WAL internals

Key takeaways

1. A database commits atomically by logging pages, not by renaming a file.
2. The durable instant is a tiny marker: journal delete, or the commit record.
3. WAL trades per-commit fsyncs for checkpoints and keeps readers concurrent.

Common misunderstandings of SQLite WAL

Which claim would you defend right now? Choose before the evidence.

"SQLite must rename the whole database file for every atomic commit."

"WAL mode allows multiple writers to commit concurrently."

"Checkpointing is the instant that decides whether a transaction committed."

Keep your choice. Look for the mechanism or counterexample that would make it fail.

You already know the recipe

In W3 the durable-write recipe swapped a file in whole: write a temp copy, fsync it, atomically rename it over the target, fsync the directory. The rename was the single all-or-nothing instant a crash could not tear.

W3: durability across two files

```
write_index(tmp); os.fsync(tmp_fd)   # new bytes safe on the platter
os.replace(tmp, "idx.db")            # atomic switch: all-old OR all-new
os.fsync(dir_fd)                     # the switch itself is durable
```

SQLite runs this same dance inside one file — continuously, once per transaction.

The files next to your database

In P1 your email index was a SQLite file. Look beside it and you find sidecars — this is where the crash-safety machinery lives. Which ones appear tells you the mode.

ls beside a SQLite database

```
mail.db           # the database: committed pages
mail.db-wal       # write-ahead log: NEW page images not yet checkpointed
mail.db-shm       # shared-memory wal-index: helps readers find newest frame
mail.db-journal   # rollback journal: OLD page images (the other mode)
```

WAL mode leaves -wal + -shm; rollback mode leaves -journal. Never both — two designs for one job.

