SQLite Course

Flavio Copes

Updated August 3, 2026

Contents

Learn SQLite from the ground up: database files, the command-line shell, schemas, transactions, indexes, application access, backups, and practical limits.

What you'll learn: Build a small SQLite database, query it safely from an application, inspect its performance, and back it up without running a database server.

A database in one file

Choose SQLite, open the shell, and create your first database file.

What SQLite is

Understand how an embedded database differs from a database server and why one ordinary file can hold a complete relational database.

SQLite is a relational database engine embedded inside the program that uses it. There is no separate server process to install, configure, or keep running.

Compare that with PostgreSQL or MySQL. Those are **database servers**: a separate program runs all the time, your application connects to it over a network socket, and it authenticates with a username and password. Someone has to install it, start it, patch it, and back it up.

SQLite removes all of that. It's a library compiled into your application. When your code runs a query, the SQLite code inside your own process reads and writes the database directly. No network, no credentials, no daemon.

One file is the whole database

A database normally lives in one file. Your application opens that file and SQLite reads and writes it directly while still giving you tables, SQL, indexes, constraints, and transactions.

You can see this yourself. This shell command opens (and on first write, creates) a database:

```
sqlite3 notes.db
```

Inside the shell, plain SQL works exactly as you'd expect from any relational database:

```
SELECT sqlite_version();
-- 3.43.2
```

Everything you create inside it — every table, index, and row — ends up in that single `notes.db` file. Copy the file to another machine and you copied the database. Delete the file and the database is gone.

This is radically different from a server database, where the data lives inside a directory tree managed by the server and you interact with it only through connections.

Where you already use it

That small operational footprint is the main reason SQLite appears in local tools, desktop and mobile apps, tests, caches, and many web applications.

Your phone runs dozens of SQLite databases right now: browsers store history in it, messaging apps store conversations in it, macOS uses it throughout the system. It's likely the most widely deployed database engine in the world.

One expectation to reset before you continue: ”no server” does not mean ”not a real database”. SQLite is ACID-compliant and tested obsessively. What it doesn't do is accept connections from other machines. If a query needs to travel over a network, something else has to carry it. That boundary, and when it matters, is exactly what the next lesson covers.

When SQLite fits

Choose SQLite for the workloads it handles well and recognize the concurrency or deployment requirements that point to a database server.

SQLite is a great default when one application owns the database and the data belongs on one machine.

That covers more ground than most people assume. Command-line tools, desktop and mobile apps, test suites, caches, data analysis scripts, and plenty of web applications all fit that description. A single web server handling thousands of requests per day is comfortably inside SQLite territory.

The write concurrency question

It handles many readers well. Writes are serialized, so applications with many concurrent writers need more care.

Serialized writes mean SQLite applies one write transaction at a time. Each write is fast — often well under a millisecond — so a queue of writers usually drains quickly. But if dozens of processes hammer the database with writes simultaneously, some of them wait, and past a point they start failing with `database is locked` errors.

A database server is usually a better fit when several machines must write directly to the same database. SQLite has no network protocol of its own, and putting the database file on a network filesystem is a known source of corruption. If your architecture is "three app servers, one database", that database should be PostgreSQL or MySQL.

A quick decision test

Ask these questions about the project in front of you:

```
Does more than one machine need to write to this data?   -> server
Do many processes write heavily at the same time?        -> server
Is the data owned by one app on one machine?             -> SQLite
Is this a test suite, CLI tool, or local cache?          -> SQLite
```

Notice the questions are about writes and machines, not about data size. SQLite handles databases of many gigabytes without trouble.

Don't buy scale you don't need

Start from the workload you have. Do not add a database server only because the project might become large one day.

Choosing PostgreSQL "just in case" costs you real money and effort today: a server to run, credentials to manage, a connection pool to tune, one more thing to back up and monitor. SQLite costs you a file.

And migrating later is a well-understood move. Your SQL knowledge transfers, and because SQLite lives behind the same query patterns, swapping the engine is far cheaper than carrying an unneeded server for years.

Install and open SQLite

Verify the SQLite command-line shell, install it when needed, and open your first local database.

Start by checking whether the SQLite shell is already available:

```
sqlite3 --version
# 3.43.2 2023-10-10 13:08:14 4310099cce5a487035fa535dd3002c59ac7f1d1bec68d7cf317fd3e769484790
```

The first number is what matters. Anything reasonably recent works for this course; you'll want 3.37 or newer later for `STRICT` tables.

macOS includes a system copy. On macOS you can also install a newer copy with `brew install sqlite` — note that Homebrew installs it keg-only, so the system `sqlite3` stays first on your PATH unless you use the full Homebrew path. On Windows or Linux, use the package linked from the official SQLite download page or your operating system's package manager. On Debian or Ubuntu that's:

```
sudo apt install sqlite3
```

There is nothing else to install. No service to start, no port to open, no user account to create. The `sqlite3` binary is the entire toolchain.

Open your first database

Now open a database named `notes.db`:

```
sqlite3 notes.db
```

You should see a `sqlite>` prompt:

```
SQLite version 3.43.2 2023-10-10 13:08:14
Enter ".help" for usage hints.
sqlite>
```

Run `SELECT sqlite_version();` to check the library used by this shell, then leave with `.quit`.

