SQLite Handbook

Flavio Copes
flaviocopes.com

November 2026

Contents

What SQLite is, and how it differs from a server 3
Installing and opening databases with `sqlite3` 4
File permissions, locks, and OS-level access 7
Storage classes, type affinity, and `STRICT` tables 8
`rowid`, `INTEGER PRIMARY KEY`, `AUTOINCREMENT`, and `WITHOUT ROWID` 11
Constraints, `PRAGMA foreign_keys`, and `ALTER TABLE` limits 12
JSON and full-text search (FTS5) built in 15
Transactions, locking, and `SQLITE_BUSY` 17
WAL mode and checkpointing 19
`EXPLAIN QUERY PLAN` and indexes 22
Using SQLite from Node.js (`node:sqlite`) 24
Migrations and in-memory tests 27
Backups, `VACUUM`, and integrity checks 28
When SQLite fits in production (and when to move to PostgreSQL) 31

Preface

This is the SQLite Handbook, written by Flavio Copes. You can find more programming tutorials and free handbooks at https://flaviocopes.com.

This book is about SQLite itself: the single-file architecture, storage classes and `STRICT` tables, `rowid` and `WITHOUT ROWID`, foreign keys that start turned off, JSON and FTS5 full-text search, `SQLITE_BUSY` locking, WAL mode, `EXPLAIN QUERY PLAN` and covering indexes, `node:sqlite`, in-memory tests, online backups with `VACUUM INTO`, and when SQLite fits in production.

It assumes you know SQL. If `SELECT`, `JOIN`, and `GROUP BY` are new to you, start with my SQL Handbook, which teaches the language on SQLite, PostgreSQL, and MySQL. This book goes deeper into how SQLite works.

We use the same real dataset as the SQL and PostgreSQL handbooks: the 2,192 published posts of my blog (over 2 million words of Markdown and their JSON frontmatter) plus a 300,000-row `post_views` table so indexes, `EXPLAIN QUERY PLAN`, WAL concurrency, and `VACUUM` show real, measurable numbers.

Every command, timing, and output in this book comes from a real run on my Mac, in November 2026, on these versions:

Software	Version
SQLite (`sqlite3` CLI)	3.54.0
Node.js (`node:sqlite`)	22.23.5
Bun (`bun:sqlite`, for comparison)	1.4.5

What SQLite is, and how it differs from a server

SQLite is a relational database packaged as a C library, not a server process.

When you use PostgreSQL or MySQL, a separate server process owns the data files and listens on a port. Your application connects over a socket, authenticates as a role, sends SQL, and waits for a reply.

With SQLite, there is no server process, no port, no daemon to keep alive, and no network hop. Your program links the SQLite library (or uses the one built into Node.js, Bun, Python, or Go) and reads and writes a single file on disk directly.

That single difference explains both SQLite's superpowers and its limits:

- **Zero setup.** Creating a database is opening a file path. Deleting it is `rm blog.db`.
- **Microsecond queries.** There is no network round trip between your code and the database, so a query that takes 1 ms over localhost to PostgreSQL takes 30 microseconds in SQLite.
- **Portable.** A SQLite file is cross-platform and self-contained. You can copy it from macOS to Linux, attach it to a bug report, or inspect it with `sqlite3` or Datasette.

- **One writer at a time.** Because there is no central server coordinating writes across machines, SQLite lives on one disk and allows one write transaction at a time.

What a SQLite file looks like

After loading my 2,192 blog posts into `blog.db`, the whole database is one 18 MB file:

```
ls -lh blog.db
file blog.db
```

```
-rw-r--r--@ 1 flaviocopes  staff    18M Nov 28 21:38 blog.db
blog.db: SQLite 3.x database, last written using SQLite version 3054001, writer version 2, read version
    ↪ 2, file counter 2, database pages 4605, cookie 0x3, schema 4, UTF-8, version-valid-for 2
```

Inside that file, SQLite stores tables, indexes, triggers, views, and the schema itself in fixed-size pages (4,096 bytes by default). Here 4,605 pages make up the 18 MB database.

Installing and opening databases with `sqlite3`

On macOS, `sqlite3` is already installed:

```
sqlite3 --version
```

```
3.54.0 2026-04-09 12:25:13 8fa8248e303219400c646a885e36dfc52eae33d83f4412e3f369b2be5373aapl (64-bit)
```

On Ubuntu or Debian, install it with `sudo apt install sqlite3`. On Windows, download the precompiled binaries from sqlite.org/download.html or install with `winget install SQLite.SQLite`.

Opening a file or an in-memory database

Pass a file path to open an existing database or create a new one:

```
sqlite3 blog.db
```

SQLite doesn't create the file until you actually write something (like a `CREATE TABLE` or `VACUUM;`). If you open a path and quit immediately, nothing is written to disk.

For quick experiments, pass `:memory:` instead of a filename:

```
sqlite3 -box :memory:
```

The `-box` flag turns on clean box-drawing tables from the start. Everything in `:memory:` lives in RAM and disappears when you quit.

Dot-commands you use every day

Inside `sqlite3`, commands starting with a dot are handled by the CLI itself, not sent as SQL:

Command	What it does
`.tables`	list all tables in the database
`.schema tags`	print the `CREATE TABLE` (and indexes) for `tags`
`.mode box`	switch output format (`box`, `table`, `csv`, `json`, `line`)
`.timer on`	print real, user, and system time after each query
`.headers on`	show column names (automatic in `box` and `table` modes)
`.read file.sql`	execute a SQL file
`.dump`	export the whole database as SQL statements
`.backup file.db`	take a safe online backup to another file
`.databases`	show attached database files and their paths
`.quit`	exit `sqlite3`

Let's inspect `blog.db` with a few of them:

```
sqlite3 -box blog.db ".tables"
sqlite3 blog.db ".schema tags"
sqlite3 -box blog.db "SELECT count(*) AS posts, sum(words) AS words FROM posts;"
```

```
post_tags     posts     tags
CREATE TABLE tags (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL UNIQUE
  ) STRICT;
+-------+---------+
| posts |  words  |
+-------+---------+
|  2192 | 2087634 |
+-------+---------+
```

You can put `.mode box`, `.headers on`, and `.timer on` in `~/.sqliterc` so every interactive `sqlite3` session starts with them enabled.

Exporting CSV and JSON from the command line

`sqlite3` is also a handy data-export tool. Pass `-csv -header` or `-json` to turn any query into a file another program can read:

```
sqlite3 -csv -header blog.db "SELECT slug, published_on, words FROM posts ORDER BY words DESC LIMIT 3;"
```

```
slug,published_on,words
grok-bot,2026-08-22,19669
```

```
claude-code-guide,2026-11-23,18848
livekit,2026-10-08,16045
```

```
sqlite3 -json blog.db "SELECT slug, words FROM posts ORDER BY words DESC LIMIT 2;"
```

```
[{"slug":"grok-bot","words":19669},
{"slug":"claude-code-guide","words":18848}]
```

To go the other direction and load a CSV file into a table, run `.import --csv data.csv my_table` inside `sqlite3`.

