Write-Ahead Logging

▶ Table Of Contents

1. Overview

The default method by which SQLite implements atomic commit and rollback is a rollback journal. Beginning with version 3.7.0 (2010-07-21), a new "Write-Ahead Log" option (hereafter referred to as "WAL") is available.

There are advantages and disadvantages to using WAL instead of a rollback journal. Advantages include:

1. WAL is significantly faster in most scenarios.
2. WAL provides more concurrency as readers do not block writers and a writer does not block readers. Reading and writing can proceed concurrently.
3. Disk I/O operations tends to be more sequential using WAL.
4. WAL uses many fewer fsync() operations and is thus less vulnerable to problems on systems where the fsync() system call is broken.

But there are also disadvantages:

1. All processes using a database must be on the same host computer; WAL does not work over a network filesystem. This is because WAL requires all processes to share a small amount of memory and processes on separate host machines obviously cannot share memory with each other.
2. Transactions that involve changes against multiple ATTACHed databases are atomic for each individual database, but are not atomic across all databases as a set.
3. It is not possible to change the page_size after entering WAL mode, either on an empty database or by using VACUUM or by restoring from a backup using the backup API. You must be in a rollback journal mode to change the page size.
4. ~~It is not possible to open read-only WAL databases. The opening process must have write privileges for "-shm" wal-index shared memory file associated with the database, if that file exists, or else write access on the directory containing the database file if the "-shm" file does not exist.~~ Beginning with version 3.22.0 (2018-01-22), a read-only WAL-mode database file can be opened if the `-shm` and `-wal` files already exist or those files can be created or the database is immutable.
5. WAL might be very slightly slower (perhaps 1% or 2% slower) than the traditional rollback-journal approach in applications that do mostly reads and seldom write.
6. There is an additional quasi-persistent "`-wal`" file and "`-shm`" shared memory file associated with each database, which can make SQLite less appealing for use as an application file-format.

7. There is the extra operation of checkpointing which, though automatic by default, is still something that application developers need to be mindful of.
8. ~~WAL works best with smaller transactions. WAL does not work well for very large transactions. For transactions larger than about 100 megabytes, traditional rollback journal modes will likely be faster. For transactions in excess of a gigabyte, WAL mode may fail with an I/O or disk-full error. It is recommended that one of the rollback journal modes be used for transactions larger than a few dozen megabytes.~~ Beginning with version 3.11.0 (2016-02-15), WAL mode works as efficiently with large transactions as does rollback mode.

2. How WAL Works

The traditional rollback journal works by writing a copy of the original unchanged database content into a separate rollback journal file and then writing changes directly into the database file. In the event of a crash or ROLLBACK, the original content contained in the rollback journal is played back into the database file to revert the database file to its original state. The COMMIT occurs when the rollback journal is deleted.

The WAL approach inverts this. The original content is preserved in the database file and the changes are appended into a separate WAL file. A COMMIT occurs when a special record indicating a commit is appended to the WAL. Thus a COMMIT can happen without ever writing to the original database, which allows readers to continue operating from the original unaltered database while changes are simultaneously being committed into the WAL. Multiple transactions can be appended to the end of a single WAL file.

2.1. Checkpointing

Of course, one wants to eventually transfer all the transactions that are appended in the WAL file back into the original database. Moving the WAL file transactions back into the database is called a "*checkpoint*".

Another way to think about the difference between rollback and write-ahead log is that in the rollback-journal approach, there are two primitive operations, reading and writing, whereas with a write-ahead log there are now three primitive operations: reading, writing, and checkpointing.

By default, SQLite does a checkpoint automatically when the WAL file reaches a threshold size of 1000 pages. (The SQLITE_DEFAULT_WAL_AUTOCHECKPOINT compile-time option can be used to specify a different default.) Applications using WAL do not have to do anything in order for these checkpoints to occur. But if they want to, applications can adjust the automatic checkpoint threshold. Or they can turn off the automatic checkpoints and run checkpoints during idle moments or in a separate thread or process.