That version check matters more than it looks. The `sqlite3` shell and the SQLite library linked into your application (Node, Python, an ORM) can be different versions with different feature support. When something works in the shell but fails in your app, compare versions first.

A quiet gotcha

If you run `ls` after quitting, you may find no `notes.db` file at all. That's expected: SQLite delays creating the file until you write a schema or data to it. Opening alone isn't enough. The next lesson uses that behavior to create the real file.

The flip side bites more often: `sqlite3` never complains about a wrong path. Mistype `sqlite3 notse.db` and you get a fresh empty database instead of an error. If a database you know has tables suddenly looks empty, check which file you actually opened before assuming data loss.

Create a SQLite database file

Create a real SQLite database by writing its first schema instead of making an empty placeholder file.

Opening a new path does not immediately write a database file. SQLite creates the real file when you write its first schema or data.

This design is deliberate. It lets programs open a database ”just in case” without littering the disk with empty files. But it means the moment of creation is the first write, not the open.

Open the database:

```
sqlite3 notes.db
```

Create its first table:

```
CREATE TABLE notes (
  id INTEGER PRIMARY KEY,
  title TEXT NOT NULL
) STRICT;
```

The statement returns silently. In SQLite's shell, no output means success — errors are always printed.

Run `.databases` to see the full path, then leave with `.quit`. `notes.db` now contains a valid SQLite header and schema.

Verify what you created

Back in your terminal, look at the file:

```
ls -l notes.db
# -rw-r--r--  1 flavio  staff  12288 Aug  3 10:12 notes.db
file notes.db
# notes.db: SQLite 3.x database, ...
```

The `file` command recognizes the SQLite header in the first 16 bytes. A real database is never zero bytes: the smallest one is a few kilobytes, because SQLite allocates whole pages (4096 bytes by default).

You can also confirm the schema survived by reopening and running `.schema`, which prints the `CREATE TABLE` statement back to you.

Two things not to do

Do not use `touch` to create a database. It only creates a zero-byte placeholder. SQLite will accept a zero-byte file and treat it as an empty database, but any tooling that checks the header sees an invalid file, and you gain nothing — SQLite creates the file properly on its own.

Also, do not treat an ordinary file copy as a safe backup while the database is active. A `cp` that runs mid-write can capture a half-applied transaction, and in WAL mode part of your committed data lives in a sidecar file next to the main one. We will make a consistent backup later in the course with tools built for exactly that.

One habit worth forming now: keep the database file in a directory your application owns, like a `data/` folder, not in whatever directory you happened to launch the shell from. Relative paths are the main way people end up with three half-filled copies of the same database.

Use the SQLite shell

Open a database, run SQL, and use dot commands to inspect the shell without confusing them with SQL statements.

Open a database file from the terminal:

```
sqlite3 notes.db
```

You're now inside the shell, and everything you type goes to one of two interpreters. Knowing which one you're talking to saves a lot of confusion.

SQL statements such as `SELECT 1;` end with a semicolon. If you press enter without one, the shell shows a continuation prompt (`...>`) and waits — it thinks your statement isn't finished. Type the missing `;` and press enter.

Dot commands start with a dot, belong to the shell itself, take no semicolon, and must fit on one line. Try `.databases`, `.tables`, `.schema`, and `.help`.

```
sqlite> .tables
notes
sqlite> .schema notes
CREATE TABLE notes (
  id INTEGER PRIMARY KEY,
  title TEXT NOT NULL
) STRICT;
```

`.tables` lists what exists. `.schema` prints the exact SQL that created it — this is how you inspect any unfamiliar SQLite file. `.databases` shows the full path of the open file, which settles any doubt about which database you're actually in.

Make query output readable

The default output is bare values separated by `|`. Two settings make it much friendlier:

```
sqlite> .headers on
sqlite> .mode box
sqlite> SELECT id, title FROM notes;

  id       title

  1    Plan the week
```

These settings last for the session. Leave with `.quit` and they reset.

Run a query without entering the shell

You can also pass SQL directly as an argument, which is handy in scripts:

```
sqlite3 notes.db "SELECT count(*) FROM notes;"
# 1
```

The shell runs the query, prints the result, and exits.

The classic mistake here is mixing the two languages: typing `.tables;` (dot command with a semicolon) or `SELECT 1` (SQL without one). The first prints an error, the second leaves you stuck at the `...>` continuation prompt. If the shell seems to hang, look at the prompt — `...>` means it's waiting for your semicolon, and typing `;` on its own line finishes the statement.

Check your understanding: a database in one file

Check the SQLite architecture, database files, shell commands, installation, and the workloads that fit an embedded database.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Schema and data

Create tables and work with SQLite types, keys, constraints, and rows.

SQLite storage classes and type affinity

Use SQLite storage classes, type affinity, and STRICT tables without inventing native boolean or date types.

SQLite values use five storage classes: `NULL`, `INTEGER`, `REAL`, `TEXT`, and `BLOB`.

In an ordinary table, a declared type gives a column an affinity. SQLite tries to convert values toward that affinity, but it can still store a value with another storage class.

Use a `STRICT` table when you want SQLite to reject values that do not match the declared type:

```
CREATE TABLE tasks (
  id INTEGER PRIMARY KEY,
  title TEXT NOT NULL,
  completed INTEGER NOT NULL DEFAULT 0
    CHECK (completed IN (0, 1)),
  created_at TEXT NOT NULL
) STRICT;
```

`STRICT` tables require SQLite 3.37 or newer. Check with `SELECT sqlite_version();` if the shell reports a syntax error.