Loading the posts database

Here is the complete Node.js script (`load.mjs`) that built `blog.db` from the 2,192 published Markdown files of my blog using `node:sqlite`:

```
import { DatabaseSync } from 'node:sqlite'
import { readdir, readFile } from 'node:fs/promises'
import path from 'node:path'
import { parse } from 'yaml'

const dir = process.argv[2] || 'src/posts'
const dbPath = process.argv[3] || 'blog.db'
const today = new Date()

const posts = []
for (const file of (await readdir(dir)).sort()) {
  if (!file.endsWith('.md')) continue
  const source = await readFile(path.join(dir, file), 'utf8')
  const match = source.match(/^---\r?\n([\s\S]*?)\r?\n---\r?\n([\s\S]*)$/)
  if (!match) continue
  const data = parse(match[1]) ?? {}
  const body = match[2]
  const date = new Date(data.date)
  if (data.draft || Number.isNaN(date.getTime()) || date > today) continue
  const tags = [...new Set([data.tag ?? []].flat().map((t) => String(t).toLowerCase()))]
  const words = body.replace(/```[\s\S]*?```/g, '').split(/\s+/).filter(Boolean).length
  posts.push({
    slug: file.replace(/\.md$/, ''),
    title: String(data.title),
    published_on: date.toISOString().slice(0, 10),
    words,
    body,
    frontmatter: JSON.stringify(data),
    tags,
  })
}

posts.sort((a, b) => a.published_on.localeCompare(b.published_on) || a.slug.localeCompare(b.slug))
const tagNames = [...new Set(posts.flatMap((p) => p.tags))].sort()

const db = new DatabaseSync(dbPath)
db.exec(`
  PRAGMA journal_mode = WAL;
  PRAGMA foreign_keys = ON;
  CREATE TABLE posts (
    id INTEGER PRIMARY KEY,
    slug TEXT NOT NULL UNIQUE,
    title TEXT NOT NULL,
    published_on TEXT NOT NULL,
    words INTEGER NOT NULL CHECK (words >= 0),
    body TEXT NOT NULL,
```

```
    frontmatter TEXT NOT NULL
  ) STRICT;
  CREATE TABLE tags (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL UNIQUE
  ) STRICT;
  CREATE TABLE post_tags (
    post_id INTEGER NOT NULL REFERENCES posts(id) ON DELETE CASCADE,
    tag_id INTEGER NOT NULL REFERENCES tags(id) ON DELETE CASCADE,
    PRIMARY KEY (post_id, tag_id)
  ) STRICT, WITHOUT ROWID;
`)

const insPost = db.prepare(
  'INSERT INTO posts (slug, title, published_on, words, body, frontmatter) VALUES (?, ?, ?, ?, ?, ?)',
)
const insTag = db.prepare('INSERT INTO tags (name) VALUES (?)')
const insLink = db.prepare('INSERT INTO post_tags (post_id, tag_id) VALUES (?, ?)')

const t0 = performance.now()
db.exec('BEGIN')
const tagIds = new Map()
for (const name of tagNames) {
  const { lastInsertRowid } = insTag.run(name)
  tagIds.set(name, lastInsertRowid)
}
for (const p of posts) {
  const { lastInsertRowid } = insPost.run(
    p.slug, p.title, p.published_on, p.words, p.body, p.frontmatter,
  )
  for (const t of p.tags) insLink.run(lastInsertRowid, tagIds.get(t))
}
db.exec('COMMIT')
console.log(`Loaded ${posts.length} posts, ${tagNames.length} tags in ${(performance.now() - t0).
    ↪ toFixed(1)} ms`)
db.close()
```

```
Loaded 2192 posts, 49 tags in 88.0 ms
```

88 milliseconds to create three `STRICT` tables and insert over 2 million words of Markdown.

File permissions, locks, and OS-level access

Coming from PostgreSQL or MySQL, people often ask how to run `CREATE USER` or `GRANT SELECT` in SQLite. You can't. SQLite has no users, no passwords, and no roles.

Access control is handled entirely by the operating system. Whoever can read the file on disk can read the database. Whoever can write to the file (and its folder) can modify the database.

Locking down a database file on a server

On a Linux server, put the database in a dedicated directory owned by your application user, and restrict permissions so no other local user can read it:

```
sudo chown -R app:app /var/lib/myapp
sudo chmod 700 /var/lib/myapp
```

```
sudo chmod 600 /var/lib/myapp/blog.db
```

Notice the directory permission (`700`) is just as important as the file permission (`600`). SQLite needs write permission on the **directory**, not only the `.db` file, because it creates and removes companion journal files (`blog.db-journal`, `blog.db-wal`, `blog.db-shm`) next to the database. If the directory is read-only, writes fail even when the `.db` file itself is writable.

Opening a database read-only

When a reporting script or CLI only needs to read, open the file in read-only mode with a URI so a stray `DELETE` or `UPDATE` can never touch the data:

```
sqlite3 "file:blog.db?mode=ro" "SELECT count(*) FROM tags;" "DELETE FROM tags WHERE id = 1;"
```

```
49
Error in 3rd command line argument: attempt to write a readonly database
```

In Node.js (`node:sqlite`) you pass `{ readOnly: true }`, and in Bun (`bun:sqlite`) `{ readonly: true }`. Both use the same SQLite read-only flag under the hood.

Storage classes, type affinity, and `STRICT` tables

This is the single biggest surprise for anyone moving to SQLite from PostgreSQL or MySQL: **by default, SQLite does not enforce column types.**

Dynamic typing and the 5 storage classes

Every value stored in SQLite has one of five **storage classes**:

1. `NULL`
2. `INTEGER` (signed integer, 1 to 8 bytes)
3. `REAL` (64-bit IEEE floating-point number)
4. `TEXT` (UTF-8 string)
5. `BLOB` (raw bytes)

Notice there is no separate `BOOLEAN` (use `0` and `1` as `INTEGER`) and no native `DATE` or `TIMESTAMP` (store ISO-8601 strings like `'2026-11-28'` as `TEXT`, or Unix timestamps as `INTEGER`).

In a classic SQLite table, the column type you write in `CREATE TABLE` is only a hint called **type affinity**. SQLite tries to convert values to that type, and if it can't, it stores the value anyway with whatever type you passed:

```
CREATE TABLE loose (id INTEGER PRIMARY KEY, words INTEGER);
INSERT INTO loose VALUES (1, 500), (2, 'five hundred'), (3, 3.14), (4, NULL);
SELECT id, words, typeof(words) AS type FROM loose;
```

```
+----+--------------+---------+
| id |    words     |  type   |
+----+--------------+---------+
|  1 |          500 | integer |
|  2 | five hundred | text    |
|  3 |         3.14 | real    |
|  4 |              | null    |
+----+--------------+---------+
```

`words` was declared `INTEGER`, yet SQLite happily stored `'five hundred'` as `text` and `3.14` as `real` in the same column. That quirk has caused countless silent bugs when a bad string slips past application validation.

`STRICT` tables

Since SQLite 3.37, you can add `STRICT` at the end of `CREATE TABLE` to make SQLite enforce real column types:

```
CREATE TABLE typed (id INTEGER PRIMARY KEY, words INT NOT NULL, extra ANY) STRICT;
INSERT INTO typed VALUES (1, 500, 'anything');
INSERT INTO typed VALUES (2, 'five hundred', 42);
```

```
Error near line 7: cannot store TEXT value in INT column typed.words
```

That's the error you actually want. Inside a `STRICT` table, every column must use one of six types: `INT`, `INTEGER`, `REAL`, `TEXT`, `BLOB`, or `ANY` (for columns where you intentionally allow mixed types).

	Classic table	`STRICT` table
Allowed column type names	Anything (even `VARCHAR(255)` or `FOOBAR`)	`INT`, `INTEGER`, `REAL`, `TEXT`, `BLOB`, `ANY`
`'five hundred'` in `INT` column	Stored as `TEXT`	Rejected with an error
`'500'` in `INT` column	Converted to `500` (`INTEGER`)	Converted to `500` (`INTEGER`)
`NULL` without `NOT NULL`	Allowed	Allowed

My rule is simple: put `STRICT` on every new SQLite table you create. Every table in our `blog.db` uses it.

Dates, times, and time zones in SQLite

Because SQLite has no dedicated `DATE` or `TIMESTAMP` storage class, you pick one of two representations:

- **ISO-8601 UTC strings** (`TEXT`), like `'2026-11-28'` or `'2026-11-28T12:00:00Z'`. They are human-readable and sort chronologically with plain `<`, `>`, and `ORDER BY`.
- **Unix timestamps** (`INTEGER`), like `1795867200` seconds since 1970. Compact and easy to do arithmetic on.

SQLite ships built-in date functions (`date()`, `time()`, `datetime()`, `unixepoch()`, `strftime()`) that work on both formats and accept human-readable modifiers like `'+7 days'` or `'start of month'`:

```
SELECT
  date('2026-11-28', '+7 days') AS next_week,
  date('2026-11-28', 'start of month') AS month_start,
  unixepoch('2026-11-28T12:00:00Z') AS epoch,
  datetime(1795780800, 'unixepoch') AS back_to_iso;
```

