WAL internals

Key takeaways

1. A database cannot rename a multi-gigabyte file for every transaction, so it commits atomically by keeping a log of page images inside its own files.

2. The all-or-nothing instant is one tiny marker, not the bulk page writes: deleting the rollback journal, or appending the commit record to the log.

3. Write-ahead logging cuts a commit from four fsyncs to one and defers the rest to a checkpoint; because the database file stays put, readers keep running while the writer commits.

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.

The rename recipe: write, fsync, rename, fsync

Crash-consistent applications swapped a file in whole: write a temp copy, fsync it, rename it over the target, fsync the directory. The rename was the one all-or-nothing instant a crash could not tear.

Crash-consistent applications: 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
```

Why two: fsync(tmp) covers that file's bytes and length, nothing else. The name idx.db lives in the directory's own blocks, and rename edits those — new entry or old one repointed, it is the same edit. Skip fsync(dir) and a crash leaves the old idx.db beside a leftover tmp: consistent, just not saved.

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

Sidecar files for crash consistency

Your email index is one SQLite file plus whatever SQLite keeps beside it. Which sidecars exist tells you which crash-safety design the file is running.

mail.db	the database file
mail.db-wal	NEW pages committed since the last checkpoint — WAL mode
mail.db-shm	index over -wal: page number -> its newest frame — WAL mode
mail.db-journal	OLD pages of the transaction in flight — rollback mode

The -wal file is part of the database: copy mail.db without it and every commit since the last checkpoint is gone. The -shm file is scratch: the next open rebuilds it from -wal.

WAL mode leaves -wal + -shm; rollback mode leaves -journal — never both.

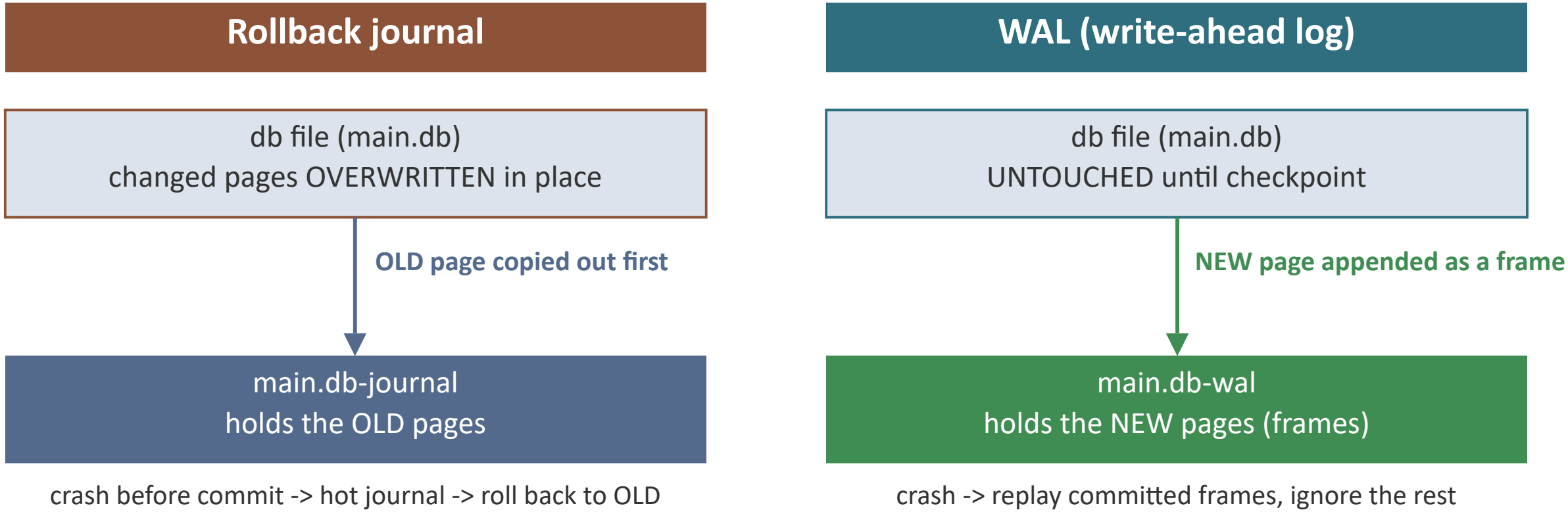
Instapoll

Instapoll · Review · Sep 15 A

Your database is in WAL mode and a transaction has just committed. Where are the new bytes right now?