2.2. Concurrency

When a read operation begins on a WAL-mode database, it first remembers the location of the last valid commit record in the WAL. Call this point the "end mark". Because the WAL can be growing and adding new commit records while various readers connect to the database, each reader can potentially have its own end mark. But for any particular reader, the end mark is unchanged for the duration of the transaction, thus ensuring that a single read transaction only sees the database content as it existed at a single point in time.

When a reader needs a page of content, it first checks the WAL to see if that page appears there, and if so it pulls in the last copy of the page that occurs in the WAL prior to the reader's end mark. If no

copy of the page exists in the WAL prior to the reader's end mark, then the page is read from the original database file. Readers can exist in separate processes, so to avoid forcing every reader to scan the entire WAL looking for pages (the WAL file can grow to multiple megabytes, depending on how often checkpoints are run), a data structure called the "wal-index" is maintained in shared memory which helps readers locate pages in the WAL quickly and with a minimum of I/O. The wal-index greatly improves the performance of readers, but the use of shared memory means that all readers must exist on the same machine. This is why the write-ahead log implementation will not work on a network filesystem.

Writers merely append new content to the end of the WAL file. Because writers do nothing that would interfere with the actions of readers, writers and readers can run at the same time. However, since there is only one WAL file, there can only be one writer at a time.

A checkpoint operation takes content from the WAL file and transfers it back into the original database file. A checkpoint can run concurrently with readers, however the checkpoint must stop when it reaches a page in the WAL that is past the end mark of any current reader. The checkpoint has to stop at that point because otherwise it might overwrite part of the database file that the reader is actively using. The checkpoint remembers (in the wal-index) how far it got and will resume transferring content from the WAL to the database from where it left off on the next invocation.

Thus a long-running read transaction can prevent a checkpointer from making progress. But presumably every read transaction will eventually end and the checkpointer will be able to continue.

Whenever a write operation occurs, the writer checks how much progress the checkpointer has made, and if the entire WAL has been transferred into the database and synced and if no readers are making use of the WAL, then the writer will rewind the WAL back to the beginning and start putting new transactions at the beginning of the WAL. This mechanism prevents a WAL file from growing without bound.

2.3. Performance Considerations

Write transactions are very fast since they only involve writing the content once (versus twice for rollback-journal transactions) and because the writes are all sequential. Further, syncing the content to the disk is not required, as long as the application is willing to sacrifice durability following a power loss or hard reboot. (Writers sync the WAL on every transaction commit if PRAGMA synchronous is set to FULL but omit this sync if PRAGMA synchronous is set to NORMAL.)

On the other hand, read performance deteriorates as the WAL file grows in size since each reader must check the WAL file for the content and the time needed to check the WAL file is proportional to the size of the WAL file. The wal-index helps find content in the WAL file much faster, but performance still falls off with increasing WAL file size. Hence, to maintain good read performance it is important to keep the WAL file size down by running checkpoints at regular intervals.

Checkpointing does require sync operations in order to avoid the possibility of database corruption following a power loss or hard reboot. The WAL must be synced to persistent storage prior to moving content from the WAL into the database and the database file must be synced prior to resetting the WAL. Checkpoint also requires more seeking. The checkpointer makes an effort to do as many sequential page writes to the database as it can (the pages are transferred from WAL to database in ascending order) but even then there will typically be many seek operations interspersed among the page writes. These factors combine to make checkpoints slower than write transactions.

The default strategy is to allow successive write transactions to grow the WAL until the WAL becomes about 1000 pages in size, then to run a checkpoint operation for each subsequent COMMIT until the WAL is reset to be smaller than 1000 pages. By default, the checkpoint will be run automatically by the

same thread that does the COMMIT that pushes the WAL over its size limit. This has the effect of causing most COMMIT operations to be very fast but an occasional COMMIT (those that trigger a checkpoint) to be much slower. If that effect is undesirable, then the application can disable automatic checkpointing and run the periodic checkpoints in a separate thread, or separate process. (Links to commands and interfaces to accomplish this are shown below.)