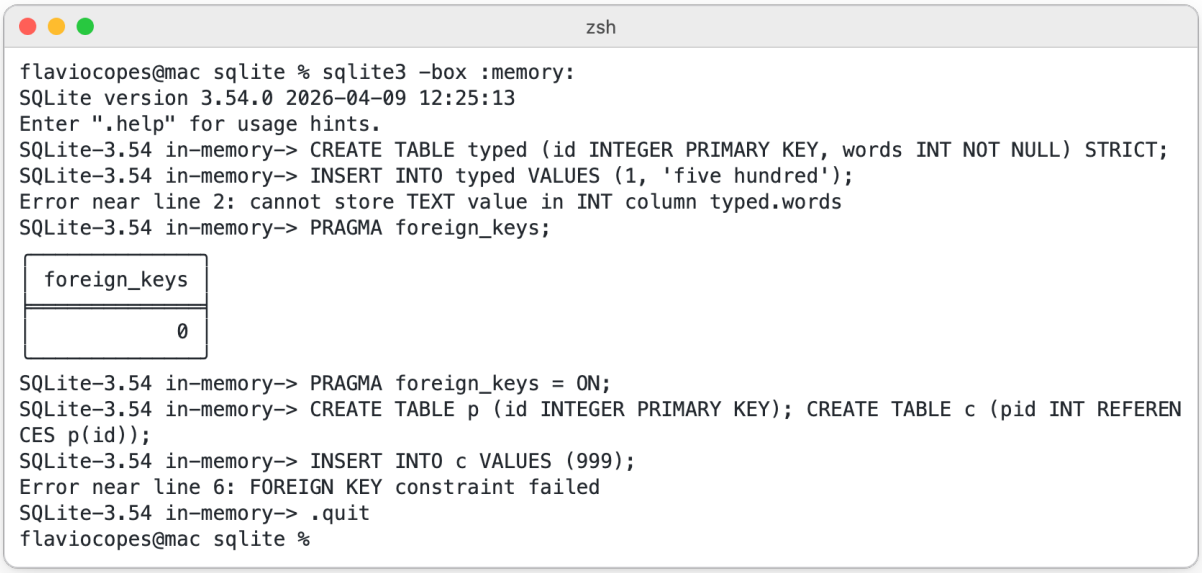
```
+------------+-------------+------------+---------------------+
| next_week  | month_start |   epoch    |     back_to_iso     |
+------------+-------------+------------+---------------------+
| 2026-12-05 | 2026-11-01  | 1795867200 | 2026-11-27 12:00:00 |
+------------+-------------+------------+---------------------+
```

Here is a real query on `blog.db` grouping the last five years of posts with `strftime('%Y', ...)`:

```
SELECT strftime('%Y', published_on) AS year,
       count(*) AS posts,
       round(avg(words)) AS avg_words
FROM posts
WHERE published_on >= date('2026-11-28', '-4 years', 'start of year')
GROUP BY 1 ORDER BY 1 DESC;
```

```
+------+-------+-----------+
| year | posts | avg_words |
+------+-------+-----------+
| 2026 |   509 |    2332.0 |
| 2025 |    25 |     637.0 |
| 2024 |    66 |     544.0 |
| 2023 |   108 |     435.0 |
| 2022 |   133 |     511.0 |
+------+-------+-----------+
```

Stick to UTC everywhere in the database, and convert to the user's local time zone only when displaying the date.



`rowid`, `INTEGER PRIMARY KEY`, `AUTOINCREMENT`, and `WITHOUT ROWID`

Every regular SQLite table automatically has a hidden 64-bit signed integer column called **`rowid`**:

```
CREATE TABLE notes (title TEXT);
INSERT INTO notes VALUES ('first'), ('second');
SELECT rowid, title FROM notes;
```

```
+-------+--------+
| rowid | title  |
+-------+--------+
|     1 | first  |
|     2 | second |
+-------+--------+
```

SQLite stores the table as a B-tree keyed by `rowid`, which makes lookups by `rowid` twice as fast as lookups through a secondary index.

`INTEGER PRIMARY KEY` vs `AUTOINCREMENT`

When you define a column as `id INTEGER PRIMARY KEY` (exact spelling: `INTEGER`, not `INT`), SQLite makes `id` a direct alias for `rowid`. Inserting a row without specifying `id` automatically assigns the next integer, with zero extra overhead.

People often add `AUTOINCREMENT` out of habit from MySQL. Let's see what `AUTOINCREMENT` actually does:

```
CREATE TABLE auto_demo (id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT);
INSERT INTO auto_demo (title) VALUES ('a'), ('b');
DELETE FROM auto_demo WHERE id = 2;
INSERT INTO auto_demo (title) VALUES ('c');
SELECT * FROM auto_demo;
```

```
SELECT * FROM sqlite_sequence;
```

```
+----+-------+
| id | title |
+----+-------+
|  1 | a     |
|  3 | c     |
+----+-------+
+-----------+-----+
|   name    | seq |
+-----------+-----+
| auto_demo |   3 |
+-----------+-----+
```

Without `AUTOINCREMENT`, deleting row 2 and inserting `'c'` reuses ID 2. With `AUTOINCREMENT`, SQLite creates a hidden `sqlite_sequence` table and updates it on every insert to guarantee IDs are never reused. That costs extra disk writes and lock contention.

The SQLite documentation itself recommends avoiding `AUTOINCREMENT` unless you have a strict requirement that deleted IDs must never be reused. Stick to `id INTEGER PRIMARY KEY`.

`WITHOUT ROWID`

For join tables like `post_tags (post_id, tag_id)` where the primary key is already a composite `(post_id, tag_id)`, a hidden `rowid` is wasted space. Adding `WITHOUT ROWID` stores the rows directly in the primary key's B-tree:

```
CREATE TABLE post_tags (
  post_id INTEGER NOT NULL REFERENCES posts(id) ON DELETE CASCADE,
  tag_id INTEGER NOT NULL REFERENCES tags(id) ON DELETE CASCADE,
  PRIMARY KEY (post_id, tag_id)
) STRICT, WITHOUT ROWID;
```

Use `WITHOUT ROWID` for narrow tables with a non-integer or composite primary key. Keep regular `rowid` tables for everything with large `TEXT` or `BLOB` columns (like `posts.body`), because `WITHOUT ROWID` keeps the entire row inside the B-tree pages.

Constraints, `PRAGMA foreign_keys`, and `ALTER TABLE` limits

SQLite supports `NOT NULL`, `UNIQUE`, `CHECK`, `DEFAULT`, generated columns, and foreign keys. There is one huge catch with foreign keys.

Foreign keys are OFF by default

For backwards compatibility with SQLite databases from 2009, every new connection starts with `foreign_keys = 0`. Watch what happens if you don't turn it on:

```
PRAGMA foreign_keys;
CREATE TABLE parent (id INTEGER PRIMARY KEY);
CREATE TABLE child (pid INTEGER REFERENCES parent(id) ON DELETE CASCADE);
INSERT INTO child VALUES (999);
SELECT count(*) AS orphans_allowed FROM child;
```

```
+--------------+
| foreign_keys |
+--------------+
|            0 |
+--------------+
+-----------------+
| orphans_allowed |
+-----------------+
|               1 |
+-----------------+
```

Row `999` went straight into `child` even though `parent` is empty. Now clear it and turn foreign keys on:

```
DELETE FROM child;
PRAGMA foreign_keys = ON;
INSERT INTO child VALUES (999);
```

```
Error near line 8: FOREIGN KEY constraint failed
```

`PRAGMA foreign_keys = ON` is a **per-connection** setting, not stored in the database file. Every time your app opens a SQLite connection (or you open `sqlite3`), run `PRAGMA foreign_keys = ON;`. One nice surprise: Node.js's built-in `node:sqlite` sets `foreign_keys = 1` automatically for you, as we verified when `post_views` rejected an out-of-range `post_id`. Setting it explicitly in your connection helper still protects you everywhere.

`ALTER TABLE` limits and the table-rebuild pattern

SQLite's `ALTER TABLE` can rename a table, rename a column, add a column, and drop a column (`DROP COLUMN`, added in SQLite 3.35).

It cannot change a column's type, add a `NOT NULL` constraint to an existing column, add a foreign key to an existing table, or change a `CHECK` constraint.

When you need one of those changes, SQLite's documented pattern is a transactional table rebuild:

1. `PRAGMA foreign_keys = OFF;`
2. `BEGIN TRANSACTION;`
3. Create `new_table` with the updated schema.
4. `INSERT INTO new_table SELECT ... FROM old_table;`
5. `DROP TABLE old_table;`

6. `ALTER TABLE new_table RENAME TO old_table;`
7. Recreate indexes and triggers.
8. `PRAGMA foreign_key_check;`
9. `COMMIT;`
10. `PRAGMA foreign_keys = ON;`

Because DDL in SQLite is transactional, if anything fails halfway through, `ROLLBACK` restores the original table untouched.

