Engineer

SQLite

a whole database in a single file

What it is

SQLite is an SQL database engine that runs inside the program using it, as a library, with no server process, no network port and no administrator, and keeps its tables, indexes, triggers and views together in one ordinary file on disk. The name also appears as `sqlite3` : that is what the command-line shell is called and the prefix every function in its C interface carries. The one C source file the project ships is named `sqlite3.c` . The project's own summary is that it is self-contained, serverless, zero-configuration and transactional, and it asks to be compared with `fopen()` rather than with Oracle. What it replaces is the home-made data file, not the database server.

The project began on 9 May 2000, and the code is in the public domain, so there is no licence to clear before shipping it inside a product. The file format is part of the bargain. A database written on a 32-bit machine can be copied to a 64-bit one, or between big-endian and little-endian processors, and opened unchanged. The developers say they intend to support SQLite through the year 2050, and the US Library of Congress lists its files as a recommended format for keeping data. With every feature enabled the library can be smaller than 900KiB, which is why it sits inside phones, cars, watches and desktop applications as well as on servers.

Inside, it is a complete SQL engine: joins, subqueries, views, triggers, and transactions that stay atomic, consistent, isolated and durable even when the power fails halfway through a write. Two design choices set it apart from server databases. The first is concurrency. Any number of connections may read at once, but only one may write at any instant, and the lock covers the whole file. The second is typing. A column's declared type is a preference, which SQLite calls affinity, and the type actually belongs to each value, so a column declared `INTEGER` will accept text that does not look like a number.

Against the alternatives, the line is clean. PostgreSQL, MySQL and SQL Server are shared services that many machines talk to over a network. SQLite is a private store for one application on one host. Against CSV or JSON files, it adds a query language, indexes, constraints and transactions, at the price of one dependency that most languages already bundle.

Use the concept in a decision

Put “What is SQLite?” to work in a real decision. We can help relate the concept to the system you are building or buying. A structure that matches how the business actually works, so the questions you need answered later are still possible to ask.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

For a large share of software the hard question about a database is not how fast it is, but who runs it. A server needs installing, upgrading, securing, monitoring and backing up, and somebody has to be paid to do all of that. SQLite removes the whole category. Opening a database is opening a file, a backup is a copy of that file, taken through the backup API while it is in use, and a test suite can give every test a fresh database of its own.

It also changes what a file can be. A program that would otherwise invent a format, write a parser for it and hope that a crash never lands mid-save can store its state in SQLite and get atomic saves for free. The project names this use directly: as an application file format, as a container for moving data between systems, where the receiver runs SQL instead of writing a parser, and as a scratch database for sorting and summarising data that arrived in some other shape.

The decision that matters is narrow, and it can be checked before a line of code is written: how many processes write, and on how many machines? One program on one host, writing in short bursts, is SQLite’s home ground, and plenty of websites with low to medium traffic run on it comfortably. Several servers writing to the same data, or a workload where many users write at every moment, belong on a client-server database. Size is rarely what decides it, since one file can hold up to 281 terabytes. The writer count is.

There are costs to weigh even inside that home ground. `ALTER TABLE` supports only renaming a table or a column and adding or dropping a column, so most other changes to a table's shape mean building a new table, copying the rows across and swapping the names. Foreign keys are switched off until each connection switches them on. And loose typing lets bad data in quietly, unless a table is declared to refuse it.

What it looks like in practice

At the documented development stage, twenty-three tables in one SQLite file, from achievements and companies to profiles and references, were the source of truth for a generator of CVs, reference sheets, portfolios and cover letters. One command turned an empty SQLite file into that database in nineteen ordered steps, and because each step could safely run twice, a rebuild over a database that already existed added no duplicate rows. The six output formats came from 41 Python modules of 10,580 lines that read that one file, with exactly two runtime dependencies. A document aimed at one kind of reader was a query over the achievements, chosen by a profile, rather than an edited copy. The price was plain: the system had become the only way to produce a document.

At the documented export stage, the same database was also a build input, because this website's own portfolio was generated from it. An exporter opened the SQLite file read-only and wrote 495 pages in three languages, using only Python's standard library, and the output was committed so the site could build without the generator's environment. Page addresses were stored as rows, one per address per achievement, so rewording a sentence no longer moved a page. That work also turned up the trap described in the next section: a period comparison against a text column matched all thirty-two rows.

The smallest role, a place to look at results, is also the most typical. A pipeline that turned raw electricity readings into hourly consumption per zone wrote its results either as CSV or into a lightweight SQLite database for inspection, and the Python version processed a day of data in 34ms and thirty days in 180ms. A database used that way needs no setup at all: anyone with the `sqlite3` shell can open the file and query it.

Where it goes wrong

Most SQLite trouble comes from treating it as a small PostgreSQL. It is a different machine with different edges.

A filter on the year matches every row. SQLite converts a value before a comparison only when one side carries an affinity. A function result such as `substr()` carries none, and neither does a bare number, so SQLite compares the text `'2019'` with the integer `2025` as they stand, and in its sort order every number comes before every piece of text:

```
CREATE TABLE stint (period TEXT, hours TEXT);
INSERT INTO stint VALUES
  ('2019-03', '9'), ('2024-11', '10'), ('2026-01', '42');

SELECT count(*) FROM stint
WHERE substr(period, 1, 4) >= 2025;                   -- 3

SELECT count(*) FROM stint
WHERE CAST(substr(period, 1, 4) AS INTEGER) >= 2025;  -- 1
```

Cast explicitly wherever text meets a number. For new tables, declare real types and add the `STRICT` keyword, available since version 3.37.0: a strict `INTEGER` column still accepts `'42'` and stores it as a number, and it rejects `'ten'` with an error instead of storing it.

Writes start failing with “database is locked”. One connection is holding the write lock, and a second writer either waits or gives up at once. The cause is usually a long transaction, or a reader in the default rollback-journal mode holding up a writer. Switch the database to write-ahead logging, where readers and the writer stop blocking each other, keep write transactions short, and set a busy timeout so a writer waits briefly instead of erroring. If the writers are genuinely many and constant, that is the signal to move to a server.

The file corrupts on a shared drive. SQLite depends on the operating system’s file locks, and many network filesystems implement those locks with bugs, so two machines can end up writing the same part of the file at once. Write-ahead logging does not work over a network filesystem at all, because every process using the database has to share memory on one host. Keep the database on a disk local to the one program that writes it, and put a server in front of it if other machines need the data.

A row outlives the parent it points at. Foreign key constraints are declared in the schema but stay inactive until each connection runs `PRAGMA foreign_keys = ON`, so a delete in a connection that skipped the pragma leaves orphans behind

without complaint. Set the pragma wherever the application opens a connection, in the one place that does so, and test that it is on.

Further reading

- About SQLite — the project's own description: in-process, serverless, public domain, one cross-platform file.
- Datatypes In SQLite — storage classes, type affinity and the comparison rules behind the trap above.
- STRICT Tables — how to make a table enforce its declared column types.
- Write-Ahead Logging — readers and a writer in parallel, and why it stays on one host.
- Appropriate Uses For SQLite — where it fits, where a client-server database fits better, and a checklist.
- SQLite Foreign Key Support — why constraints are off by default and how to switch them on.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is SQLite?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/sqlite/