- A. In mail.db — the commit wrote them there, and mail.db-wal keeps the old pages
- B. In mail.db-wal — mail.db keeps the old pages until a checkpoint copies them over
- C. In both, because a commit writes every page twice so one copy always survives
- D. In mail.db-shm, the staging area the writer and every reader share

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 is the default; WAL is opt-in

Both commit atomically; they differ in cost, in what mail.db alone means, and in where they run.

	Rollback journal (default)	WAL
fsyncs per commit	4 (FULL, the default): journal pages + the journal's new name, journal header, then the db	1 (FULL, the default) or 0 (NORMAL) — an append to the existing -wal changes no name; checkpoint syncs -wal, then db
mail.db by itself	Complete: a copy holds every commit	Missing commits since the last checkpoint: they live in -wal
A commit while readers are open	Waits for every reader to close	Appends past them; each keeps the view it started with
Where it runs	Any filesystem	One host with shared memory (-shm): not over NFS

WAL: cheap commits, unblocked readers; the cost is a checkpoint and a two-file db.

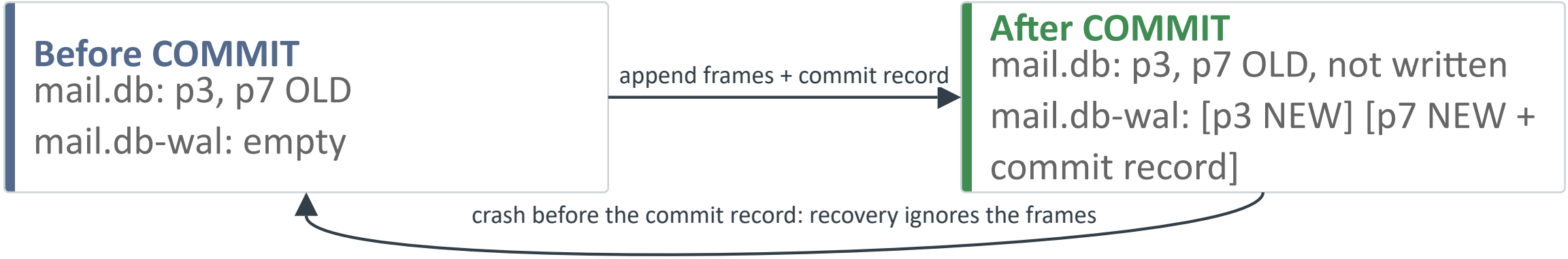
Rollback commit: save OLD, overwrite, delete

1. **Write the OLD pages to mail.db-journal**
   fsync(journal) + fsync(dir): a new file each commit, and its name must be durable before step 3 touches mail.db — else a torn db with no journal to find
2. **Set the journal header's page count 0 -> N**
   fsync(journal) again, same file: pages before header, so N never vouches for garbage
3. **Write the NEW pages into mail.db, in place — no sync yet**
4. **fsync(mail.db): the changes are durable before the journal goes**
5. **Delete mail.db-journal — no fsync(dir); synchronous=EXTRA adds one**
   this is the commit: a crash can undo the delete, but that only leaves a hot journal, and this one commit rolls back to OLD — durability lost, nothing corrupt

Four fsyncs; the delete is the commit. Crash before it -> a hot journal -> roll back to OLD.

WAL: append pages, then the commit record