Generated columns, upserts, and `RETURNING`

SQLite supports **generated columns** (`VIRTUAL`, computed on read, or `STORED`, saved on disk and indexable):

```
CREATE TEMP TABLE articles (
  id INTEGER PRIMARY KEY,
  title TEXT NOT NULL,
  words INT NOT NULL,
  read_minutes INT GENERATED ALWAYS AS (max(1, (words + 199) / 200)) VIRTUAL
) STRICT;

INSERT INTO articles (title, words) VALUES ('Short note', 120), ('Deep guide', 3450);
SELECT * FROM articles;
```

```
+----+------------+-------+--------------+
| id |   title    | words | read_minutes |
+----+------------+-------+--------------+
|  1 | Short note |   120 |            1 |
|  2 | Deep guide |  3450 |           18 |
+----+------------+-------+--------------+
```

Modern SQLite also supports PostgreSQL-style `ON CONFLICT DO UPDATE` upserts and `RETURNING` on `INSERT`, `UPDATE`, and `DELETE`:

```
CREATE TEMP TABLE daily_hits (slug TEXT PRIMARY KEY, hits INT NOT NULL) STRICT;

INSERT INTO daily_hits VALUES ('node-sqlite', 1)
  ON CONFLICT (slug) DO UPDATE SET hits = daily_hits.hits + excluded.hits
  RETURNING slug, hits;

INSERT INTO daily_hits VALUES ('node-sqlite', 4)
  ON CONFLICT (slug) DO UPDATE SET hits = daily_hits.hits + excluded.hits
  RETURNING slug, hits;
```

```
+-------------+------+
|    slug     | hits |
+-------------+------+
| node-sqlite |    1 |
+-------------+------+
+-------------+------+
|    slug     | hits |
+-------------+------+
| node-sqlite |    5 |
+-------------+------+
```

Joining across two files with `ATTACH DATABASE`

Unlike PostgreSQL, where a connection sees only one database, SQLite lets you attach extra `.db` files to the same session and query across them in one statement:

```
ATTACH DATABASE 'wal.db' AS metrics;
SELECT m.name, m.value, (SELECT count(*) FROM main.posts) AS total_posts
FROM metrics.counters AS m;
DETACH DATABASE metrics;
```

```
+------+-------+-------------+
| name | value | total_posts |
+------+-------+-------------+
| hits |    99 |        2192 |
+------+-------+-------------+
```

That's especially useful when you keep large append-only analytics or logs in a separate SQLite file so your main database stays small and fast to back up.

JSON and full-text search (FTS5) built in

Modern SQLite builds include both the JSON functions (`JSON1`) and the `FTS5` full-text search engine out of the box. You don't need to install any extensions.

Querying JSON with `->`, `->>`, and `json_each`

Our `blog.db` stores each post's YAML frontmatter as a JSON string in `posts.frontmatter`. SQLite supports PostgreSQL-style `->` (returns JSON) and `->>` (returns a SQL value, unquoted):

```
SELECT json_type(frontmatter, '$.tag') AS tag_type, count(*) AS posts
FROM posts GROUP BY 1 ORDER BY 2 DESC;
```

```
+----------+-------+
| tag_type | posts |
+----------+-------+
| text     |  2085 |
| array    |   107 |
+----------+-------+
```

Finding the most recently updated posts using `->>`:

```
SELECT slug, frontmatter ->> '$.updated' AS updated
FROM posts
WHERE frontmatter ->> '$.updated' IS NOT NULL
ORDER BY updated DESC LIMIT 4;
```

```
+---------------------------------+------------+
|               slug              |  updated   |
```

```
+----------------------------------+------------+
| browser-dev-tools                | 2026-11-28 |
| web-storage-api                  | 2026-11-28 |
| websockets                       | 2026-11-28 |
| how-to-redirect-using-javascript | 2026-11-28 |
+----------------------------------+------------+
```

And `json_each()` expands a JSON array into rows. Here it counts the top 5 tags across all 2,192 posts, handling both single-string tags and tag arrays:

```
SELECT j.value AS tag, count(*) AS posts
FROM posts, json_each(
  CASE WHEN json_type(frontmatter, '$.tag') = 'array'
       THEN frontmatter -> '$.tag'
       ELSE json_array(frontmatter ->> '$.tag') END
) AS j
GROUP BY j.value ORDER BY 2 DESC LIMIT 5;
```

```
+----------+-------+
|   tag    | posts |
+----------+-------+
| js       |   352 |
| ai       |   211 |
| node     |   151 |
| react    |   102 |
| platform |   100 |
+----------+-------+
```

If you filter on a JSON field often, index the expression directly: `CREATE INDEX posts_updated_idx ON posts (frontmatter ->> '$.updated')`.

Full-text search with FTS5

An `FTS5` virtual table builds an inverted index over text columns, with English stemming (`porter`), boolean queries (`AND`, `OR`, `NOT`, `"phrases"`), `bm25()` relevance ranking, and `snippet()` excerpts:

```
.timer on
CREATE VIRTUAL TABLE posts_fts USING fts5(
  slug UNINDEXED,
  title,
  body,
  tokenize = 'porter unicode61'
);
INSERT INTO posts_fts (slug, title, body) SELECT slug, title, body FROM posts;

SELECT slug, round(bm25(posts_fts, 0.0, 5.0, 1.0), 2) AS score,
       snippet(posts_fts, 2, '[', ']', '...', 10) AS snippet
FROM posts_fts
WHERE posts_fts MATCH 'sqlite AND vacuum'
ORDER BY score LIMIT 3;
```

```
Run Time: real 0.234025 user 0.175810 sys 0.052016
+------------------------------+--------+-------------------------------------------------------------------
|             slug             | score  |                              snippet
    ↪     |
+------------------------------+--------+-------------------------------------------------------------------
| create-empty-sqllite-database | -14.49 | ...the [SQLite] header: sqlite3 storage.db "[VACUUM];"...
    ↪     |
| node-sqlite                  | -13.46 | ...Instead, let [SQLite] write a consistent online snapshot
    ↪ ...  |
| exe-dev                      |  -5.09 | ...The originals stay in Shelley's [SQLite] database if the
    ↪ ...  |
+------------------------------+--------+-------------------------------------------------------------------
Run Time: real 0.000943 user 0.000871 sys 0.000067
```

234 ms to index 2 million words, and **0.94 ms** per ranked search. `slug UNINDEXED` stores the slug so we can return it without indexing it as searchable text.

To keep `posts_fts` automatically synchronized whenever rows in `posts` are inserted, updated, or deleted, add three triggers:

```
CREATE TRIGGER posts_ai AFTER INSERT ON posts BEGIN
  INSERT INTO posts_fts (rowid, slug, title, body)
  VALUES (new.id, new.slug, new.title, new.body);
END;

CREATE TRIGGER posts_ad AFTER DELETE ON posts BEGIN
  DELETE FROM posts_fts WHERE rowid = old.id;
END;

CREATE TRIGGER posts_au AFTER UPDATE ON posts BEGIN
  DELETE FROM posts_fts WHERE rowid = old.id;
  INSERT INTO posts_fts (rowid, slug, title, body)
  VALUES (new.id, new.slug, new.title, new.body);
END;
```

Now every write to `posts` updates the full-text index inside the same transaction, with no separate indexing job to maintain.

Transactions, locking, and `SQLITE_BUSY`

SQLite's transaction and locking model is simpler than PostgreSQL's, and understanding it prevents the one error every SQLite user eventually hits: `database is locked`.

Why batching writes in one transaction matters

By default, every single `INSERT`, `UPDATE`, or `DELETE` runs in its own automatic transaction. Each commit flushes the journal to disk (`fsync`), which takes a fraction of a millisecond per statement.

I timed inserting 200 rows without a transaction against inserting 50,000 rows inside one explicit `BEGIN` / `COMMIT` on my Mac:

```
import { DatabaseSync } from 'node:sqlite'
```

```
const db = new DatabaseSync('/tmp/sqlb/bench.db')
db.exec('PRAGMA journal_mode = DELETE; CREATE TABLE items (id INTEGER PRIMARY KEY, n INT) STRICT;')
const ins = db.prepare('INSERT INTO items (n) VALUES (?)')

const t0 = performance.now()
for (let i = 0; i < 200; i++) ins.run(i)
const noTx = (performance.now() - t0).toFixed(1)

const t1 = performance.now()
db.exec('BEGIN')
for (let i = 0; i < 50000; i++) ins.run(i)
db.exec('COMMIT')
const withTx = (performance.now() - t1).toFixed(1)

console.log(`200 inserts without transaction: ${noTx} ms`)
console.log(`50,000 inserts in one transaction: ${withTx} ms`)
```

```
200 inserts without transaction: 41.4 ms
50,000 inserts in one transaction: 17.0 ms
```

250 times more rows in less than half the time: 50,000 inserts in 17 milliseconds. Whenever you insert or update more than a handful of rows, wrap the loop in one transaction.

`DEFERRED` vs `IMMEDIATE` transactions

SQLite has three ways to start a transaction:

- **`BEGIN` (default, `BEGIN DEFERRED`)**: acquires no lock until the first read or write. If a transaction reads first and then tries to upgrade to a write lock while another writer is active, SQLite fails immediately with `SQLITE_BUSY` (even if `busy_timeout` is set!) to avoid a deadlock.
- **`BEGIN IMMEDIATE`**: acquires the write lock right away, waiting up to `busy_timeout` if another writer is active.
- **`BEGIN EXCLUSIVE`**: blocks readers too (rarely needed in WAL mode).

Rule of thumb: if you know a transaction will write, start it with **`BEGIN IMMEDIATE`**.