Note that with PRAGMA synchronous set to NORMAL, the checkpoint is the only operation to issue an I/O barrier or sync operation (fsync() on unix or FlushFileBuffers() on windows). If an application therefore runs checkpoint in a separate thread or process, the main thread or process that is doing database queries and updates will never block on a sync operation. This helps to prevent "latch-up" in applications running on a busy disk drive. The downside to this configuration is that transactions are no longer durable and might rollback following a power failure or hard reset.

Notice too that there is a tradeoff between average read performance and average write performance. To maximize the read performance, one wants to keep the WAL as small as possible and hence run checkpoints frequently, perhaps as often as every COMMIT. To maximize write performance, one wants to amortize the cost of each checkpoint over as many writes as possible, meaning that one wants to run checkpoints infrequently and let the WAL grow as large as possible before each checkpoint. The decision of how often to run checkpoints may therefore vary from one application to another depending on the relative read and write performance requirements of the application. The default strategy is to run a checkpoint once the WAL reaches 1000 pages and this strategy seems to work well in test applications on workstations, but other strategies might work better on different platforms or for different workloads.

3. Activating And Configuring WAL Mode

An SQLite database connection defaults to journal_mode=DELETE. To convert to WAL mode, use the following pragma:

```
PRAGMA journal_mode=WAL;
```

The journal_mode pragma returns a string which is the new journal mode. On success, the pragma will return the string "`wal`". If the conversion to WAL could not be completed (for example, if the VFS does not support the necessary shared-memory primitives) then the journaling mode will be unchanged and the string returned from the primitive will be the prior journaling mode (for example "`delete`").

3.1. Automatic Checkpoint

By default, SQLite will automatically checkpoint whenever a COMMIT occurs that causes the WAL file to be 1000 pages or more in size, or when the last database connection on a database file closes. The default configuration is intended to work well for most applications. But programs that want more control can force a checkpoint using the wal_checkpoint pragma or by calling the sqlite3_wal_checkpoint() C interface. The automatic checkpoint threshold can be changed or automatic checkpointing can be completely disabled using the wal_autocheckpoint pragma or by calling the sqlite3_wal_autocheckpoint() C interface. A program can also use sqlite3_wal_hook() to register a callback to be invoked whenever any transaction commits to the WAL. This callback can then invoke sqlite3_wal_checkpoint() or sqlite3_wal_checkpoint_v2() based on whatever criteria it thinks is appropriate. (The automatic checkpoint mechanism is implemented as a simple wrapper around sqlite3_wal_hook().)

3.2. Application-Initiated Checkpoints

An application can initiate a checkpoint using any writable database connection on the database simply by invoking sqlite3_wal_checkpoint() or sqlite3_wal_checkpoint_v2(). There are three subtypes of checkpoints that vary in their aggressiveness: PASSIVE, FULL, and RESTART. The default checkpoint style is PASSIVE, which does as much work as it can without interfering with other database connections, and which might not run to completion if there are concurrent readers or writers. All checkpoints initiated by sqlite3_wal_checkpoint() and by the automatic checkpoint mechanism are PASSIVE. FULL and RESTART checkpoints try harder to run the checkpoint to completion and can only be initiated by a call to sqlite3_wal_checkpoint_v2(). See the sqlite3_wal_checkpoint_v2() documentation for additional information on FULL and RESET checkpoints.

3.3. Persistence of WAL mode

Unlike the other journaling modes, PRAGMA journal_mode=WAL is persistent. If a process sets WAL mode, then closes and reopens the database, the database will come back in WAL mode. In contrast, if a process sets (for example) PRAGMA journal_mode=TRUNCATE and then closes and reopens the database will come back up in the default rollback mode of DELETE rather than the previous TRUNCATE setting.

The persistence of WAL mode means that applications can be converted to using SQLite in WAL mode without making any changes to the application itself. One has merely to run "`PRAGMA journal_mode=WAL;`" on the database file(s) using the command-line shell or other utility, then restart the application.

The WAL journal mode will be set on all connections to the same database file if it is set on any one connection.

