Zaxon Working Group
Request for Discussion and Implementation Record

ZDS 0002
Architectural Specification

Zaxonlite: Product and Delivery Plan

STATE
published

CREATED
2026-07-19

LAST UPDATED
2026-07-21

INTENDED STATUS
Published

AUTHORS
paxos-zig project

DISCUSSION

Implementation specification for the Zaxonlite embedded replicated SQLite library and the zaxon CLI

LABELS
architecture, zaxonlite, product

Status of This Memo

This document is an internal Zaxon discussion record authored in Typst and tracked in git. It intentionally follows RFC-style structure so the design scope, rationale, trade-offs, and operational constraints remain explicit. Documents that still use the placeholder number XXXXX are provisional drafts. Numbered ZDS documents are part of the project record.

Contents

1. Decision summary

Build an embeddable Zig library named **Zaxonlite**, that links SQLite into the application process and uses this repository's `paxos.ReplicatedLog` as its only consensus engine. It will expose a deliberately small API for opening a node, executing SQL, querying, joining a cluster, backing up, and inspecting health. A companion `zaxon` CLI will provide an interactive SQL shell, a standalone host for applications that do not embed the library directly, and operational commands over the same client and administration protocols.

Replicate SQLite's committed WAL/page changes rather than SQL strings. The leader executes a transaction once; followers apply the exact resulting page frames in Paxos slot order. This supports SQL that uses timestamps, randomness, triggers, and application-defined functions without asking replicas to produce identical results independently.

The Paxos journal is authoritative durable state. The SQLite database image on each node is a materialized state machine that can be reconstructed from a snapshot plus the committed journal suffix. This avoids making two unrelated files jointly authoritative and gives recovery one explicit order.

Important performance claim

Paxos does not by itself guarantee higher write throughput than dqlite's Raft implementation. Both can use a stable leader and commit in one network round trip. Our measurable opportunity is the combination of bounded pipelining, group sync, transaction batching, asynchronous I/O, and optional flexible quorum policy. "Faster than dqlite" is therefore a benchmark gate, not a design assumption.

1.1. Product principles

1. **Embedded first.** SQLite, the Paxos node, storage, and transport live in the caller's process. A standalone server may be a thin example, never a required deployment component.
2. **One API for every supported topology.** A one-node membership uses the same journal, recovery, transaction, and snapshot code as a cluster. The voter set is deliberately small; non-voting replicas and gateways are runtime registry entries and never enlarge a Paxos quorum.
3. **Durability before evidence.** Persist every `Effects.writesSlice()` in order, sync it, call `confirmWritesDurable()`, and only then send `messagesSlice()` or apply `committedSlice()`.
4. **SQLite semantics over statement replay.** Replicate transaction effects, not SQL text.
5. **Safe defaults.** A successful write is durable on a quorum. Reads are linearizable by default. Stale local reads require an explicit option.
6. **Small operational surface.** An explicit bootstrap registry and voter set, one data directory per storage node, one endpoint, and a compact health API.

7. **Evidence over slogans.** Safety, recovery, and performance claims each have a named test or benchmark and an exit threshold.

2. Scope and product contract

2.1. First release

The first production-shaped release includes:

1. a Zig API and a small C ABI suitable for embedding;
2. a `zaxon` CLI for serving a node, interactive SQL, scripted execution, status, backup, snapshots, and integrity checks;
3. SQLite compiled and linked into the same process;
4. a one-node durable mode and TCP clusters with one to nine configured voters plus runtime-sized non-voting replicas;
5. explicit `data-voter`, `witness`, `standby`, `read-replica`, and `gateway` roles with capability validation and a role-pinned data-directory identity;
6. a transport-owning Zig facade and matching C facade for embedded clusters;
7. leader discovery and transparent client redirection/retry;
8. implicit and explicit SQLite transactions with one writer at a time;
9. exact transaction-effect replication using captured WAL frames;
10. a maximum encoded transaction payload of 64 MiB minus 73 bytes of hash and authenticated wire framing in protocol v6; larger transactions fail without entering Paxos;
11. linearizable leader reads using a Paxos quorum read fence, with a committed barrier as the bring-up/fallback path, and opt-in bounded-stale local reads;
12. idempotent write retry using bounded, replicated client sessions and monotonic request sequence numbers;
13. crash-safe journal recovery, snapshots, catch-up, and checksummed files;
14. mutual TLS 1.3 on every production TCP storage connection, including loopback, with canonical per-node certificate identities; optional PSK/HMAC runs inside TLS, while a local single-node service may use an owner-only UDS;
15. health, leader, applied-slot, snapshot, and integrity inspection APIs.

2.2. Explicit non-goals for the first release

1. multi-writer execution on different leaders;
2. geo-distributed latency optimization;
3. more than nine voting acceptors in one consensus group, automatic voter replacement, or claiming that millions of all-to-all voters are practical;
4. follower writes without leader forwarding;
5. distributed transactions across multiple database files;
6. compatibility with every SQLite VFS or virtual-table extension;
7. claiming Byzantine fault tolerance;
8. preserving an ordinary SQLite database file as the sole durable source of truth.