Reproducing `database is locked` and fixing it with `busy_timeout`

Let's trigger `SQLITE_BUSY` for real with two `sqlite3` processes. Process 1 starts `BEGIN IMMEDIATE`, updates a counter, and sleeps 2 seconds before committing. While it sleeps, Process 2 tries to update the same table:

```
# Process 2, default busy_timeout = 0
sqlite3 lock.db "UPDATE counters SET value = value + 1 WHERE name = 'hits';"
```

```
Error in 2nd command line argument: database is locked
```

Because `busy_timeout` defaults to `0` milliseconds, Process 2 gave up immediately instead of waiting.

Now run Process 2 with `PRAGMA busy_timeout = 5000` (wait up to 5 seconds):

```
sqlite3 lock.db "PRAGMA busy_timeout = 5000; UPDATE counters SET value = value + 1 WHERE name = 'hits';
    ↪ SELECT * FROM counters;"


5000
hits|2
real 1.71
```

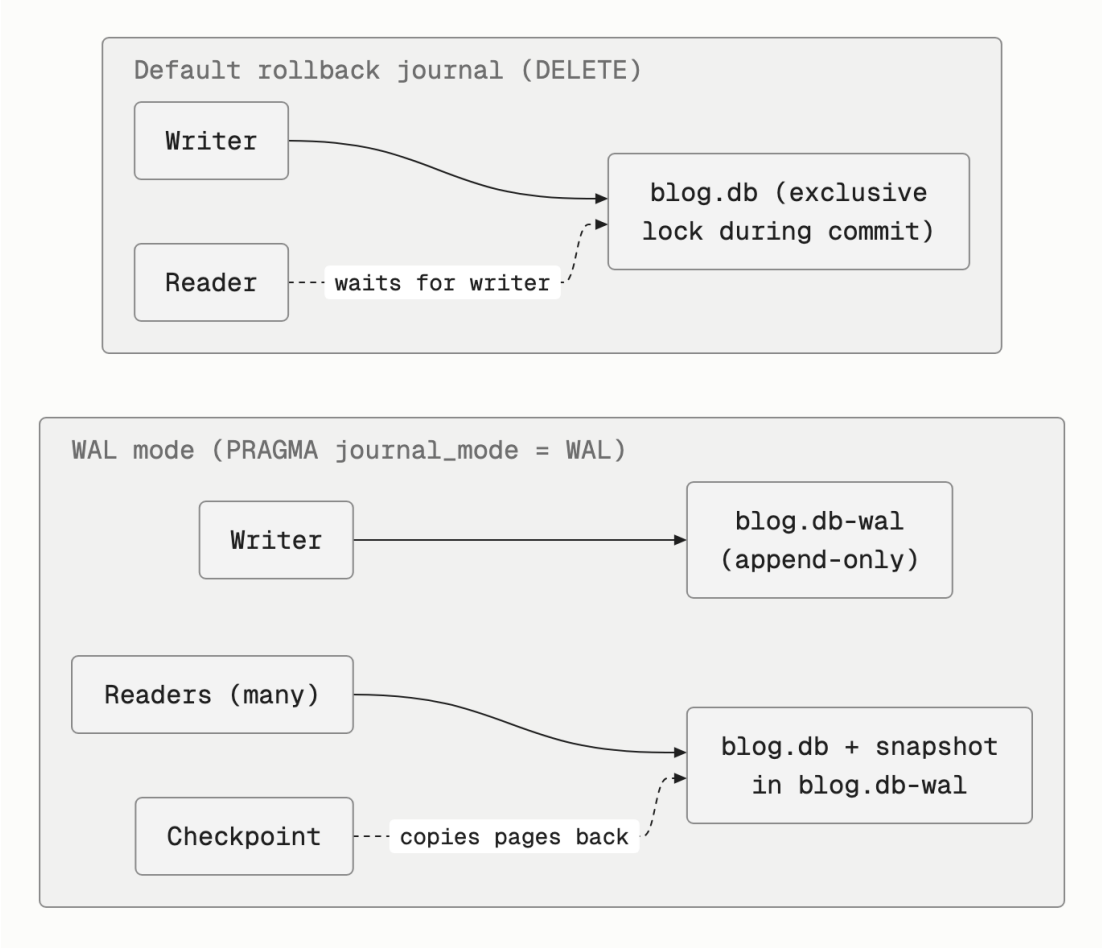
Process 2 waited 1.71 seconds for Process 1 to commit, acquired the lock, incremented the counter to 2, and succeeded cleanly. Every application connection should set `PRAGMA busy_timeout = 5000` at startup.

WAL mode and checkpointing

By default, a brand-new SQLite database uses the **rollback journal** (`PRAGMA journal_mode = DELETE`). When a writer modifies pages, it writes the original pages to a `-journal` file and locks the database so readers and writers take turns.

Switching to **Write-Ahead Logging (WAL)** changes everything:

```
PRAGMA journal_mode = WAL;
```



In WAL mode, writers append changes to `wal.db-wal` instead of overwriting `wal.db` directly, and a small shared-memory index `wal.db-shm` coordinates readers and writers. Readers never block writers, and a writer never blocks readers.

Let's verify it with two real processes. Process 1 runs `BEGIN IMMEDIATE`, updates `hits` from `10` to `99`, and sleeps 1.5 seconds before committing. While Process 1 is mid-transaction, we list the files and run a reader:

```
ls -lh wal.db*
sqlite3 -box wal.db "SELECT * FROM counters;"
```

```
-rw-r--r--@ 1 flaviocopes  staff    12K Nov 28 21:38 wal.db
-rw-r--r--@ 1 flaviocopes  staff    32K Nov 28 21:38 wal.db-shm
-rw-r--r--@ 1 flaviocopes  staff     0B Nov 28 21:38 wal.db-wal
+------+-------+
| name | value |
+------+-------+
| hits |    10 |
+------+-------+
```

The reader didn't wait at all, and it saw a clean snapshot (`10`), not the uncommitted write. Once Process 1 commits, the next read immediately returns `99`.

`synchronous = NORMAL` and checkpoints

In WAL mode, `PRAGMA synchronous = NORMAL` is safe against application crashes and power loss (you can at most lose the last uncommitted transaction on a sudden power cut, never corrupt the database) and makes commits dramatically faster because SQLite only calls `fsync` on checkpoints instead of every transaction.

SQLite automatically runs a **checkpoint** (copying committed frames from `-wal` back into the main `.db` file) whenever the WAL reaches 1,000 pages (~4 MB). You can also trigger one manually and shrink the `-wal` file back to 0 bytes:

```
PRAGMA wal_checkpoint(TRUNCATE);
```

```
0|0|0
```

The three numbers are `busy` (`0` = not blocked), `log` (pages in WAL), and `checkpointed` (pages copied).

	Rollback journal (`DELETE`)	WAL mode (`WAL`)
Readers while writing	Blocked during commit	Never blocked
Writers while reading	Blocked until readers finish	Never blocked
Extra files on disk	`*.db-journal` (temporary)	`*.db-wal` and `*.db-shm`
Persisted in database file	Yes	Yes (set once per database)

	Rollback journal (`DELETE`)	WAL mode (`WAL`)
Works on network drives (NFS/SMB)	Yes (slow)	No (requires local filesystem)

Turn on WAL mode for every local web app, API, or CLI database. Keep the default journal only if the file lives on a network filesystem or is distributed read-only.

Measuring `DELETE` vs `WAL + NORMAL` and tuning pragmas

To see how much `WAL` plus `synchronous = NORMAL` speeds up individual writes (where each HTTP request commits one row), I timed 500 single-row commits on my Mac:

```
import { DatabaseSync } from 'node:sqlite'

for (const [label, setup] of [
  ['DELETE + FULL (default)', 'PRAGMA journal_mode=DELETE; PRAGMA synchronous=FULL;'],
  ['WAL + NORMAL', 'PRAGMA journal_mode=WAL; PRAGMA synchronous=NORMAL;'],
]) {
  const path = `/tmp/sqlb/perf-${label.split(' ')[0].toLowerCase()}.db`
  const db = new DatabaseSync(path)
  db.exec(setup)
  db.exec('DROP TABLE IF EXISTS ev; CREATE TABLE ev (id INTEGER PRIMARY KEY, v INT) STRICT;')
  const ins = db.prepare('INSERT INTO ev (v) VALUES (?)')
  const t0 = performance.now()
  for (let i = 0; i < 500; i++) ins.run(i)
  const ms = (performance.now() - t0).toFixed(1)
  console.log(`${label.padEnd(26)} 500 single-row commits: ${ms} ms`)
  db.close()
}
```

```
DELETE + FULL (default)    500 single-row commits: 105.5 ms
WAL + NORMAL               500 single-row commits: 4.3 ms
```

4.3 milliseconds instead of 105.5 milliseconds, **24.5 times faster** for single-row commits, with zero changes to your SQL.

