SQLite: the write queue and locking

How the SQLite driver of Exponential 6.0.15 queues concurrent writers with a lock file and BEGIN IMMEDIATE, how long a transaction waits (SQLiteTransactionWait) and how that relates to Velocity's and PHP-FPM's request...

Source file: `doc/eng-us/help/database-administrators/sqlite-write-queue-and-locking.md` Last changed 2026-11-05
https://doc.exponential.earth/learninminutes/exphelp/viewer/help/database-administrators/sqlite-write-queue-and-locking.md

This page is for database administrators who run Exponential on SQLite and need to understand what happens when two editors publish at the same moment. It answers: why plain SQLite transactions fail with "database is locked", what the driver does instead since 6.0.15, how long a write waits, what a timeout looks like and how to find its cause. Sources: Queued writes and SQLite: transactions queue for the write lock.

Contents

1. The problem
2. What the driver does
3. How long a transaction waits
4. What a timeout looks like
5. When timeouts repeat
6. Forked processes and other users
7. Measured behavior
8. Checklist

The problem

A plain `BEGIN` in SQLite is deferred: the transaction reads first and asks for the write lock only at its first write. If another connection committed in between, the snapshot it read is stale, and SQLite answers `SQLITE_BUSY` immediately, without waiting, because waiting cannot make the snapshot current. Before 6.0.15 a publish could fail with "database is locked" however long the busy timeout was.

What the driver does

`eZSQLite3DB::beginQuery()` starts every transaction like this:

1. It opens, once per connection and again in every forked process, a lock file next to the database: `<database file>.writer-lock`, for example `var/storage/sqlite3/exponential.db.writer-lock`.
2. It takes an exclusive `flock()` on that file, retrying every 1 to 15 ms; writers that have waited more than a second try more often, which serves them roughly in arrival order. While it waits it holds a shared `flock()` on a second file, `<database file>.writer-queue`, which tells the others that a writer is waiting. A writer that finds the lock free while others wait

leaves it to them for 30 ms before its own first try. Without that, a loop that commits and begins again at once (the `indexcontent` cronjob part, a content job's batches) took the lock back within a fraction of a millisecond of every commit and kept it for minutes, and a page's save behind it failed with "database is busy" after its whole wait.

3. With the lock held it runs `BEGIN IMMEDIATE`, which takes SQLite's write lock at the start, where the busy timeout applies. Writers outside the queue (a single statement, another program) are waited for by SQLite's busy handler within the time left.
4. At `COMMIT` or `ROLLBACK` it releases the lock file. If the process dies, the operating system releases it, so a crash never blocks the queue.

Waiting at the start is safe: before `BEGIN IMMEDIATE` succeeds the transaction has written nothing, and afterwards nothing inside it can lose a lock race.

Every process that writes the database uses the same `.writer-lock`: PHP-FPM workers, Velocity workers, cronjobs and console scripts queue against each other.

How long a transaction waits

Up to `[DatabaseSettings] SQLiteTransactionWait` seconds (default `25`; values below 1 are raised to 1). Keep it below the web server's request timeout, so a waiting request fails with a clear message instead of being killed:

Server	Request timeout	Note
Velocity	`requestTimeout` (`Q.webserver.requestTimeout`), 30 s unless changed	the client gets a 504 when it is exceeded
PHP-FPM	the pool's `request_terminate_timeout` (off unless set)	`max_execution_time` does not count waiting time on Linux

Releases before 6.0.15 shipped `60`; if your override still sets `60`, lower it to `25`. To wait longer, raise the request timeout and `SQLiteTransactionWait` together, the wait always a few seconds below the timeout.

Shell

```
php bin/php/console exp:ini get site.ini/DatabaseSettings/SQLiteTransactionWait --allow-root-user
```

On the command line (cronjobs, imports, scripts) the wait is `SQLiteCLITransactionWait` instead, default `300` seconds: nobody waits for a page there, and a cronjob waits for a long import rather than stopping with "A database transaction in Exponential failed".

Single statements

A write outside a transaction (an `INSERT` of a link, an `UPDATE` of a cronjob) takes its turn on the same `.writer-lock` and waits up to `SQLiteBusyTimeout` milliseconds in a web request (default `5000`) or `SQLiteCLIBusyTimeout` on the command line (default `30000`). A lock that SQLite reports at once is tried again within that time. Reads never queue.

What a timeout looks like

In `var/log/error.log` and on the error page:

```
database is busy: the transaction could not start within 25 s (waited 25.0 s, [DatabaseSettings]
SQLiteTransactionWait), another write held the lock all that time; nothing was written
database is busy: the statement waited 5.0 s for another write to end (limit 5 s, [DatabaseSettin
gs] SQLiteBusyTimeout) and was not run; nothing was written
```

Any other failure at the start reads `the transaction could not start: <SQLite's message>; nothing was written`. In both cases nothing was written; retry the action.

When timeouts repeat

One write held the lock for the whole wait. Look at what was running at that time in `var/log/error.log` and in the cron log:

- a `VACUUM` (it holds the write lock for its whole run);
- a large import or a script that opened a transaction and then waited for something else;
- a `sqlite3` shell left open inside `BEGIN`.

Count the timeouts with `grep -c "database is busy" var/log/error.log`. Raising `SQLiteTransactionWait` helps only when the holder was legitimately long. If the queue is regularly longer than the wait during normal editing, a server database with row-level locking serves the editors better (Choosing an engine).

Forked processes and other users

`flock()` belongs to an open file, so a forked process must not reuse its parent's handle. The driver remembers the process id that opened the lock file and opens a new one in each fork; Velocity workers and the asynchronous publisher's children queue correctly. The driver opens `.writer-lock` for writing and, failing that, read-only: `flock()` works on a read-only handle, so a lock file created by another user still queues correctly.

Measured behavior

From the change's own tests on the development server: 192 publishes from 64 processes at once all completed, where the earlier start lost 6 of them; 48 transactions of 300 ms each, started together, finished in 14.7 seconds, about the 14.4 seconds they take one after another. The queue costs almost nothing beyond the waiting itself.

Checklist

- ☐ `SQLiteTransactionWait` is below Velocity's `requestTimeout` and PHP-FPM's `request_terminate_timeout`.
- ☐ No override still sets the old value `60`.
- ☐ `VACUUM` and large imports run in maintenance mode.
- ☐ `grep -c "database is busy" var/log/error.log` is 0 or explained.

Related pages

- SQLite 3 for database administrators
- SQLite maintenance
- Troubleshooting database errors
- Velocity workers and pools
- Free documentation: this page is free to copy, translate and improve under the GNU FDL 1.3.

> **Become an Exponential Partner**
>
> Partners share their information, provide funding, contribute back and develop the products with us, and that is how Exponential grows. A partner website and dashboard are planned for mid-2027: please become a partner today to speed up our growth rate and land new features faster.
>
> Partners: start here | How to apply

Exponential Copyright © 1998-2026 Exponential Foundation and 7x
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".
This documentation is free documentation, licensed under the GNU Free Documentation License 1.3 or later:
https://www.gnu.org/licenses/fdl-1.3.html.