

Forum:[Hack Forums](#)

Thread: [SQLite in production: the settings that keep it fast and safe](#)

iconia - August 19, 2026, 12:47 pm

SQLite in production: the settings that keep it fast and safe

SQLite is a real production database — it runs phones, browsers, and plenty of live services. But its defaults are tuned for "works everywhere," not "fast and safe under load." A handful of one-line settings change everything. Here are the ones that matter, why, and the exact pragmas to paste.

Part 1 of 2 — the pragmas that fix concurrency, the dreaded "database is locked", durability, and a silent data-integrity trap. Part 2 (space reclamation, backups done right, isolation, and reading query plans) is in the reply below.

Quote:If you've ever seen "database is locked" under load and concluded "SQLite can't handle concurrency" — you were one pragma away from it being fine. Read on.

1. WAL mode — the single most important change

By default SQLite uses a rollback journal: a writer takes an exclusive lock and readers are blocked while anyone writes. Under any real concurrency that serializes everything.

Write-Ahead Logging (WAL) flips it: writes append to a separate -wal file, so readers and a writer run at the same time. Readers see a consistent snapshot; the writer doesn't block them.

Code `PRAGMA journal_mode = WAL;`

Set it once per database — it's persistent, stored in the file header, not per-connection. This alone turns "SQLite is slow under load" into "SQLite is fine."

Code <code>ROLLBACK JOURNAL</code> (default)	WAL
writer ===== locks the db	writer -> appends to -wal
readers blocked	readers -> read freely, concurrently
one at a time	reads + 1 write in parallel

2. `busy_timeout` — the actual fix for "database is locked"

SQLite allows one writer at a time. If a second write arrives while the first holds the lock, by default SQLite returns `SQLITE_BUSY` ("database is locked") immediately instead of waiting. That instant failure is what most people hit.

Tell it to wait instead:

Code `PRAGMA busy_timeout = 5000;` -- wait up to 5000 ms for the lock

Now a competing write blocks for up to 5 seconds (plenty for a fast write to finish) instead of erroring out. Set this on every connection. Between WAL + `busy_timeout`, the vast majority of "SQLite can't do concurrency" problems vanish.

3. `synchronous = NORMAL` — safe AND fast (with WAL)

The default `synchronous = FULL` fsyncs on every commit — safe, but slow. In WAL mode, you can drop to `NORMAL` and still be crash-safe (you can only lose the very last transaction on an OS/power

crash, never corruption):

```
Code PRAGMA synchronous = NORMAL;
```

Big write throughput win, no corruption risk. (Keep FULL only if losing the last committed transaction on a power cut is unacceptable — most apps are fine with NORMAL.)

4. `foreign_keys = ON` — it's OFF by default (yes, really)

This one silently corrupts data. SQLite parses FOREIGN KEY constraints but does not enforce them unless you turn them on — and it's off by default, per connection. So your "referential integrity" isn't being checked at all.

```
Code PRAGMA foreign_keys = ON;      -- every connection, every time
```

If you've been running without this, run a check before enabling — you may already have orphaned rows:

```
Code PRAGMA foreign_key_check;      -- lists violating rows; empty = clean
```

5. Put it together — the connection setup block

WAL and (usually) `auto_vacuum` are persistent file settings; the rest are per-connection and must be set every time you open the database. A typical setup:

Code import sqlite3

```
def connect(path):
```

```
    conn = sqlite3.connect(path, timeout=5.0)
```

```
    conn.execute("PRAGMA journal_mode = WAL;")    # persistent, but cheap to re-assert
```

```
    conn.execute("PRAGMA busy_timeout = 5000;")    # per-connection
```

```
    conn.execute("PRAGMA synchronous = NORMAL;")    # per-connection
```

```
    conn.execute("PRAGMA foreign_keys = ON;")    # per-connection
```

```
    return conn
```

Four lines. They're the difference between "SQLite fell over under load" and "SQLite quietly handled it."

-> Continued in Part 2 below: reclaiming space (and the WAL-file-keeps-growing gotcha), backing up a live database the RIGHT way, per-tenant isolation, and using EXPLAIN QUERY PLAN to confirm your indexes actually work.

Part 2 of 2 — space, backups, isolation, and query plans

(Continued from Part 1, where we set WAL, `busy_timeout`, `synchronous`, and `foreign_keys`. Those make it fast and safe under load. These keep it that way over time — and keep your data recoverable.)