SQLite has no separate boolean or date storage class. This schema stores booleans as `0` or `1` and dates as consistently formatted text, such as `2026-07-29T14:30:00Z`.

My advice is to choose one representation, enforce it with constraints where possible, and use it everywhere in the application.

Create a table

Create a small table with a primary key and required columns before inserting any data.

You created the `notes` table when you made the database file. Now add a table for tags:

```
CREATE TABLE tags (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL UNIQUE
) STRICT;
```

The schema gives each tag an identifier, requires a name, and prevents duplicate tag names.

Let's unpack each decision, because every table you create in SQLite repeats this pattern.

`id INTEGER PRIMARY KEY` gives every row a unique numeric identifier that SQLite assigns for you. The next lesson explains why this exact spelling is special in SQLite.

`name TEXT NOT NULL UNIQUE` stacks two rules on the column. `NOT NULL` rejects a missing name;

`UNIQUE` rejects a duplicate one. Rules that must always hold belong in the schema, where SQLite enforces them for every writer.

`STRICT` at the end tells SQLite to enforce the declared types. Without it, SQLite's flexible typing lets a well-meaning script store text in an integer column. Strict tables accept only `INT`, `INTEGER`, `REAL`, `TEXT`, `BLOB`, and `ANY` as column types, and they reject values that don't match. It needs SQLite 3.37 or newer.

Verify the table exists

`.tables` now lists both tables, and `.schema tags` prints the definition back:

```
sqlite> .tables
notes  tags
sqlite> .schema tags
CREATE TABLE tags (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL UNIQUE
) STRICT;
```

What `.schema` prints is not a reconstruction. SQLite stores the literal text of your `CREATE TABLE` statement and gives it back verbatim.

Watch out for silent type names

If you bring habits from MySQL or PostgreSQL, you might write `VARCHAR(20)` instead of `TEXT`. On an ordinary table SQLite accepts it, quietly maps it to text affinity, and ignores the 20 — a 500-character name inserts without complaint. No error, just an unenforced limit.

On a `STRICT` table the same column fails immediately:

```
sqlite> CREATE TABLE t (name VARCHAR(20)) STRICT;
Parse error: unknown datatype for t.name: "VARCHAR(20)"
```

That early error is a feature. My advice is to declare every table `STRICT`, use `TEXT` for strings, and enforce lengths with a `CHECK` constraint when a limit genuinely matters. A later lesson on storage classes covers what flexible typing does in detail.

rowid and INTEGER PRIMARY KEY

Understand the special relationship between an INTEGER PRIMARY KEY column and SQLite's internal row identifier.

Most SQLite tables have an internal integer `rowid`. It's how SQLite physically addresses each row: the table is stored as a tree keyed by rowid, so looking a row up by it is the fastest access path the engine has.

You can query this hidden column even though you never declared it:

```
SELECT rowid, title FROM notes;
-- 1|Plan the week
```

The alias rule

A column declared exactly as `INTEGER PRIMARY KEY` becomes an alias for that rowid. SQLite assigns the next available value when an insert omits the column.

That's why our `notes` table gets ids 1, 2, 3 without any sequence or auto-increment setup:

```
INSERT INTO notes (title) VALUES ('Buy groceries');
SELECT id, rowid FROM notes WHERE title = 'Buy groceries';
-- 2|2
```

`id` and `rowid` return the same value because they are the same value. One column, two names.

The word "exactly" carries weight. `INT PRIMARY KEY` does not create the alias — only the full spelling `INTEGER PRIMARY KEY` does. This is one of SQLite's sharpest edges: with `INT`, the column is an ordinary unique column and the table keeps a separate hidden rowid, which costs you the fast path and can leave the column `NULL`. If your ids behave strangely, check the declaration first.

Why you rarely need AUTOINCREMENT

You normally do not need `AUTOINCREMENT`. It changes reuse rules and adds overhead; use it only when old identifiers must never appear again.

Here's the difference. Without `AUTOINCREMENT`, SQLite picks one more than the current largest rowid. If you delete the row with the highest id, its number can be handed out again to a future insert. With `AUTOINCREMENT`, SQLite records the highest value ever used in an internal `sqlite_sequence` table and never reissues it, at the cost of extra bookkeeping on every insert.

For most tables, reuse is harmless — the ids only need to be unique among live rows. It matters when identifiers escape the database: an id printed on an invoice, embedded in a URL, or synced to another system should never be recycled to mean a different record. That's the case `AUTOINCREMENT` exists for.

After an insert, you can read the id SQLite just assigned:

```
SELECT last_insert_rowid();
-- 2
```

Every SQLite driver exposes this, and it's the standard way to link freshly inserted rows to related tables.

Protect data with constraints

Enable and verify foreign keys, then use constraints to reject invalid rows close to the data.

Constraints make database rules explicit. A required title, a unique name, or a check on a boolean value belongs in the schema.

The alternative is enforcing rules in application code, and that fails the moment anything else touches the database: a migration script, a second service, you in the shell at midnight. The schema is the one gate every write passes through.