4. The WAL File

While a database connection is open on a WAL-mode database, SQLite maintains an extra journal file called a "Write Ahead Log" or "WAL File". The name of this file on disk is usually the name of the database file with an extra "`-wal`" suffix, though different naming rules may apply if SQLite is compiled with SQLITE_ENABLE_8_3_NAMES.

The WAL file exists for as long as any database connection has the database open. Usually, the WAL file is deleted automatically when the last connection to the database closes. However, if the last process to have the database open exits without cleanly shutting down the database connection, or if the SQLITE_FCNTL_PERSIST_WAL file control is used, then the WAL file might be retained on disk after all connections to the database have been closed. The WAL file is part of the persistent state of the database and should be kept with the database if the database is copied or moved. If a database file is separated from its WAL file, then transactions that were previously committed to the database might be lost, or the database file might become corrupted. The only safe way to remove a WAL file is to open the database file using one of the sqlite3_open() interfaces then immediately close the database using sqlite3_close().

The WAL file format is precisely defined and is cross-platform.

5. Read-Only Databases

Older versions of SQLite could not read a WAL-mode database that was read-only. In other words, write access was required in order to read a WAL-mode database. This constraint was relaxed

beginning with SQLite version 3.22.0 (2018-01-22).

On newer versions of SQLite, a WAL-mode database on read-only media, or a WAL-mode database that lacks write permission, can still be read as long as one or more of the following conditions are met:

1. The `-shm` and `-wal` files already exist and are readable.
2. There is write permission on the directory containing the database so that the `-shm` and `-wal` files can be created.
3. The database connection is opened using the immutable query parameter.

Even though it is possible to open a read-only WAL-mode database, it is good practice to convert the database to PRAGMA journal_mode=DELETE prior to burning an SQLite database image onto read-only media.

6. Avoiding Excessively Large WAL Files

In normal cases, new content is appended to the WAL file until the WAL file accumulates about 1000 pages (and is thus about 4MB in size) at which point a checkpoint is automatically run and the WAL file is recycled. The checkpoint does not normally truncate the WAL file (unless the journal_size_limit pragma is set). Instead, it merely causes SQLite to start overwriting the WAL file from the beginning. This is done because it is normally faster to overwrite an existing file than to append. When the last connection to a database closes, that connection does one last checkpoint and then deletes the WAL and its associated shared-memory file, to clean up the disk.

So in the vast majority of cases, applications need not worry about the WAL file at all. SQLite will automatically take care of it. But it is possible to get SQLite into a state where the WAL file will grow without bound, causing excess disk space usage and slow query speeds. The following bullets enumerate some of the ways that this can happen and how to avoid them.

- **Disabling the automatic checkpoint mechanism.** In its default configuration, SQLite will checkpoint the WAL file at the conclusion of any transaction when the WAL file is more than 1000 pages long. However, compile-time and run-time options exist that can disable or defer this automatic checkpoint. If an application disables the automatic checkpoint, then there is nothing to prevent the WAL file from growing excessively.

- **Checkpoint starvation.** A checkpoint is only able to run to completion, and reset the WAL file, if there are no other database connections using the WAL file. If another connection has a read transaction open, then the checkpoint cannot reset the WAL file because doing so might delete content out from under the reader. The checkpoint will do as much work as it can without upsetting the reader, but it cannot run to completion. The checkpoint will start up again where it left off after the next write transaction. This repeats until some checkpoint is able to complete.

  However, if a database has many concurrent overlapping readers and there is always at least one active reader, then no checkpoints will be able to complete and hence the WAL file will grow without bound.

  This scenario can be avoided by ensuring that there are "reader gaps": times when no processes are reading from the database and that checkpoints are attempted during those times. In applications with many concurrent readers, one might also consider running manual checkpoints with the SQLITE_CHECKPOINT_RESTART or SQLITE_CHECKPOINT_TRUNCATE option which will ensure that the checkpoint runs to completion before returning. The disadvantage of using SQLITE_CHECKPOINT_RESTART and SQLITE_CHECKPOINT_TRUNCATE is that readers might block while the checkpoint is running.