PRAGMA journal_mode=WAL, once, and from then on each changed page is appended to -wal as a frame; the transaction's last frame carries the commit mark and is the commit record. A half-written record fails its checksum, so recovery treats it as absent.



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

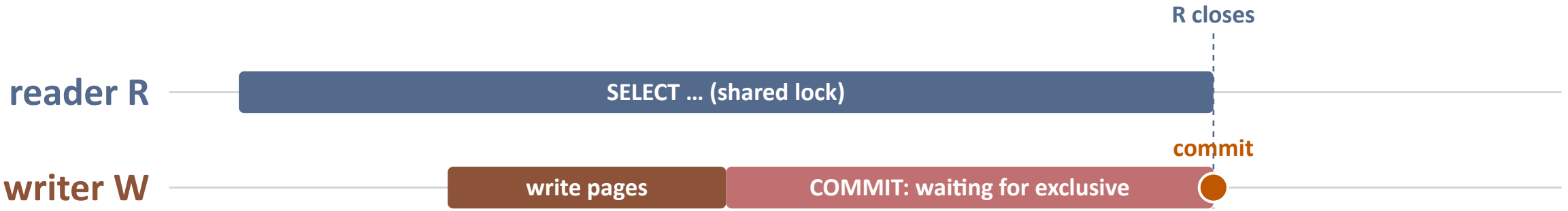
Append, commit, checkpoint, reset

	1 append p3, p7 frames	**2 commit** commit record	**3 checkpoint** copied to db	**4 reset** log starts over
WAL file mail.db-wal	**NEW frames** no commit yet	**NEW + commit**	**NEW + commit** already copied	empty
db file mail.db	**OLD**	**OLD** untouched	**NEW**	**NEW**
reader sees starting now	**OLD** frames ignored	**NEW** from the WAL	**NEW** from either	**NEW** from the db

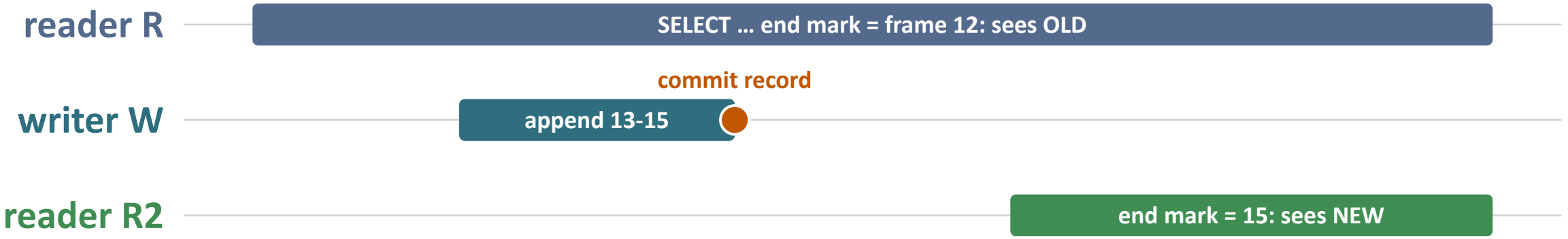
A reader’s view flips at the commit record; the db file catches up at checkpoint (auto past 1000 WAL pages).

The writer never waits for a reader

ROLLBACK JOURNAL: COMMIT WAITS FOR THE READER



WAL: THE WRITER COMMITS PAST THE READER'S END MARK — STILL ONE WRITER AT A TIME



Five crash states — and none corrupts

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

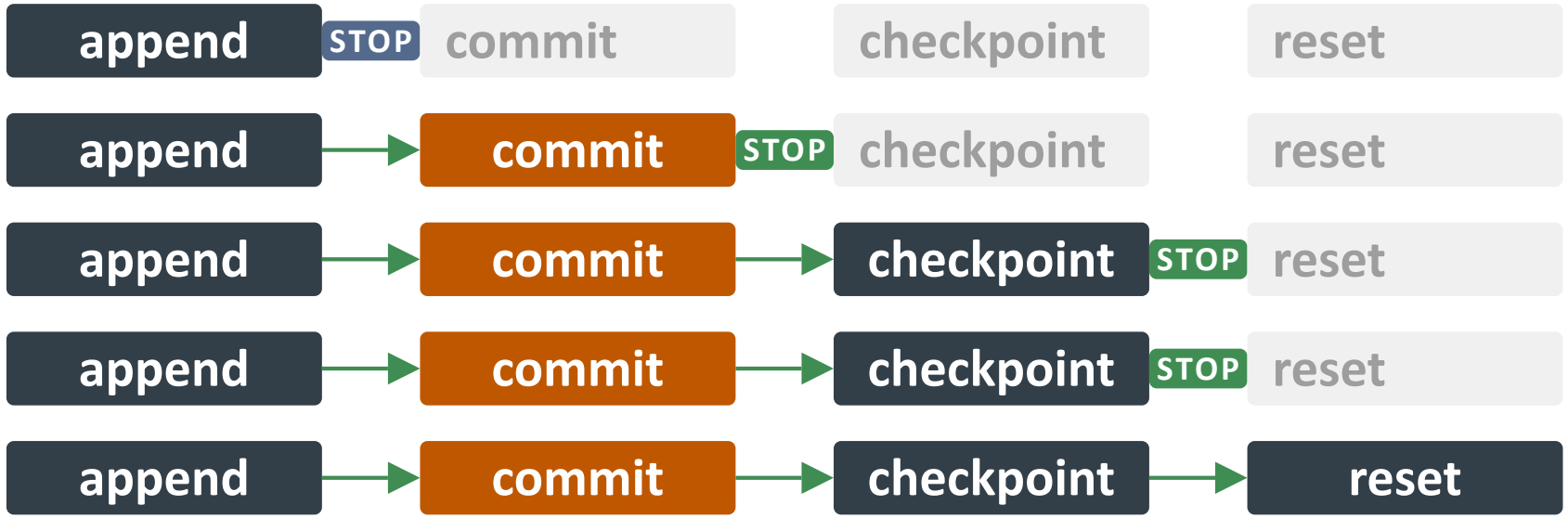
MID-APPEND
OLD: frames ignored

COMMIT RECORD DURABLE
NEW: replay the WAL

CHECKPOINT MID-COPY
NEW: the WAL still wins

CHECKPOINT DONE, NOT RESET
NEW: db is current

AFTER RESET
NEW: nothing to replay



Your spec in WAL terms: a crash and a reader

Each guarantee is one mechanism from this deck, and each has a one-line check

SPEC-v2 says	The WAL fact	Check it on your index
“An update may be terminated without advance notice. The index must remain openable, with either the prior complete publication or the complete replacement visible.”	The commit record decides: crash before it recovers OLD, after it NEW, never a mix.	fault_point(“commit”) sits just before the step that appends that record, “committed” just after. Run each with exit; open the index.
“Each reader retains a valid published view throughout its lifetime. The writer must be able to complete while readers are active.”	A reader notes the log’s end mark when it starts and never reads past it; the writer appends beyond it. A rollback-journal commit waits for the reader instead.	PRAGMA journal_mode; on your index. delete means one held-open search blocks every update.