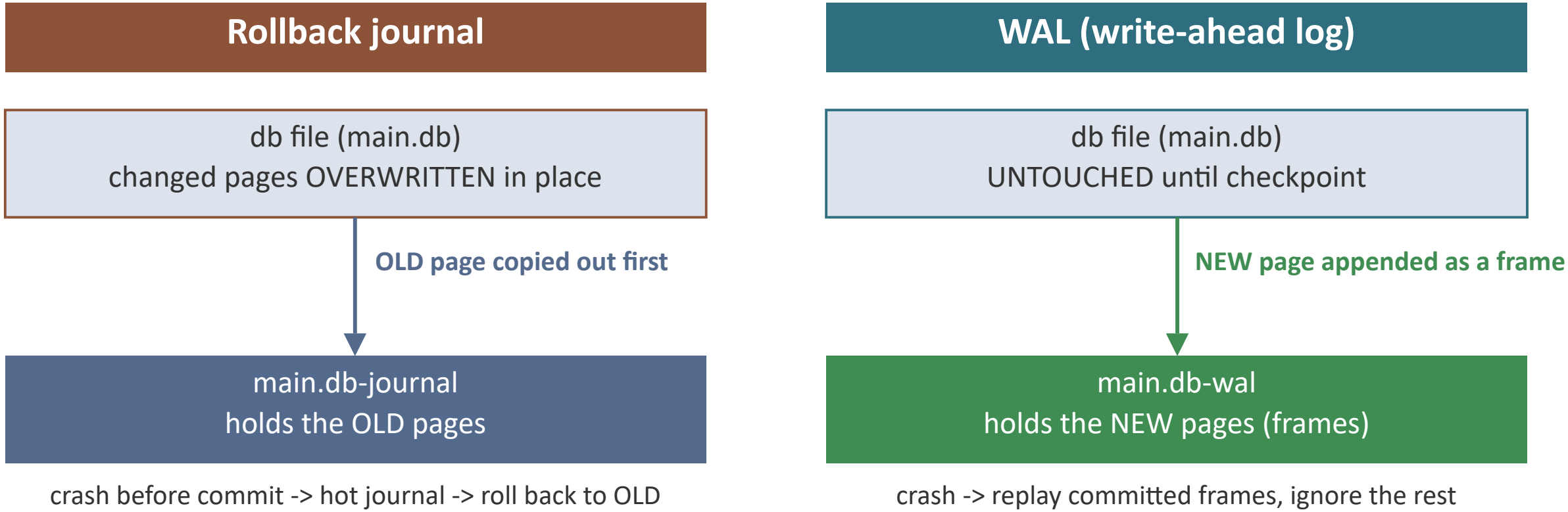
Instapoll

Instapoll · Review · Sep 24 A

You see mail.db, mail.db-wal, and mail.db-shm beside an email index. Which file would you be most reluctant to delete while the database is open?

- A. mail.db because it contains checkpointed database pages
- B. mail.db-wal because it may contain committed pages not yet checkpointed
- C. mail.db-shm because it is the only durable copy of every database row
- D. All three are disposable because SQLite reconstructs them from queries

Two designs, opposite payloads



Rollback journal saves the OLD page and overwrites the db; WAL leaves the db alone and appends the NEW page.

Rollback journal: the fsync sequence

To commit, SQLite copies the old pages out, then overwrites the db in place. The order of fsyncs is what makes a crash recoverable — and it costs two journal syncs plus one db sync per commit.

Commit = 2 journal fsyncs + 1 db fsync, then delete the journal

```
1. write OLD pages -> mail.db-journal ; fsync(journal)   # originals safe
2. set journal header page-count 0 -> N ; fsync(journal) # now it is "hot"
3. write NEW pages into mail.db (in place)
4. fsync(mail.db)                                         # changes durable
5. delete mail.db-journal                                # <- the commit
```

Deleting the journal is the commit. Crash before step 5 -> a hot journal -> roll back to OLD.

Torn pages: journal the whole sector

SQLite assumes a sector write is linear but not atomic: a power cut can leave the sector it was writing half-changed. So when any page in a sector changes, it saves every page in that sector — otherwise recovery could not rebuild the neighbors the hardware had to rewrite.

Torn-write safety is a sector property, not a page property

```
# SQLite's own example: pages 1-4 share one 512-byte sector
change page 2   ->  hardware rewrites the whole sector (pages 1-4)
crash mid-write ->  pages 1, 3, 4 may be garbage too
so the journal holds pages 1,2,3,4  +  a 32-bit checksum per page
```

A failed checksum marks the journal incomplete — roll back nothing; the db is already OLD.

WAL: append the new page, never overwrite

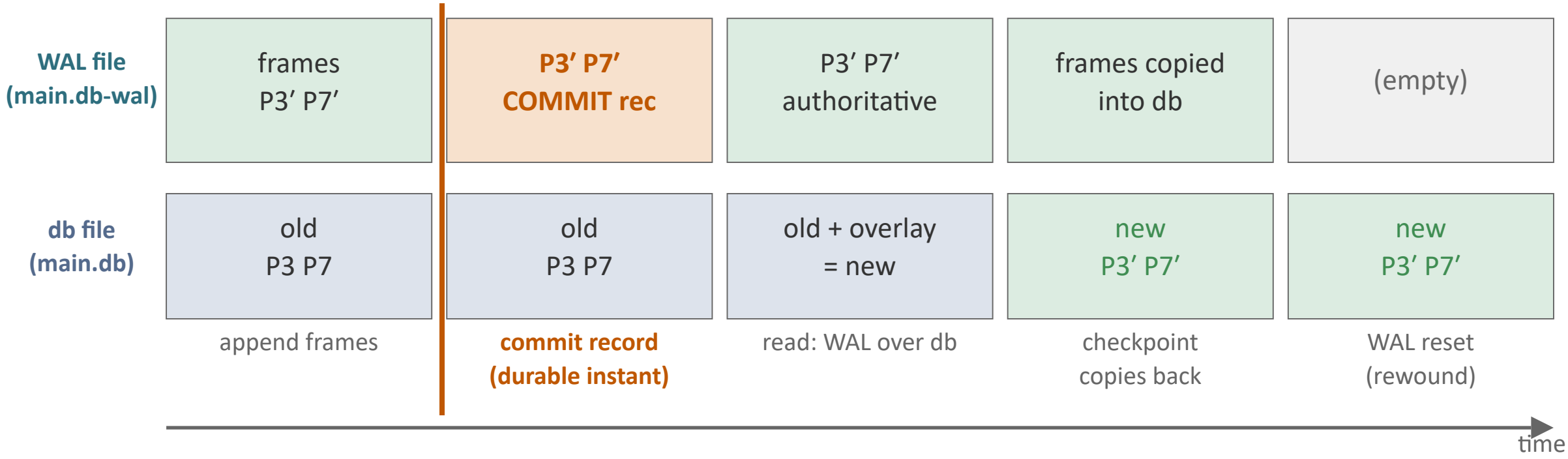
Turn on WAL and the db file goes read-mostly. Each changed page is appended to -wal as a frame; the db is not touched until checkpoint. A commit is one special frame — the commit record — appended to the log.