You've already used `NOT NULL` and `UNIQUE`. Watch them work:

```
sqlite> INSERT INTO tags (name) VALUES ('planning');
sqlite> INSERT INTO tags (name) VALUES ('planning');
Runtime error: UNIQUE constraint failed: tags.name (19)
```

The first insert succeeds, the duplicate is rejected, and nothing was written by the failed statement. The error names the table and column, so violations are easy to trace.

Foreign keys are off by default

SQLite foreign-key enforcement is a connection setting. Enable it before starting a transaction:

```
PRAGMA foreign_keys = ON;
```

Now read the setting back:

```
PRAGMA foreign_keys;
-- 1
```

The result must be `1`. This is a per-connection switch kept off for historical compatibility — it does not persist in the database file. Do this check for every connection unless your database library documents that it enables foreign keys for you.

Once enforcement is active, this relationship rejects a `note_id` that does not exist:

```
CREATE TABLE note_tags (
  note_id INTEGER NOT NULL REFERENCES notes(id) ON DELETE CASCADE,
  tag_id INTEGER NOT NULL REFERENCES tags(id) ON DELETE CASCADE,
  PRIMARY KEY (note_id, tag_id)
) STRICT;
```

Try it. Inserting a link to a note id that was never created fails with `Runtime error: FOREIGN KEY constraint failed (19)`. And `ON DELETE CASCADE` means deleting a note automatically deletes its rows in `note_tags`, so no orphaned links accumulate.

Find violations that already exist

Enforcement only guards new writes. Rows inserted while the pragma was off may already break the rules. Use `PRAGMA foreign_key_check;` to find existing violations. No returned rows means SQLite found none.

Run that check after enabling foreign keys on an existing database and after any bulk import. Discovering orphaned rows during a quiet moment is a much better experience than discovering them when a join silently returns fewer results than expected.

The realistic failure here is forgetting the pragma in the application. Everything appears to work — inserts succeed, queries run — but references are never verified, and months of orphaned rows pile up. Make the `PRAGMA foreign_keys` check part of every connection setup, right next to opening the database.

Insert, update, and delete rows

Use ordinary SQL to change SQLite data and always make the target of an update or delete explicit.

SQLite uses the same basic data-changing statements you learned in the SQL course:

```
INSERT INTO notes (title) VALUES ('Plan the week');
UPDATE notes SET title = 'Plan Monday' WHERE id = 1;
DELETE FROM notes WHERE id = 1;
```

Naming the columns in the `INSERT` matters. `INSERT INTO notes VALUES (...)` relies on column order, and it breaks the first time a migration adds a column. Columns you leave out get their default — here `id` is assigned automatically because it's the `INTEGER PRIMARY KEY`.

You can insert several rows in one statement, which is also faster because SQLite commits once

instead of once per row:

```
INSERT INTO notes (title) VALUES
  ('Plan the week'),
  ('Review pull requests'),
  ('Write the changelog');
```

Get data back from a write

SQLite supports `RETURNING` (since 3.35), which hands you values from the rows a statement touched:

```
INSERT INTO notes (title) VALUES ('Call the accountant')
RETURNING id;
-- 4
```

That saves the separate `SELECT last_insert_rowid()` round trip, and it works on `UPDATE` and `DELETE` too.

Verify what a statement did

The shell doesn't print anything for a successful write, so ask how many rows the last statement changed:

```
UPDATE notes SET title = 'Plan Monday' WHERE id = 1;
SELECT changes();
-- 1
```

`changes()` returning `0` after an update you expected to work usually means the `WHERE` clause matched nothing — a wrong id, a typo in a title. The statement didn't fail; it just found no target. Check with a `SELECT` using the same `WHERE`.

The missing WHERE clause

The classic disaster in every SQL database:

```
UPDATE notes SET title = 'Plan Monday';
```

No `WHERE`, so every row in the table now has the same title. `DELETE FROM notes;` with no `WHERE` empties the table the same way. SQLite executes both without hesitation and there is no undo outside a transaction or a backup.

Run the matching `SELECT` first when a `WHERE` clause affects important data. `SELECT count(*) FROM notes WHERE id = 1;` costs a second and tells you exactly how many rows the write will touch. For anything beyond a throwaway database, wrap risky changes in `BEGIN` ... `COMMIT` so a wrong result can still be rolled back — transactions get their own lesson shortly.

Check your understanding: schema and data

Check storage classes, affinity, primary keys, rowid behavior, constraints, foreign keys, and safe row changes.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Transactions and performance

Understand locking, WAL mode, indexes, and query plans.

Use transactions

Group related changes so SQLite commits all of them or none of them.

Start a transaction before changes that must succeed together:

```
BEGIN;
UPDATE accounts SET balance = balance - 20 WHERE id = 1;
UPDATE accounts SET balance = balance + 20 WHERE id = 2;
COMMIT;
```

Without the transaction, a failure after the first `UPDATE` removes money from one account without adding it to the other. The transaction makes the pair one unit: either `COMMIT` makes both changes durable, or `ROLLBACK` discards both.

Applications should roll back when any statement fails:

```
db.exec('BEGIN')

try {
  debit.run(20, 1)
  credit.run(20, 2)
  db.exec('COMMIT')
} catch (error) {
  db.exec('ROLLBACK')
  throw error
}
```

SQLite starts an implicit transaction for a statement when you do not start one explicitly. An explicit transaction matters when several statements share one business outcome, or when a batch of inserts should avoid the overhead of committing each row separately.