On a busy server with plenty of RAM, three more pragmas are worth knowing (here are their defaults on macOS SQLite 3.54.0):

Pragma	Default	Server starting point	What it does
`PRAGMA cache_size`	`2000` (pages, ~8 MB)	`-64000` (64 MB in KiB)	Pages cached in memory per connection (negative means KiB)
`PRAGMA temp_store`	`0` (default/file)	`MEMORY`	Keep temporary B-trees and sorts in RAM
`PRAGMA mmap_size`	`0` (off)	`268435456` (256 MB)	Memory-map the database file for faster reads

Start with the four core pragmas (`journal_mode = WAL`, `busy_timeout = 5000`, `foreign_keys = ON`, `synchronous = NORMAL`), and only tune `cache_size` or `mmap_size` after measuring a real bottleneck.

`EXPLAIN QUERY PLAN` and indexes

SQLite's query planner is simpler than PostgreSQL's, which makes `EXPLAIN QUERY PLAN` quick to read. There are only a few lines to recognize:

- `SCAN table`: reads every row in the table (full table scan)
- `SEARCH table USING INDEX idx (...)`: walks a B-tree index and fetches matching rows from the table
- `SEARCH table USING COVERING INDEX idx (...)`: answers the query from the index alone, never touching the table
- `USE TEMP B-TREE FOR ORDER BY`: sorts results in memory or a temp file because no index matches the sort order

To see real numbers, I added a `post_views` table to `blog.db` and inserted **300,000 rows** in 321 ms.

Before and after a covering index

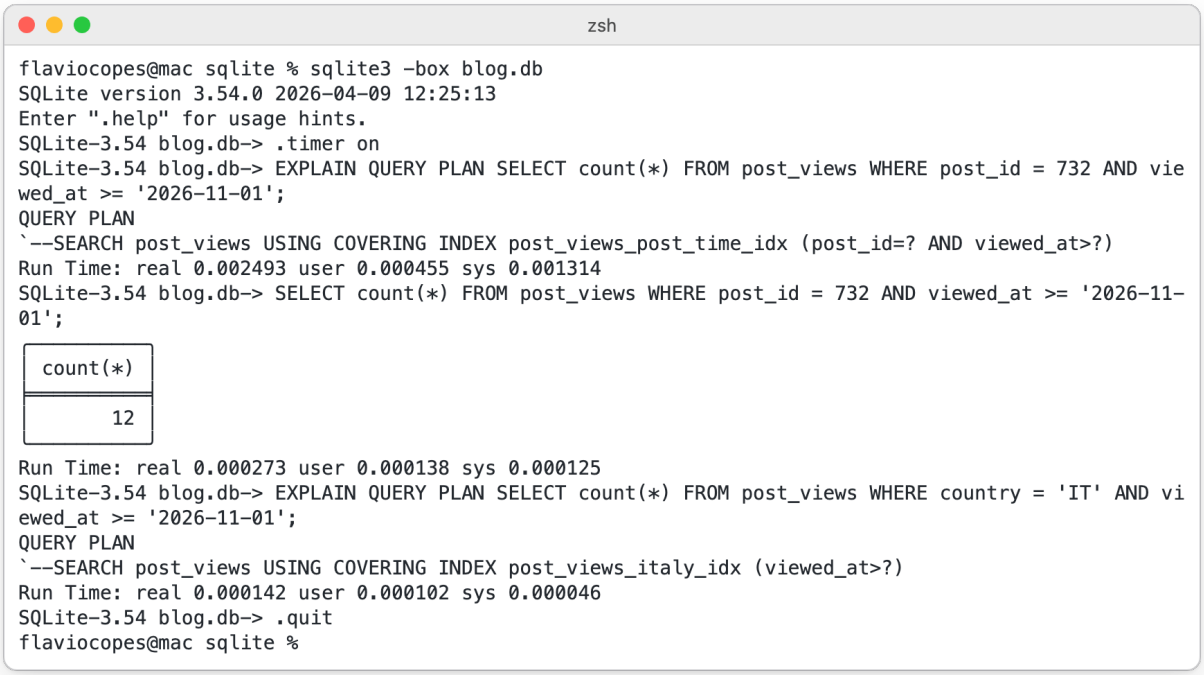
Count views for post 732 since November 1, 2026, with no index on `post_views`:

```
.timer on
EXPLAIN QUERY PLAN SELECT count(*) FROM post_views WHERE post_id = 732 AND viewed_at >= '2026-11-01';
SELECT count(*) FROM post_views WHERE post_id = 732 AND viewed_at >= '2026-11-01';


QUERY PLAN
`--SCAN post_views
+----------+
| count(*) |
+----------+
|       12 |
+----------+
Run Time: real 0.006223 user 0.004584 sys 0.001639
```

6.22 ms to scan 300,000 rows for 12 matches. Now add a composite index on `(post_id, viewed_at)`:

```
CREATE INDEX post_views_post_time_idx ON post_views (post_id, viewed_at);
EXPLAIN QUERY PLAN SELECT count(*) FROM post_views WHERE post_id = 732 AND viewed_at >= '2026-11-01';
SELECT count(*) FROM post_views WHERE post_id = 732 AND viewed_at >= '2026-11-01';


QUERY PLAN
`--SEARCH post_views USING COVERING INDEX post_views_post_time_idx (post_id=? AND viewed_at>?)
+----------+
| count(*) |
+----------+
|       12 |
+----------+
Run Time: real 0.000033 user 0.000023 sys 0.000010
```

From **6.22 ms** down to **0.033 ms** (33 microseconds, **188× faster**), and notice `USING COVERING INDEX`: because the query only needs `post_id` and `viewed_at`, SQLite never reads the main table at all.



Partial indexes, expression indexes, and `ANALYZE`

SQLite also supports **partial indexes** (indexing only rows matching a `WHERE`) and **expression indexes** (indexing a computed value like `lower(title)`):

```
CREATE INDEX post_views_italy_idx ON post_views (viewed_at) WHERE country = 'IT';
EXPLAIN QUERY PLAN SELECT count(*) FROM post_views WHERE country = 'IT' AND viewed_at >= '2026-11-01';

CREATE INDEX posts_lower_title_idx ON posts (lower(title));
EXPLAIN QUERY PLAN SELECT slug FROM posts WHERE lower(title) = 'introduction to postgresql';
```

```
QUERY PLAN
`--SEARCH post_views USING COVERING INDEX post_views_italy_idx (viewed_at>?)

QUERY PLAN
`--SEARCH posts USING INDEX posts_lower_title_idx (<expr>=?)
```

The Italy query counted 2,727 rows in 0.044 ms using only the partial index.

Run `ANALYZE;` once your table has real data so SQLite records row-count estimates in `sqlite_stat1`:

```
ANALYZE;
SELECT tbl, idx, stat FROM sqlite_stat1 WHERE tbl = 'post_views';
```

```
+------------+--------------------------+--------------+
|    tbl     |           idx            |     stat     |
```

```
+------------+--------------------------+--------------+
| post_views | post_views_italy_idx     | 30000 195    |
| post_views | post_views_post_time_idx | 300000 137 2 |
+------------+--------------------------+--------------+
```

`300000 137 2` tells the planner: 300,000 rows total, ~137 rows per distinct `post_id`, and ~2 rows per `(post_id, viewed_at)` pair. On a long-running app, run `PRAGMA optimize;` periodically (for instance before closing a connection) so SQLite refreshes stats only when needed.

Using SQLite from Node.js (`node:sqlite`)

Node.js 22 ships with a built-in SQLite module: `node:sqlite`. You don't need to compile native C++ addons with `node-gyp` just to use SQLite.

A clean startup helper

Every time your app opens a SQLite database, configure four pragmas right away so you get WAL concurrency, lock waiting, foreign-key enforcement, and fast commits:

```
import { DatabaseSync } from 'node:sqlite'

export function openDb(path) {
  const db = new DatabaseSync(path)
  db.exec(`
    PRAGMA journal_mode = WAL;
    PRAGMA busy_timeout = 5000;
    PRAGMA foreign_keys = ON;
    PRAGMA synchronous = NORMAL;
  `)
  return db
}
```

Prepared statements and SQL injection

Use `db.prepare(sql)` and bind user input with `?` (positional) or `:name` (named) parameters. Never interpolate strings into SQL:

```
const db = openDb('/tmp/sqlb/blog.db')
const input = "x' OR '1'='1"

const bad = db.prepare(`SELECT count(*) AS n FROM posts WHERE slug = '${input}'`).get()
const good = db.prepare('SELECT count(*) AS n FROM posts WHERE slug = ?').get(input)

console.log({ interpolated: bad.n, parameterized: good.n })
```

```
{ interpolated: 2192, parameterized: 0 }
```

The interpolated query matched all 2,192 posts because `OR '1'='1'` rewrote the `WHERE` clause. The parameterized query treated the whole string as a literal slug and returned `0`.