6. Reclaiming space — and the WAL gotcha

Deleting rows does not shrink the database file — SQLite keeps the freed pages to reuse. To actually return space to the OS:

```
Code VACUUM;                        -- rewrites the db compactly, frees pages to disk
```

VACUUM rewrites the whole file, so run it on a schedule (or after big deletes), not on every request. And measure size honestly: in WAL mode the `-wal` file holds committed pages until a checkpoint folds them back into the main file. If you check size without checkpointing, you're lying to yourself:

Code `PRAGMA wal_checkpoint(TRUNCATE);` -- fold -wal back in and shrink it to 0
`VACUUM;` -- then compact the main file

If your -wal file grows without bound, you have long-lived read transactions preventing checkpoints — close your read connections promptly.

7. Backups — you cannot just copy the file

The #1 SQLite data-loss mistake: `cp app.db backup.db` while the app is running. Mid-write, you copy a torn file (and you miss the -wal), and your "backup" is corrupt. Copying the three files (db, -wal, -shm) out of sync is just as bad.

Do it one of these safe ways instead:

Code -- A) atomic, consistent snapshot to a new file (simplest):
`VACUUM INTO 'backup.db';`

-- B) the online backup API (safe even under active writes), from Python:

```
import sqlite3
src = sqlite3.connect('app.db')
dst = sqlite3.connect('backup.db')
with dst:
    src.backup(dst)          # consistent, page-by-page, live-safe
dst.close(); src.close()
```

Both produce a clean, single-file backup you can copy anywhere. `VACUUM INTO` also compacts as it goes — backup + defrag in one.

8. Per-database isolation — a clean multi-tenant pattern

SQLite databases are just files, and opening one is cheap. For strong isolation (per tenant, per project, per test), one database file each beats a shared `tenant_id` column on everything:

Hard isolation — no query can ever leak across tenants; there's no shared table to get a `WHERE` clause wrong on.

Trivial per-tenant backup/export/delete — it's one file. Move it, ship it, drop it.

Smaller indexes — each file only holds one tenant's data, so lookups stay fast.

The tradeoff: cross-tenant queries need `ATTACH DATABASE` or app-side fan-out, so it fits when tenants are naturally separate (which is most SaaS). For a few-hundred-tenant service, a file-per-tenant directory is clean and fast.

9. EXPLAIN QUERY PLAN — confirm the index is actually used

Adding an index doesn't guarantee your query uses it. Ask SQLite directly:

Code `EXPLAIN QUERY PLAN`

`SELECT * FROM events WHERE user_id = ? AND created_at > ?;`SCAN events -> full table scan.
Bad. You're missing (or not using) an index.

`SEARCH events USING INDEX idx_user_time` -> good, it's using the index.

If you see `SCAN` on a hot query, add a covering index for the `WHERE + ORDER BY` columns and check again. Ten seconds, and it's the difference between a 1 ms and a 400 ms query at scale.

10. The paste-and-go checklist

`PRAGMA journal_mode=WAL` once per db (concurrency).

PRAGMA busy_timeout=5000 every connection (kills "database is locked").
PRAGMA synchronous=NORMAL every connection (fast + crash-safe under WAL).
PRAGMA foreign_keys=ON every connection (it's OFF by default!).
VACUUM on a schedule; wal_checkpoint(TRUNCATE) before you trust a size.
Back up with VACUUM INTO or the .backup API — never cp a live db.
EXPLAIN QUERY PLAN your hot queries — SEARCH good, SCAN bad.

Wrap-up: SQLite isn't a toy you outgrow — it's a serious engine with conservative defaults. WAL + busy_timeout make it handle real concurrency, NORMAL makes it fast, foreign_keys keeps your data honest, and safe backups keep it recoverable. Four lines on connect and a couple of habits, and it'll carry a production workload further than most people expect.

Happy to go deeper on ATTACH-based cross-tenant queries or WAL checkpoint tuning if there's interest.

SUI-3183372 - August 19, 2026, 1:52 pm
Thanks Claude

iconia - August 19, 2026, 1:53 pm
(Aug 19, 2026 01:52 PM)SUI-3183372 Wrote: [\[CITE\]](#) Thanks Claude
i quote every post with "i had ai help me write this" and the one post i didnt, gets one of these.
noted.

itcorp - August 29, 2026, 7:55 am
This is a great setup for production, but if you have long-lived transactions or forget to close connections, the wal file can bloat to gigabytes while the main DB stays small, and if the power fails, you might lose the very last transaction that hadn't been fully synced to disk... So it's fast and practical, but these trade offs can have serious consequences if you're not careful.

iconia - September 14, 2026, 10:37 am
(Aug 29, 2026 07:55 AM)itcorp Wrote: [\[CITE\]](#) This is a great setup for production, but if you have long-lived transactions or forget to close connections, the wal file can bloat to gigabytes while the main DB stays small, and if the power fails, you might lose the very last transaction that hadn't been fully synced to disk... So it's fast and practical, but these trade offs can have serious consequences if you're not careful.
super useful info thanks for the feedback. very true