`BEGIN` is deferred by default: SQLite does not acquire a write lock until the first write. `BEGIN IMMEDIATE` attempts to start the write transaction immediately. That can be useful when you prefer to discover writer contention before doing preparatory work.

SQLite allows multiple readers, but only one writer at a time. Do not open a transaction, call a remote API, wait for user input, and then commit. That holds the write transaction much longer than necessary and makes other writers wait or fail with `SQLITE_BUSY`.

Keep the transaction around database work only. Compute and validate first, begin, perform the related statements, commit, then do slow external work.

Try it: add a constraint that makes the second transfer update fail. Confirm that your error path rolls back and the first balance remains unchanged.

Test SQLite locking with two connections

Use two shell connections to see one-writer locking, busy timeouts, and BEGIN IMMEDIATE in practice.

SQLite allows many readers, but only one connection can write at a time. Let's see the lock instead

of only reading about it.

Open `notes.db` in two terminals. In both shells, set a three-second busy timeout:

```
.timeout 3000
```

In terminal A, claim the write lock and keep the transaction open:

```
BEGIN IMMEDIATE;
UPDATE notes SET title = 'Held by terminal A' WHERE id = 1;
```

`BEGIN IMMEDIATE` asks for the write lock at the start. Do not commit yet.

In terminal B, try another write:

```
BEGIN IMMEDIATE;
```

Terminal B waits for up to three seconds, then reports `database is locked`. Return to terminal A and run `COMMIT;`. Terminal B can now retry and acquire the lock.

A busy timeout handles a brief conflict. It does not fix long transactions. Never wait for a network request or slow calculation while holding a write transaction open.

Use write-ahead logging when it helps

Understand WAL concurrency, sidecar files, local-filesystem requirements, and checkpoints.

By default, SQLite uses a rollback journal: a writer copies the pages it's about to change into `notes.db-journal`, modifies the main file, and readers must wait while that happens. Write-ahead logging inverts the flow — new changes go to a log file and the main database stays untouched until later.

Enable write-ahead logging with:

```
PRAGMA journal_mode = WAL;
-- wal
```

The pragma answers `wal` to confirm the switch. Unlike most pragmas, this one is persistent: it's recorded in the database file, so every future connection opens in WAL mode without repeating it.

SQLite stores new changes in `notes.db-wal`. Connections coordinate through `notes.db-shm`. These sidecar files are part of the active database, so do not delete, move, or copy them separately.

You can watch them appear:

```
ls notes.db*
# notes.db  notes.db-shm  notes.db-wal
```

What WAL buys you

WAL mode lets readers keep using an existing snapshot while one writer appends changes. It does not create multiple writers.

That's the whole trade in two sentences. In the default mode, a write blocks readers and readers block the writer. In WAL mode, readers never wait for the writer and the writer never waits for readers, which is why WAL is the usual choice for web applications where reads and writes overlap constantly. Writes are still serialized — one at a time — exactly as before.

Checkpoints

A **checkpoint** moves committed pages from the WAL file back into the main database. SQLite normally checkpoints automatically once the WAL grows past about 1000 pages. You can request a non-blocking checkpoint with:

```
PRAGMA wal_checkpoint(PASSIVE);
-- 0|55|55
```

The three numbers report whether the checkpoint was blocked, the WAL's size in pages, and how many pages were moved back. When the last two match, the whole log was transferred.

The failure mode to watch: a long-running read transaction pins the WAL, checkpoints can't complete past that reader's snapshot, and `notes.db-wal` grows without bound. If you find a WAL file several times larger than the database, look for a connection holding a transaction open — often a forgotten shell session or a leaked connection in application code.

Keep a WAL database on local storage. WAL needs shared memory between processes on the same machine and does not work over a network filesystem. If the database must live on NFS or a mounted volume, stay on the default rollback journal.

Add a useful index

Create an index for a real lookup and remember that every index also makes writes more expensive.

If the application frequently finds notes by title, an index can avoid scanning every row:

```
CREATE INDEX notes_title_idx ON notes(title);
```

An index is a second data structure ordered by the indexed value. SQLite can search that smaller structure and use the stored row reference instead of examining every row in `notes`.

Verify the access plan before and after creating it:

```
EXPLAIN QUERY PLAN
SELECT id, title FROM notes WHERE title = 'SQLite checklist';
```

On an unindexed table you will usually see a table scan. After creating the index, the plan should mention searching with `notes_title_idx`. The plan is evidence; the presence of an index in the schema is not proof that a query uses it.

For a query with more than one filter, column order matters:

```
CREATE INDEX notes_owner_created_idx
ON notes(owner_id, created_at);
```

This is useful for queries that constrain `owner_id` and optionally sort or filter by `created_at`. It is generally not useful for a query that only constrains `created_at`, because the leading indexed column is missing.

Indexes are not free. Every insert, relevant update, and delete must update them. They also make the database file larger. A low-cardinality column such as a boolean may be a poor standalone index when most rows share the same value.

Add indexes for observed access patterns: frequent filters, joins, uniqueness requirements, and ordering that matters. Remove indexes that duplicate another index's leading columns unless a measured workload justifies both.

Try it: create 10,000 notes owned by several users. Compare the query plan for `WHERE owner_id = ? ORDER BY created_at` before and after adding the compound index.