source: your email-index kit: SPEC-v2.md, Resilience guarantees; https://sqlite.org/wal.html

Your spec in WAL terms: writers and size

SPEC-v2 says	The WAL fact	Check it on your index
“Competing updates must not write simultaneously. A contender may wait, or exit promptly without writing with update in progress on stderr.”	One writer per database in either mode: the second BEGIN IMMEDIATE gets SQLITE_BUSY.	sqlite3.connect(timeout=…) is “wait”; catching OperationalError is “exit promptly”. Choose one and print what the spec asks for.
“All persistent preparation and coordination artifacts count as index data. A successful update with no overlapping processes must restore the Build size bound.”	-wal holds every frame since the last checkpoint and cannot rewind while a reader is open. Auto-checkpoint fires past 1000 pages; the last connection to close runs one more.	ls -l PATH* after update. PRAGMA wal_checkpoint(TRUNCATE) before exit keeps the bound from depending on who closed last.

source: your email-index kit: SPEC-v2.md, Resilience guarantees; https://sqlite.org/wal.html

Key Takeaways

1. A database gets atomic commit without renaming anything by writing a log first. The rollback journal saves the old page and then overwrites the database in place; the write-ahead log leaves the database alone and appends the new page.

2. The bulk page writes are not the commit. One small marker decides it: deleting the journal, or appending the commit record to the log. When you reason about a crash, ask only whether that marker survived.

3. WAL cuts a commit from four fsyncs to one; a periodic checkpoint copies frames back into the database, and readers never block on the writer. Either mode still allows only one writer at a time. Each Resilience guarantee names one of these mechanisms.

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. Then find the line in your update where fault_point(“commit”) runs and say which side of that boundary it is on. Deliverable: the five-row table, the boundary, and that line.