- **Very large write transactions.** A checkpoint can only complete when no other transactions are running, which means the WAL file cannot be reset in the middle of a write transaction. So a large change to a large database might result in a large WAL file. The WAL file will be checkpointed once the write transaction completes (assuming there are no other readers blocking it) but in the meantime, the file can grow very big.

  As of SQLite version 3.11.0 (2016-02-15), the WAL file for a single transaction should be proportional in size to the transaction itself. Pages that are changed by the transaction should only be written into the WAL file once. However, with older versions of SQLite, the same page might be written into the WAL file multiple times if the transaction grows larger than the page cache.

7. Implementation Of Shared-Memory For The WAL-Index

The wal-index is implemented using an ordinary file that is mmapped for robustness. Early (pre-release) implementations of WAL mode stored the wal-index in volatile shared-memory, such as files created in /dev/shm on Linux or /tmp on other unix systems. The problem with that approach is that processes with a different root directory (changed via chroot) will see different files and hence use different shared memory areas, leading to database corruption. Other methods for creating nameless shared memory blocks are not portable across the various flavors of unix. And we could not find any method to create nameless shared memory blocks on windows. The only way we have found to guarantee that all processes accessing the same database file use the same shared memory is to create the shared memory by mmapping a file in the same directory as the database itself.

Using an ordinary disk file to provide shared memory has the disadvantage that it might actually do unnecessary disk I/O by writing the shared memory to disk. However, the developers do not think this is a major concern since the wal-index rarely exceeds 32 KiB in size and is never synced. Furthermore, the wal-index backing file is deleted when the last database connection disconnects, which often prevents any real disk I/O from ever happening.

Specialized applications for which the default implementation of shared memory is unacceptable can devise alternative methods via a custom VFS. For example, if it is known that a particular database will only be accessed by threads within a single process, the wal-index can be implemented using heap memory instead of true shared memory.

8. Use of WAL Without Shared-Memory

Beginning in SQLite version 3.7.4 (2010-12-07), WAL databases can be created, read, and written even if shared memory is unavailable as long as the locking_mode is set to EXCLUSIVE before the first attempted access. In other words, a process can interact with a WAL database without using shared memory if that process is guaranteed to be the only process accessing the database. This feature allows WAL databases to be created, read, and written by legacy VFSes that lack the "version 2" shared-memory methods xShmMap, xShmLock, xShmBarrier, and xShmUnmap on the sqlite3_io_methods object.

If EXCLUSIVE locking mode is set prior to the first WAL-mode database access, then SQLite never attempts to call any of the shared-memory methods and hence no shared-memory wal-index is ever created. In that case, the database connection remains in EXCLUSIVE mode as long as the journal mode is WAL; attempts to change the locking mode using "`PRAGMA locking_mode=NORMAL;`" are no-ops. The only way to change out of EXCLUSIVE locking mode is to first change out of WAL journal mode.

If NORMAL locking mode is in effect for the first WAL-mode database access, then the shared-memory wal-index is created. This means that the underlying VFS must support the "version 2" shared-memory. If the VFS does not support shared-memory methods, then the attempt to open a database that is already in WAL mode, or the attempt to convert a database into WAL mode, will fail. As long as exactly one connection is using a shared-memory wal-index, the locking mode can be changed freely between NORMAL and EXCLUSIVE. It is only when the shared-memory wal-index is omitted, when the locking mode is EXCLUSIVE prior to the first WAL-mode database access, that the locking mode is stuck in EXCLUSIVE.

9. Sometimes Queries Return SQLITE_BUSY In WAL Mode

The second advantage of WAL-mode is that writers do not block readers and readers do not block writers. This is mostly true. But there are some obscure cases where a query against a WAL-mode database can return SQLITE_BUSY, so applications should be prepared for that happenstance.

Cases where a query against a WAL-mode database can return SQLITE_BUSY include the following:

- If another database connection has the database mode open in exclusive locking mode then all queries against the database will return SQLITE_BUSY. Both Chrome and Firefox open their database files in exclusive locking mode, so attempts to read Chrome or Firefox databases while the applications are running will run into this problem, for example.