Read EXPLAIN QUERY PLAN

Ask SQLite how it plans to find rows before guessing that a query needs another index.

When a query feels slow, the temptation is to add an index and hope. SQLite can tell you exactly what it intends to do instead, and reading that plan takes seconds.

Prefix a query with `EXPLAIN QUERY PLAN`:

```
EXPLAIN QUERY PLAN
SELECT id, title FROM notes WHERE title = 'Plan Monday';
```

On a table with no index on `title`, the output looks like this:

```
QUERY PLAN
`--SCAN notes
```

`SCAN` means SQLite will read every row in `notes` and test each one against the `WHERE` clause. For a hundred rows that's instant; for a million it's your slow query.

After `CREATE INDEX notes_title_idx ON notes(title);`, the same command reports:

```
QUERY PLAN
`--SEARCH notes USING INDEX notes_title_idx (title=?)
```

`SEARCH` means SQLite jumps to the matching entries through an index instead of visiting every row. The parenthesis shows which index columns it can use. You may also see `USING COVERING INDEX`, which is better still: every column the query needs lives in the index, so SQLite never touches the table at all.

Reading multi-table plans

For a join, the plan prints one line per table, in the order SQLite will process them:

```
EXPLAIN QUERY PLAN
SELECT notes.title, tags.name
FROM notes
JOIN note_tags ON note_tags.note_id = notes.id
JOIN tags ON tags.id = note_tags.tag_id;
```

Each line is either a `SCAN` or a `SEARCH`. One `SCAN` on the outermost table is normal — something has to drive the loop. A `SCAN` on an inner table is the expensive pattern, because it repeats for every outer row.

Note that `EXPLAIN QUERY PLAN` doesn't run the query. It only shows the strategy, so it's safe to use on any statement, even an `UPDATE` or `DELETE`.

Plans depend on data

Look for a table scan or an index search. Measure with realistic data because a scan can be the right plan for a tiny table.

This is the mistake that catches people: they check the plan on a ten-row development database, see a `SCAN`, and add indexes that production never needed — or worse, they see a `SEARCH` in development

and assume production behaves the same. SQLite's planner uses what it knows about table sizes. Test the plan against a database that resembles the real one, and run `ANALYZE;` after bulk-loading data so the planner's statistics match reality.

Check your understanding: transactions and performance

Check atomic changes, locking, WAL mode, busy writers, indexes, and SQLite query plans.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Applications and operations

Use prepared statements, test in memory, migrate, back up, and know the limits.

Use prepared statements

Bind values separately from SQL so user input stays data instead of becoming executable query text.

Prepare SQL with placeholders, then bind each value through the database library.

Suppose a search box contains this value:

```
' OR 1=1 --
```

Building the query with string concatenation turns that value into SQL syntax:

```
const sql = `SELECT * FROM notes WHERE title = '${search}'`
```

Use a placeholder instead:

```
const sql = `SELECT * FROM notes WHERE title = ?`
const rows = db.prepare(sql).all(search)
```

The SQL text and the value now travel to SQLite separately. SQLite parses the statement as code and treats `search` only as data, even when it contains quotes or SQL keywords.

Placeholders can represent values, not SQL structure. You cannot bind a table name, column name, or `ASC/DESC` direction. If the user can choose one of those, map the input to a small allowlist:

```
const sortColumns = { newest: 'created_at', title: 'title' }
const column = sortColumns[requestedSort] ?? 'created_at'
const sql = `SELECT * FROM notes ORDER BY ${column}`
```

Prepared statements prevent SQL injection at the query boundary, but they do not decide what a user may access. An authenticated user could still request another user's note through a perfectly safe prepared statement. Keep authorization checks and domain validation in place.

When the same statement runs repeatedly, reuse the prepared statement if your database library supports it. This can avoid parsing the same SQL over and over, but correctness and safety are the primary reasons to prepare it.

Try it: write a lookup by email using a placeholder, then pass an email containing a single quote. It should be handled as an ordinary value without changing the query.

Use SQLite from Node.js

Open SQLite from Node.js, configure lock waits, bind values, read rows, verify foreign keys, and close the connection.

Node.js now ships a SQLite driver in the standard library, so a script can use a real database with zero dependencies.

`node:sqlite` was added in Node.js 22.5. As of July 2026, Node marks it as a release-candidate API with stability 1.2. The example below needs a current supported Node release with the `timeout` option.

Create `app.js`:

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

const db = new DatabaseSync('notes.db', { timeout: 3000 })

db.exec('PRAGMA foreign_keys = ON')
const foreignKeys = db.prepare('PRAGMA foreign_keys').get()
console.log(foreignKeys)

db.exec(`
  CREATE TABLE IF NOT EXISTS notes (
    id INTEGER PRIMARY KEY,
    title TEXT NOT NULL
  ) STRICT
`)

const insert = db.prepare('INSERT INTO notes (title) VALUES (?)')
insert.run('Plan the week')

const notes = db.prepare(
  'SELECT id, title FROM notes ORDER BY id'
).all()
console.log(notes)

db.close()
```

Run it with `node app.js`:

```
[Object: null prototype] { foreign_keys: 1 }
[ [Object: null prototype] { id: 1, title: 'Plan the week' } ]
```

The `foreign_keys: 1` line proves enforcement is on for this connection. Current `node:sqlite` versions enable foreign keys by default, but setting and reading the pragma makes the application requirement explicit.

The pieces worth noticing

The `timeout` lets a short lock conflict wait for three seconds. Without it, a concurrent writer makes your statement fail immediately with a locked-database error instead of waiting its turn. Three seconds absorbs normal contention; if you still hit lock errors, some connection is holding a

transaction open too long.

`db.exec()` runs statements without results — pragmas, DDL. `db.prepare()` compiles a statement you can run many times with different values. The `?` placeholder is how values get in: `insert.run('Plan the week')` binds the string safely, with no string concatenation and no SQL injection surface.

Prepared statements give you three read shapes: `.get()` returns the first row, `.all()` returns every row as an array, and `.run()` executes a write and reports `{ changes, lastInsertRowid }`.

Call `db.close()` when the work is done. It releases the file locks and flushes state; a script that never closes can leave a WAL file waiting for a checkpoint.

Where the synchronous model fits

`DatabaseSync` runs every operation on the JavaScript thread. There are no promises to await, which is pleasant — but a query that takes 200 ms blocks your entire process for 200 ms.

It fits scripts, tests, command-line tools, and light application work. Long queries or heavy concurrent traffic block that thread, so choose another architecture when the workload needs it: worker threads, or a server database.

Use an in-memory database in tests

Give each test a fast, isolated SQLite database without leaving files behind.

Open `:memory:` instead of a file path. The database exists only for that connection and disappears when it closes.

That gives a test fast setup and isolation without temporary files:

```
const db = openDatabase(':memory:')
runMigrations(db)
seedMinimumData(db)
```

Create a fresh database for each test, or reset it deliberately between tests. Sharing one mutable database across a suite lets test order affect the result: one test can leave a row that makes the next test pass or fail.

Run the application's real migrations at setup rather than recreating tables with a test-only SQL string. This verifies that a new installation can reach the schema the application expects. If migration 7 forgets a column, an in-memory test should expose the mismatch.

An in-memory database belongs to a connection. Opening a second `:memory:` connection normally creates a different empty database. This surprises applications that use connection pools: the migration can run on one connection while the query runs on another. Use a single connection for the test, use SQLite's shared-memory URI deliberately, or use a temporary file when the production code requires multiple connections.

In-memory tests are useful, but they are not a perfect production simulation. They do not reproduce file permissions, disk-full failures, backup behavior, or the exact locking interactions of several processes. Keep a smaller integration-test layer that opens a real temporary database file for those concerns.

Also enable the same important connection settings as production, such as foreign-key enforcement:

```
PRAGMA foreign_keys = ON;
```

Try it: write one test that inserts an invalid foreign key and assert it fails. If it passes, your test connection is not enforcing the same invariant as production.

SQLite permissions live outside the database

Understand why SQLite has no user permissions: GRANT and REVOKE are not available because the database is a single file, so handle access in your app.

I covered user permissions in MySQL and PostgreSQL.

One thing to note about SQLite is that permissions management, using `GRANT` and `REVOKE`, is not available.

It's not available because **it's not possible**.

The reason is that an SQLite database is self-contained in a single file.

This is due to the SQLite architecture.

Anything with access to SQLite file can access anything inside the database.

There is no way to give permissions **at the database level**.

If your application needs to implement user permissions, you can do so at an application level, for example in your API server, but it's up to you.

If your app must need user permissions, you could also reconsider your DBMS choice and prefer PostgreSQL of MySQL/MariaDB instead.

Back up SQLite safely

Create and verify a consistent backup without copying one file from an active database.

If every connection is closed, you can copy the database file. While the database is active, use a SQLite-aware backup method.

The reason is timing. A plain `cp` reads the file from start to finish while writes may land in the middle, so the copy can contain half of one transaction. Do not copy only `notes.db` while writes continue. In WAL mode, committed data may still be in `notes.db-wal`, so the copy misses transactions that were committed minutes ago.

Two safe methods

From the SQLite shell, create a consistent backup with:

```
.backup notes-backup.db
```

`.backup` uses SQLite's online backup API. It copies the database page by page while coordinating with active connections, and the result is a valid snapshot even if writes happen during the copy. You can run it from a script too:

```
sqlite3 notes.db ".backup notes-backup.db"
```

Alternatively, use your library's SQLite backup API or `VACUUM INTO` when that behavior fits the application:

```
VACUUM INTO 'notes-backup.db';
```

`VACUUM INTO` writes a fresh, defragmented copy — it's a backup and a cleanup in one, and the

output file is often smaller than the original. The trade-off is that it rebuilds the database rather than copying pages, so it does more work on large databases.

Either way, back up to a different disk or machine. A backup file sitting next to the original dies with the same disk.

A backup you haven't tested is a guess

Open the backup separately and verify it:

```
sqlite3 notes-backup.db
```

```
PRAGMA integrity_check;
PRAGMA foreign_keys = ON;
PRAGMA foreign_key_check;
SELECT COUNT(*) FROM notes;
```

`integrity_check` should return `ok`. `foreign_key_check` should return no rows. The `COUNT(*)` should look plausible against what you know the database holds — a count of zero on a database you know is full means you backed up the wrong file.

A backup is useful only after you prove it restores. The realistic failure isn't a corrupt copy; it's a cron job that has been silently backing up an empty or stale file for months, discovered on the day you need it. Automate the verification queries alongside the backup itself, and alert when `integrity_check` says anything but `ok`.

Change the schema with migrations

Record schema changes in order so every environment can build the same database deliberately.

A migration is a versioned schema change. Keep each migration in source control and record which migrations ran.

Give migrations an immutable order, for example:

```
001_create_notes.sql
002_add_notes_owner.sql
003_index_notes_owner.sql
```

The application records completed versions in a migration table. At startup or deployment, it applies only the missing files in order. Never silently edit a migration that has already run in another environment. Add a new migration so an existing database and a fresh database follow the same history.

Separate the schema transition from application behavior when compatibility matters. For example, adding a required column safely may take several releases:

1. Add the column as nullable or with a safe default.
2. Deploy code that writes the new value.
3. Backfill old rows.
4. Start reading the new value.
5. Enforce the final constraint in a later migration.

SQLite supports operations such as renaming tables and columns and adding columns. More complex changes may require creating a replacement table, copying data, dropping the old table, and renaming the replacement. Preserve indexes, triggers, constraints, and foreign keys during that

process; copying only the visible columns is a common data-integrity bug.

Wrap compatible schema and data changes in a transaction so a failure does not leave a half-migrated database. Before a risky migration, make a backup and verify that it can be restored. A migration's rollback plan is often restoring that backup or deploying a forward fix, not automatically reversing every destructive statement.

Test two paths: building an empty database from migration 1, and upgrading a copy of realistic data from the currently deployed version. Measure long table rewrites too, because a migration that is correct on 20 rows may hold the only writer for too long on millions.

Try it: add a `slug` column to `notes`, backfill it, add a unique index, then run the complete migration sequence twice. The second run should have no pending work and should not damage data.

Build a small notes database

Build and verify a complete SQLite notes database with strict tables, relations, an index, and a restored backup.

Let's finish the course by building one database from an empty directory. Every step is something you've done in an earlier lesson; the point is doing them in order, the way a real project would.

Open `notes.db`, enable foreign keys, and create the schema:

```
sqlite3 notes.db
```

```
PRAGMA foreign_keys = ON;

CREATE TABLE notes (
  id INTEGER PRIMARY KEY,
  title TEXT NOT NULL,
  body TEXT,
  archived INTEGER NOT NULL DEFAULT 0
    CHECK (archived IN (0, 1)),
  created_at TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP
) STRICT;

CREATE TABLE tags (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL UNIQUE
) STRICT;

CREATE TABLE note_tags (
  note_id INTEGER NOT NULL REFERENCES notes(id) ON DELETE CASCADE,
  tag_id INTEGER NOT NULL REFERENCES tags(id) ON DELETE CASCADE,
  PRIMARY KEY (note_id, tag_id)
) STRICT;
```

Notice the schema carries its own rules: `STRICT` types, a `CHECK` that keeps `archived` boolean-shaped, a `UNIQUE` tag name, and cascading deletes so removing a note cleans up its links. None of this depends on application code behaving.

Insert related data as one transaction:

```
BEGIN IMMEDIATE;
INSERT INTO notes (title, body)
VALUES ('Plan the week', 'Pick the three important tasks');
INSERT INTO tags (name) VALUES ('planning');
INSERT INTO note_tags (note_id, tag_id)
SELECT notes.id, tags.id
FROM notes, tags
WHERE notes.title = 'Plan the week'
  AND tags.name = 'planning';
COMMIT;
```

The three inserts describe one fact — a tagged note — so they belong in one transaction. If the third insert failed, a `ROLLBACK` would leave no half-linked data behind. `BEGIN IMMEDIATE` claims the write lock up front, so a competing writer surfaces now rather than mid-transaction.

Inspect a real lookup before adding an index:

```
EXPLAIN QUERY PLAN
SELECT id, title FROM notes WHERE title = 'Plan the week';
```

You should see `SCAN notes` — a full table walk. Now add the index and run the plan again:

```
CREATE INDEX notes_title_idx ON notes(title);
```

The second plan should report a search using `notes_title_idx` instead of a table scan. That before-and-after check is the habit that keeps you from collecting indexes that do nothing.

Create a live backup:

```
.backup notes-backup.db
```

Leave with `.quit`, then open the restored copy:

```
sqlite3 notes-backup.db
```

This is the step most people skip: actually opening the backup. Finish with these checks:

```
PRAGMA foreign_keys = ON;
PRAGMA integrity_check;
PRAGMA foreign_key_check;
SELECT notes.title, tags.name
FROM notes
JOIN note_tags ON note_tags.note_id = notes.id
JOIN tags ON tags.id = note_tags.tag_id;
```

You are done when `integrity_check` returns `ok`, `foreign_key_check` returns no rows, and the final query returns `Plan the week|planning`.

If any check fails, the backup is the suspect, not the original — repeat the `.backup` while the source database is quiet and verify again. And if the join returns nothing, re-run it against `notes.db`: an empty result there means the transaction never committed, an empty result only in the copy means you backed up before the commit.

That's the full loop: schema with rules, transactional writes, evidence-based indexing, and a backup you proved restores. Every database you ship deserves the same treatment.

Check your understanding: applications and operations

Check prepared statements, Node.js access, in-memory tests, permissions, live backups, migrations, and SQLite limits.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/sqlite/.