has a single point of failure: if you lose the server, you've lost the database. That's not a flaw in SQLite; it's just inherent to the design.

Enter Litestream

There are two big reasons everyone doesn't default to SQLite. The first is resilience to storage failures, and the second is concurrency at scale. Litestream has something to say about both concerns.

How Litestream works is that it takes control of SQLite's WAL-mode journaling. In WAL mode, write operations append to a log file stored alongside SQLite's main database file. Readers check both the WAL file and the main database to satisfy queries. Normally, SQLite automatically checkpoints

pages from the WAL back to the main database. Litestream steps in the middle of this: we open an indefinite read transaction that prevents automatic checkpoints. We then capture WAL updates ourselves, replicate them, and trigger the checkpointing ourselves.

> The most important thing you should understand about Litestream is that it's just SQLite. Your application uses standard SQLite, with whatever your standard SQLite libraries are. We're not parsing your queries or proxying your transactions, or even adding a new library dependency. We're just taking advantage of the journaling and concurrency features SQLite already has, in a tool that runs alongside your application. For the most part, your code can be oblivious to Litestream's existence.
>
> Or, think of it this way: you can build a Remix application backed by Litestream-replicated SQLite, and, while it's running, crack open the database using the standard `sqlite3` REPL and make some changes. It'll just work.
>
> You can read more about how this works here.

It sounds complicated, but it's incredibly simple in practice, and if you play with it you'll see that it "just works". You run the Litestream binary on the server your database lives on in "replicate" mode:

```
$ litestream replicate fruits.db s3://my-
  bukkit:9000/fruits.db
```

And then you can "restore" it to another location:

```
$ litestream restore -o fruits-replica.db
  s3://my-bukkit:9000/fruits.db
```

Now commit a change to your database; if you restore again then you'll see the change on your new copy.

> **Sidenote:** We'll replicate almost anywhere: to S3, or Minio; to Azure, or Backblaze B2, or Digital Ocean or Google Cloud, or an SFTP server.

The ordinary way people use Litestream today is to replicate their SQLite database to S3 (it's remarkably cheap for most SQLite databases to live-replicate to S3). That, by itself, is a huge operational win: your database is as resilient as you ask it to be, and easily moved, migrated, or mucked with.

But you can do more than that with Litestream. The upcoming release of Litestream will let you live-replicate SQLite directly between databases, which means you can set up a write-leader database with distributed read replicas. Read replicas can catch writes and redirect them to the leader; most applications are read-heavy, and this setup gives those applications a globally scalable database.

Litestream SQLite, Postgres, CockroachDB, or any other database

They all work on Fly.io; we do built-in persistent storage and private networking for painless

clustering, so it's easy to try new stuff out.

Try Fly →

You Should Take This Option More Seriously

One of my first jobs in tech in the early 2000s was as an Oracle Database Administrator (DBA) for an Oracle9i database. I remember spending hours poring over books and documentation to learn the ins and outs of the Oracle database. And there were a lot. The administration guide was almost a thousand pages—and that was just one of over a hundred documentation guides.

Learning what knobs to turn to optimize queries or to improve writes could make a big difference back then. We had disk drives that could only read tens of megabytes per second so utilizing a better index could change a 5-minute query into a 30 second query.

But database optimization has become less important for typical applications. If you have a 1 GB database, an NVMe disk can slurp the whole thing into memory in under a second. As much as I love tuning SQL queries, it's becoming a dying art for most application developers. Even poorly tuned queries can execute in under a second for ordinary databases.

Modern Postgres is a miracle. I've learned a ton by reading its code over the years. It includes a slew of features like a genetic query optimizer, row-level security policies, and a half dozen different types of indexes. If you need those features, you need them. But most of you probably don't.

And if you don't need the Postgres features, they're a liability. For example, even if you don't use multiple user accounts, you'll still need to configure and debug host-based authentication. You have to firewall off your Postgres server. And more features mean more documentation, which makes it difficult to understand the software you're running. The documentation for Postgres 14 is nearly 3,000 pages.