Every prepared statement has three methods:

- `.get(...params)`: returns the first matching row, or `undefined`
- `.all(...params)`: returns an array of all matching rows
- `.run(...params)`: executes an `INSERT`, `UPDATE`, or `DELETE`, returning `{ changes, lastInsertRowid }`

One real gotcha I hit while writing the tests for this chapter: `node:sqlite` returns **null-prototype objects** (`[Object: null prototype] { ... }`) to guard against prototype pollution. If you use `assert.deepEqual` from `node:assert/strict`, spread the row first (`{ ...row }`) so the prototype matches a plain object literal.

`node:sqlite` vs `bun:sqlite` vs `better-sqlite3`

	`node:sqlite`	`bun:sqlite`	`better-sqlite3`
Built into runtime	Yes (Node.js 22+)	Yes (Bun)	No (npm + native C++ build)
API style	Synchronous (`DatabaseSync`)	Synchronous (`Database`)	Synchronous
`PRAGMA foreign_keys` default	`1` (on)	`0` (off)	`0` (off)
Custom `transaction()` helper	Write a small wrapper	Built-in `db.transaction()`	Built-in `db.transaction()`
Production maturity	Newer (still prints ExperimentalWarning on Node 22)	Mature in Bun	Battle-tested on Node.js

For new Node.js 22+ projects, `node:sqlite` removes the native-addon install headache. If you run on Bun, use `bun:sqlite`. If you need an older Node.js version or custom SQLite build flags, `better-sqlite3` is still solid.

A complete HTTP search API with `node:sqlite`

Let's put `openDb` and our `posts_fts` index together into a real HTTP server (`server.mjs`) with zero npm dependencies:

```
import http from 'node:http'
import { openDb } from './app.mjs'

const db = openDb('/tmp/sqlb/blog.db')
const searchStmt = db.prepare(`
  SELECT slug, title,
         round(bm25(posts_fts, 0.0, 5.0, 1.0), 2) AS score
  FROM posts_fts
  WHERE posts_fts MATCH ?
  ORDER BY score
  LIMIT ?
`)

const server = http.createServer((req, res) => {
  const url = new URL(req.url, 'http://localhost')
  if (req.method === 'GET' && url.pathname === '/search') {
```

```
    const q = (url.searchParams.get('q') ?? '').trim()
    const limit = Math.min(Number(url.searchParams.get('limit')) || 5, 50)
    if (!q) {
      res.writeHead(400, { 'content-type': 'application/json' })
      return res.end(JSON.stringify({ error: 'missing q' }) + '\n')
    }
    try {
      const rows = searchStmt.all(q, limit)
      res.writeHead(200, { 'content-type': 'application/json' })
      return res.end(JSON.stringify({ q, count: rows.length, rows }, null, 2) + '\n')
    } catch (err) {
      res.writeHead(400, { 'content-type': 'application/json' })
      return res.end(JSON.stringify({ error: err.message }) + '\n')
    }
  }
  res.writeHead(404, { 'content-type': 'application/json' })
  res.end(JSON.stringify({ error: 'not found' }) + '\n')
})

server.listen(3020, () => console.log('listening on http://localhost:3020'))

process.on('SIGTERM', () => {
  server.close(() => {
    db.exec('PRAGMA optimize;')
    db.close()
    console.log('closed cleanly')
    process.exit(0)
  })
})
```

Testing it with `curl`:

```
curl -s 'http://localhost:3020/search?q=sqlite+AND+vacuum&limit=2'
```


```
{
  "q": "sqlite AND vacuum",
  "count": 2,
  "rows": [
    {
      "slug": "create-empty-sqllite-database",
      "title": "How to create an empty SQLite database",
      "score": -14.49
    },
    {
      "slug": "node-sqlite",
      "title": "node:sqlite: SQLite built into Node.js",
      "score": -13.46
    }
  ]
}
```


Notice the `SIGTERM` handler: before closing the database, it runs `PRAGMA optimize;` so SQLite refreshes query-planner statistics if any table changed significantly while the server was running.

Migrations and in-memory tests

You don't need a heavy migration framework for SQLite. SQLite stores a built-in integer called **`PRAGMA user_version`** in the database header, starting at `0`, specifically for applications to track their schema version.

Dead-simple migrations with `PRAGMA user_version`

Combine `user_version` with `BEGIN IMMEDIATE` and SQLite's **transactional DDL**. Unlike MySQL, SQLite lets you roll back `CREATE TABLE` and `ALTER TABLE`:

```
const mem = openDb(':memory:')
mem.exec('BEGIN; CREATE TABLE temp_only (id INT) STRICT; ROLLBACK;')
const tables = mem.prepare("SELECT name FROM sqlite_master WHERE type='table'").all()
console.log('tables after rolled-back CREATE TABLE:', tables)
```

```
tables after rolled-back CREATE TABLE: []
```

Because DDL rolls back cleanly, if a migration fails halfway through, the database stays at its previous `user_version` with no half-applied schema:

```
const migrations = [
  `CREATE TABLE notes (
     id INTEGER PRIMARY KEY,
     title TEXT NOT NULL,
     created_at TEXT NOT NULL DEFAULT (datetime('now'))
   ) STRICT;`,
  `ALTER TABLE notes ADD COLUMN pinned INTEGER NOT NULL DEFAULT 0 CHECK (pinned IN (0, 1));`,
]

export function migrate(db) {
  let { user_version: version } = db.prepare('PRAGMA user_version').get()
  for (let i = version; i < migrations.length; i++) {
    db.exec('BEGIN IMMEDIATE')
    try {
      db.exec(migrations[i])
      db.exec(`PRAGMA user_version = ${i + 1}`)
      db.exec('COMMIT')
    } catch (err) {
      db.exec('ROLLBACK')
      throw err
    }
  }
  return db.prepare('PRAGMA user_version').get().user_version
}
```

Fast, isolated tests with `:memory:`

Because SQLite runs in memory in under a millisecond, every test can open a brand-new `openDb(':memory:')`, run `migrate(db)`, test against the same schema and `STRICT` constraints as production, and throw the database away when it finishes:

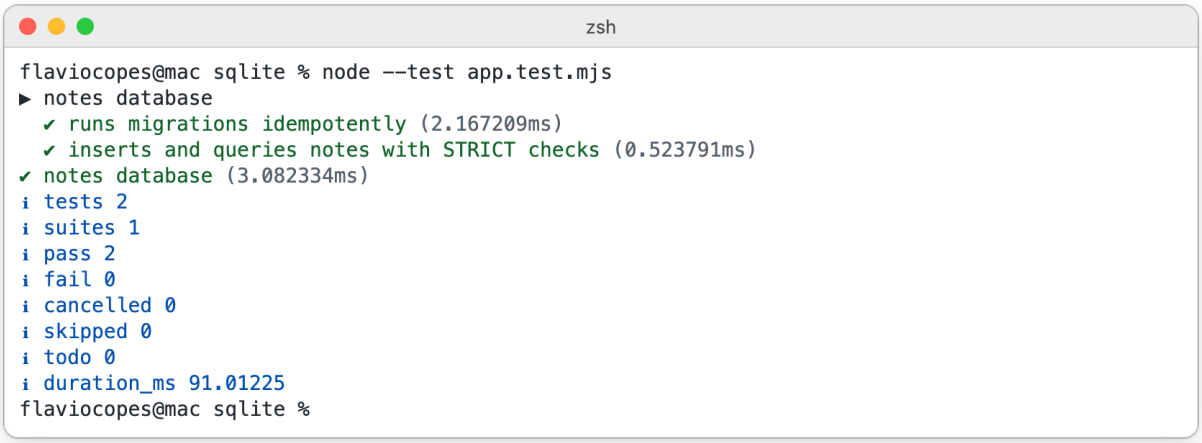
```
import assert from 'node:assert/strict'
import { describe, test } from 'node:test'
import { migrate, openDb } from './app.mjs'

describe('notes database', () => {
  test('runs migrations idempotently', () => {
```

```
    const db = openDb(':memory:')
    assert.equal(migrate(db), 2)
    assert.equal(migrate(db), 2)
    db.close()
  })

  test('inserts and queries notes with STRICT checks', () => {
    const db = openDb(':memory:')
    migrate(db)
    const ins = db.prepare('INSERT INTO notes (title, pinned) VALUES (:title, :pinned)')
    ins.run({ title: 'Learn SQLite', pinned: 1 })
    const row = db.prepare('SELECT title, pinned FROM notes WHERE pinned = ?').get(1)
    assert.deepEqual({ ...row }, { title: 'Learn SQLite', pinned: 1 })
    assert.throws(() => ins.run({ title: 'Bad', pinned: 'yes' }), /cannot store TEXT/)
    db.close()
  })
})
```

```
node --test app.test.mjs
```

```
 notes database
  [ok] runs migrations idempotently (0.615125ms)
  [ok] inserts and queries notes with STRICT checks (0.703917ms)
[ok] notes database (1.809834ms)
(i) tests 2
(i) suites 1
(i) pass 2
(i) fail 0
(i) duration_ms 66.859875
```

Both tests ran in under 2 milliseconds, with zero leftover files and zero mocking of the database layer.