One pragma; persistent across reopen

```
PRAGMA journal_mode=WAL;    -- sticky: reopen comes back in WAL mode
-- writes append frames to mail.db-wal
-- readers consult mail.db-shm (the wal-index) for the newest frame,
--   then fall back to mail.db for pages not in the log
```

The commit record is the durable instant — the WAL analog of the atomic rename.

Checkpoint: copy frames back, then reset



Auto-checkpoint fires past 1000 WAL pages (PASSIVE default): copy frames into the db, then rewind the log.

Fewer fsyncs, and readers don’t block

The rollback journal syncs on every commit; WAL moves that cost to the checkpoint, so a commit is one append (one WAL sync at synchronous=FULL, none at NORMAL). And because the db file stays put, readers and the writer run at the same time.

Rollback journal — per commit

```
fsync(journal)  # originals
fsync(journal)  # header
write db in place
fsync(db)
# 3 fsyncs; the writer
# locks readers out
```

WAL — per commit

```
append frames
append commit record
fsync(wal)   # FULL only
# 0-1 fsyncs; the rest
# batched at checkpoint
# readers stay concurrent
```

One writer at a time either way — but WAL lets any number of readers keep going.

Five crash states — and none corrupts

The commit record is the atomic switch: crash before it -> OLD, crash after it -> NEW. Never torn.

Crash point	db file has	wal file has	Recovery -> result
mid-append (no commit record yet)	old pages	partial frames, no commit	**ignore frames -> OLD**
commit record synced, not checkpointed	old pages	full txn + commit record	**replay WAL -> NEW**
checkpoint mid-copy	some new, some old	all frames present	**WAL still wins -> NEW**
checkpoint done, WAL not reset	all new pages	stale frames (redundant)	**db already current -> NEW**
after WAL reset	new pages	empty	**nothing to replay -> NEW**

Key Takeaways

1. A database commits atomically without a rename by keeping a log: the journal saves the OLD page, WAL the NEW one
2. The single durable instant is a tiny marker — the journal's deletion or the WAL's commit record — not the bulk page writes
3. WAL trades per-commit fsyncs for a periodic checkpoint and lets readers run concurrently with the writer

what does recovery replay?

10 min

- Start with db=OLD and an empty WAL. Fill five crash rows: (1) before frames; (2) frames present, no commit; (3) commit durable, before checkpoint; (4) checkpoint partly copied, WAL intact; (5) checkpoint complete and WAL reset.
- For each row, write: db contents, WAL contents, recovery=OLD or NEW.
- Work alone for three minutes, then compare with a partner and resolve the first row where you disagree.
- Circle the boundary where recovery flips from OLD to NEW and name the record that causes it. Deliverable: the five-row table and boundary.