- When the last connection to a particular database is closing, that connection will acquire an exclusive lock for a short time while it cleans up the WAL and shared-memory files. If a separate attempt is made to open and query the database while the first connection is still in the middle of its cleanup process, the second connection might get an SQLITE_BUSY error.

- If the last connection to a database crashed, then the first new connection to open the database will start a recovery process. An exclusive lock is held during recovery. So if a third database connection tries to jump in and query while the second connection is running recovery, the third connection will get an SQLITE_BUSY error.

10. Backwards Compatibility

The database file format is unchanged for WAL mode. However, the WAL file and the wal-index are new concepts and so older versions of SQLite will not know how to recover a crashed SQLite database that was operating in WAL mode when the crash occurred. To prevent older versions of SQLite (prior to version 3.7.0, 2010-07-22) from trying to recover a WAL-mode database (and making matters worse) the database file format version numbers (bytes 18 and 19 in the database header) are increased from 1 to 2 in WAL mode. Thus, if an older version of SQLite attempts to connect to an SQLite database that is operating in WAL mode, it will report an error along the lines of "file is encrypted or is not a database".

One can explicitly change out of WAL mode using a pragma such as this:

```
PRAGMA journal_mode=DELETE;
```

Deliberately changing out of WAL mode changes the database file format version numbers back to 1 so that older versions of SQLite can once again access the database file.

11. The WAL-Reset Bug

On 2026-03-03, one of the SQLite developers (Dan) found and fixed a bug that could, in rare cases, lead to database corruption. We call this the "WAL-reset bug". Key points:

- The bug is likely present in all version of SQLite from 3.7.0 (2010-07-21) through 3.51.2 (2026-01-09). It is fixed in version 3.51.3 (2026-03-13) and later. Backports of the fix are available for some earlier releases: 3.44.6 and 3.50.7.
- The bug only affects databases in WAL mode when there are two or more database connections open on the same file, in separate threads or processes, and when those two connections attempt to write or checkpoint at the same instant.
- The bug is a data race with tight timing constraints. It is unlikely to occur in common use. The developers have never been able to reproduce the bug organically and had to add special testing logic to SQLite that deliberately triggers the circumstances of the the bug in order to verify that the issue has been fixed.

11.1. Bug Details

This is what happens:

1. One connection does a checkpoint. This first checkpoint must be complete. In other words, the checkpoint must successfully copy all content from the WAL file back into the database and leave the WAL file in a state where it can potentially be reset.
2. Shortly after the first checkpoint completes, a second checkpoint is started.
3. While the second checkpoint from step 2 is starting up, another database connection commits a transaction that resets the WAL file and writes new content into the beginning of the WAL file.
4. Due to a data race, the second checkpoint from step 2 does not realize that the WAL file has been reset by the transaction commit in step 3. The second checkpoint leaves a field in the header of the WAL-Index set incorrectly. That field indicates that part of the WAL file has already been checkpointed, when in fact it has not been.
5. Additional transactions are committed to increase the number of pages in the WAL file to be more than were present for the first checkpoint from step 1.
6. Later when a third checkpoint occurs, the third checkpoint skips all or part of the transaction that was written in step 3. Thus parts of the transaction from step 3 never reach the database file, and the database file goes corrupt.

11.2. Low Probability Of Occurrence

In order for this bug to happen, many details must align at just the right moment. So much so that the SQLite developers were unable to reproduce the bug organically in the lab. The only way the developers have been able to cause the malfunction was to modify the SQLite sources to invoke a callback controlled by sqlite3_test_control() that enables a test program or script to trigger the write transaction of step 3 at just the right moment during the second checkpoint. Without that code hack, the problem has never been observed during development and testing.

This bug, though rare, does have serious consequences, and so application developers should upgrade to a version of SQLite that fixes the problem. However, this is not an emergency. Based on available telemetry, the occurrence rate of this problem in the wild appears to be less than or equal to the expected occurrence rate of SSD malfunctions and/or cosmic-ray hits. So even if you are running an unpatched version of SQLite, and unless you are doing something really unusual, you are unlikely to ever encounter this problem.

This page was last updated on 2026-04-13 10:54:51Z