Backups, `VACUUM`, and integrity checks

Because a SQLite database is "just a file", it's tempting to back it up with `cp blog.db blog.db.bak`. **Don't do that on a live database.**

If a writer commits while `cp` is halfway through reading the file (or if recent commits

are sitting in `blog.db-wal` and you only copied `blog.db`), the copy can be corrupt or missing recent writes.

Safe online backups: `.backup` and `VACUUM INTO`

SQLite gives you two safe ways to back up a live database while your application keeps running:

```
sqlite3 blog.db ".backup backup-dot.db"
sqlite3 blog.db "VACUUM INTO 'backup-vac.db';"
```

```
-rw-r--r--@ 1 flaviocopes  staff    62M Nov 28 21:40 backup-dot.db
-rw-r--r--@ 1 flaviocopes  staff    63M Nov 28 21:40 backup-vac.db
-rw-r--r--@ 1 flaviocopes  staff    62M Nov 28 21:40 blog.db
```

`VACUUM INTO` is my favorite because it's plain SQL (so you can run it straight from Node.js or Bun), it produces a single compacted file with no `-wal` needed, and it never blocks readers.

Always verify a backup with `PRAGMA integrity_check`:

```
sqlite3 backup-vac.db "PRAGMA integrity_check;" "SELECT count(*) FROM post_views;"
```

```
+-----------------+
| integrity_check |
+-----------------+
| ok              |
+-----------------+
+----------+
| count(*) |
+----------+
|   300000 |
+----------+
```

Backup method	Safe while live	Compacts free pages	Output
`cp app.db`	No (can corrupt WAL/writes)	No	Raw file copy
`.dump`	Yes (inside a read transaction)	Yes (rebuilt on import)	Plain SQL text
`.backup file.db`	Yes (uses Online Backup API)	No	Binary `.db` file
`VACUUM INTO 'file.db'`	Yes (plain SQL, single file)	Yes	Compacted `.db` file
Litestream	Yes (continuous WAL streaming)	N/A	Continuous snapshots in S3/R2

Reclaiming disk space with `VACUUM`

Deleting rows in SQLite marks their pages as free for future writes, but it doesn't shrink the file on disk. Watch what happens when I delete all 300,000 rows from `post_views` on a copy of `blog.db`:

```
ls -lh shrink.db
sqlite3 shrink.db "DELETE FROM post_views;"
ls -lh shrink.db
sqlite3 shrink.db "VACUUM;"
ls -lh shrink.db
```

```
-rw-r--r--@ 1 flaviocopes  staff    63M Nov 28 21:40 shrink.db
-rw-r--r--@ 1 flaviocopes  staff    63M Nov 28 21:40 shrink.db
-rw-r--r--@ 1 flaviocopes  staff    43M Nov 28 21:40 shrink.db
```

After `DELETE`, the file stayed 63 MB. After `VACUUM`, it dropped to 43 MB. Keep in mind that `VACUUM` rewrites the whole database into a temporary copy and holds an exclusive lock while doing so, so run it during a maintenance window, not on every delete.

Introspecting a database with `PRAGMA` functions

SQLite exposes its metadata as table-valued `PRAGMA` functions you can query with regular `SELECT`:

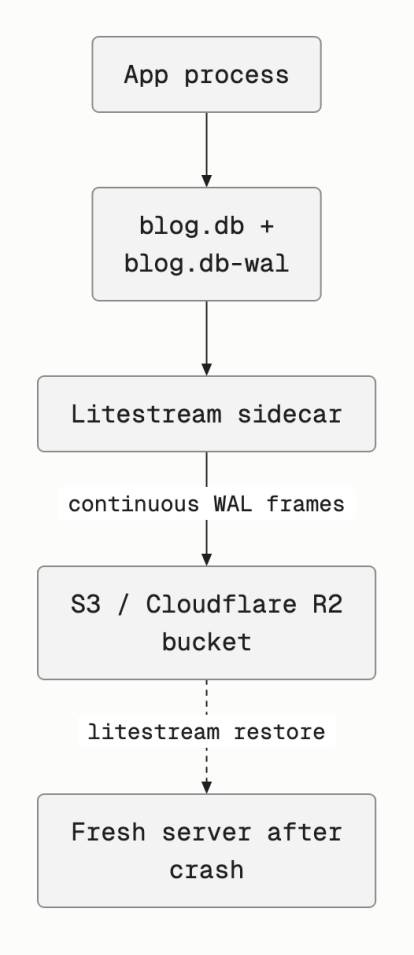
```
PRAGMA table_info(tags);
PRAGMA index_list(post_views);

SELECT
  (SELECT page_count FROM pragma_page_count) AS pages,
  (SELECT page_size FROM pragma_page_size) AS page_bytes,
  (SELECT freelist_count FROM pragma_freelist_count) AS free_pages;
```

```
+-----+------+---------+---------+------------+----+
| cid | name |  type   | notnull | dflt_value | pk |
+-----+------+---------+---------+------------+----+
|   0 | id   | INTEGER |       0 |            |  1 |
|   1 | name | TEXT    |       1 |            |  0 |
+-----+------+---------+---------+------------+----+
+-----+--------------------------+--------+--------+---------+
| seq |           name           | unique | origin | partial |
+-----+--------------------------+--------+--------+---------+
|   0 | post_views_italy_idx     |      0 | c      |       1 |
|   1 | post_views_post_time_idx |      0 | c      |       0 |
+-----+--------------------------+--------+--------+---------+
+-------+------------+------------+
| pages | page_bytes | free_pages |
+-------+------------+------------+
| 15886 |       4096 |          0 |
+-------+------------+------------+
```

When `free_pages` grows to a large fraction of `pages`, it's a signal that a `VACUUM` will reclaim real disk space.

Continuous backups with Litestream

If you run SQLite in production on a VPS, nightly `VACUUM INTO` backups mean you could lose up to 24 hours of writes if the disk dies. **Litestream** solves that: it runs as a tiny sidecar process that watches your `-wal` file and streams every change continuously to S3 or Cloudflare R2.



With Litestream, your recovery window drops from a day to a few seconds, while your application still talks to a plain local SQLite file.

When SQLite fits in production (and when to move to PostgreSQL)

SQLite is no longer "just a toy database for prototyping". A single server running SQLite in WAL mode handles thousands of reads per second and hundreds of writes per second with sub-millisecond latency and zero network overhead.

Still, it has clear physical boundaries. Knowing them upfront saves you from picking the wrong tool.

Need	SQLite	PostgreSQL
Read-heavy web app on one VPS	Excellent (microsecond reads, zero ops)	Great, more setup
Desktop app, CLI, local-first tool, or test suite	The obvious choice	Overkill
Multiple app servers writing at the same time	Doesn't fit (one local disk)	Built for it
High concurrent write throughput	Bottlenecked by single writer lock	Handles many concurrent writers (MVCC)
Fine-grained roles, row-level security, network access	None (OS file permissions only)	Built in
Rich extensions (PostGIS, pgvector, custom types)	Limited compared to Postgres	Massive ecosystem
Backups & point-in-time recovery	`VACUUM INTO` or Litestream to S3/R2	`pg_dump`, `pg_basebackup` + WAL, or managed host

My checklist for using SQLite in production

If your app runs on one server, is mostly reads with modest writes, and you want the simplest possible stack:

1. Use `STRICT` on every table and `INTEGER PRIMARY KEY` (without `AUTOINCREMENT`).
2. Run the four startup pragmas on every connection: `journal_mode = WAL`, `busy_timeout = 5000`, `foreign_keys = ON`, `synchronous = NORMAL`.
3. Start write transactions with `BEGIN IMMEDIATE` and keep them short.
4. Check slow queries with `EXPLAIN QUERY PLAN` and add covering indexes where they count.
5. Lock down the database folder (`chmod 700`) and file (`chmod 600`).
6. Back up with `VACUUM INTO` or Litestream, and test restoring it.

When you outgrow a single machine or need many concurrent writers, my PostgreSQL Handbook picks up right where SQLite leaves off.