SQLite has a subset of the Postgres feature set. But that subset is 99.9% of what I typically need. Great SQL support, windowing, CTEs, full-text search, JSON. And when it lacks a feature, the data is already next to my application. So there's little overhead to pull it in and process it in my code.

Meanwhile, the complicated problems I really need to solve aren't really addressed by core database functions. Instead, I want to optimize for just two things: latency & developer experience.

So one reason to take SQLite seriously is that it's operationally much simpler. You spend your time writing application code, not designing intricate database tiers. But then there's the other problem.

The Light Is Too Damn Slow

We're beginning to hit theoretical limits. In a vacuum, light travels about 186 miles in 1 millisecond. That's the distance from Philadelphia to New York City and back. Add in layers of network switches, firewalls, and application protocols and the latency increases further.

The per-query latency overhead for a Postgres query within a single AWS region can be up to a millisecond. That's not Postgres being slow—it's you hitting the limits of how fast data can travel. Now, handle an HTTP request in a modern application. A dozen database queries and you've burned over 10ms before business logic or rendering.

There's a magic number for application latency: **responses in 100ms or less feel instantaneous**. Snappy applications make happy users. 100ms seems like a lot, but it's easy to carelessly chew it up. The 100ms threshold is so important that people pre-render their pages and post them on CDNs just to reduce latency.

We'd rather just move our data close to our application. How much closer? Really close.

SQLite isn't just on the same machine as your application, but actually built into your application process. When you put your data right next to your application, you can see per-query latency drop to 10-20 microseconds. That's micro, with a µ. A 50-100x improvement over an intra-region Postgres query.

But wait, there's more. We've effectively eliminated per-query latency. Our application is fast, but it's also simpler. We can break up larger queries into many smaller, more manageable queries, and spend the time we've been using to hunt down corner-casey N+1 patterns building new features.

Minimizing latency isn't just for production either. Running integration tests with a traditional client/server database easily grows to take minutes locally and the pain continues once you push to CI. Reducing the feedback loop from code change to test completion doesn't just save time but also preserves our focus while developing. A one-line change to SQLite will let you run it in-memory so you can run integration tests in seconds or less.

Small, Fast, Reliable, Globally Distributed: Choose Any Four

Litestream is distributed and replicated and, most importantly, still easy to get your head around. Seriously, go try it. There's just not much to know.

My claim is this: by building reliable, easy-to-use replication for SQLite, we make it attractive for all kinds of full-stack applications to run entirely on SQLite. It was reasonable to overlook this option 170 years ago, when the Rails Blog Tutorial was first written. But SQLite today can keep up with the write load of most applications, and replicas can scale reads out to as many instances as you choose to load-balance across.

Litestream has limitations. I built it for single-node applications, so it won't work well on ephemeral, serverless platforms or when using rolling deployments. It needs to restore all changes sequentially which can make database restores take minutes to complete. We're rolling out live replication, but the separate-process model restricts us to course-grained control over replication guarantees.

We can do better. For the past year, what I've been doing is nailing down the core of Litestream and keeping a focus on correctness. I'm happy with where we've landed. It started as a simple, streaming back up tool but it's slowly evolving into a reliable, distributed database. Now it's time to make it faster and more seamless, which is my whole job at Fly.io. There are improvements coming to Litestream — improvements that aren't at all tied to Fly.io! — that I'm psyched to share.

Litestream has a new home at Fly.io, but it is and always will be an open-source project. My plan for the next several years is to keep making it more useful, no matter where your application runs, and see just how far we can take the SQLite model of how databases can work.

LAST UPDATED • MAY 9, 2022

Ben Johnson
@benbjohnson

Next post ↑
Logbook - 2022-05-13

Previous post ↓
Logbook - 2022-05-05

COMPANY
About
Pricing
Jobs

ARTICLES
Blog
Phoenix Files
Laravel Bytes
Ruby Dispatch

RESOURCES
Docs
Support
Status

CONTACT
GitHub
Twitter
Community

LEGAL
Security
Privacy policy
Terms of service