2.3. User-visible guarantees

Contract	Meaning
Successful write	The transaction was chosen by Paxos, its value was durably accepted by a write quorum, and the responding leader applied the committed frame batch. It will not be replaced by another value.

Contract	Meaning
Timed-out write	The result is unknown. Retrying the latest sequence in the same live client session returns the recorded result and never applies the transaction twice. An older or expired sequence is rejected, never re-executed.
Default read	The result includes every write acknowledged before the read began. Bring-up uses a committed barrier; the stable release uses the quorum-confirmed Paxos read fence specified below.
Local stale read	The result is a consistent SQLite snapshot at or before the node's reported `applied_slot`; recency is not guaranteed.
One failed voter	A three-voter cluster continues while any two voters can communicate and sync durable state.
No quorum	Reads may be served only under the selected stale policy; writes fail or time out and are never acknowledged as successful.
Total restart	Each node validates and replays its journal, elects a leader, resolves any chosen-but-not-yet-committed value, and rebuilds the same committed database state.

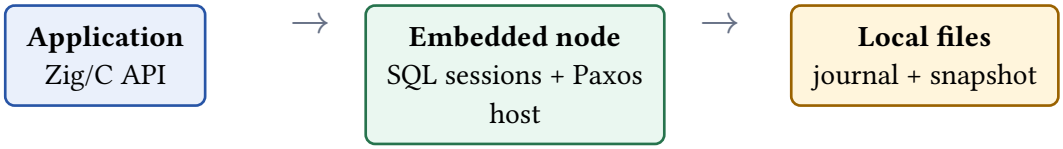
2.4. Node roles and scaling law

Type	Votes	Campaigns	SQLite reads	Purpose
`data-voter`	yes	yes	yes	Normal acceptor/proposer/learner and materialized database.
`witness`	yes	no	no	Durable promise/vote and payload copy that reduces failure-domain cost without becoming a SQL leader.
`standby`	no	no	stale local	Full chosen-log and SQLite copy, eligible for a later controlled promotion.
`read-replica`	no	no	stale local	Read scaling without changing write quorum or failure tolerance.
`gateway`	no	no	routes only	Stateless end-to-end TCP routing; owns neither Paxos nor SQLite state.

For voter set V, a majority is $q = \left\lfloor |V| \frac{1}{2} \right\rfloor + 1$ and tolerates $f = \left\lfloor \frac{|V|-1}{2} \right\rfloor$ voter failures. Learners are absent from V, so adding one million learners would not change q mathematically. It would still be an engineering error to create one million all-to-all connections or threads; actual scale is bounded by transport, storage, and placement resources and by sharding independent database groups. The implementation profile supports at most nine voters, matching the practical 3/5/7/9 operational range, while the total registry is allocator-backed rather than a seven-node array.

3. Architecture

3.1. Process topology



Each application process owns one node. In a three-node deployment the embedded nodes also maintain authenticated peer connections. SQL calls never require a separate database process. A non-leader library instance forwards a write to the leader or returns a structured leader hint according to configuration.

3.2. Component boundaries

Component	Responsibility	Does not own
`sqlite`	Prepare/step/finalize statements; transaction isolation; capture committed WAL frames through the supported WAL hook plus validated direct WAL reads, and consume them through offline page apply.	Consensus, networking, cluster durability.
`state_machine`	Turn one committed frame batch into one atomic SQLite state transition; maintain `applied_slot`, chain identity, and bounded client-session records.	Leader election.
`paxos_host`	Drive `paxos.ReplicatedLog`, serialize protocol effects, tick, route messages, and release committed entries in order.	SQL execution.
`journal`	Frame records with length, type, epoch, checksum, and sequence; append/sync/replay/truncate torn tail.	Interpreting SQL pages.
`payload_store`	Persist transaction frame payloads by content hash before a corresponding Paxos vote becomes sendable.	Choosing log order.
`transport`	Peer framing, authentication, backpressure, reconnect, and protocol/payload transfer.	Durability decisions.
`snapshot`	Atomically install database image, manifest, membership epoch, applied slot, chain identity, and session state.	Deleting data before quorum coverage.
`client`	Sessions, leader routing, deadlines, and idempotent retry.	Silently guessing an ambiguous write result.
`admin`	Authenticated status, snapshot, backup, integrity, and controlled lifecycle operations used by the CLI.	Bypassing consensus or editing live files.

Component	Responsibility	Does not own
`cli`	Interactive SQL, scripts, machine-readable output, a reference stand-alone host, and operator commands.	A second storage or consensus implementation.

3.3. Replicated command

Do not place a large transaction directly inside the current bounded Paxos `Value`. Use a fixed-size command descriptor:

```
const Command = union(enum) {
    noop: void,
    transaction_batch: struct {
        database_id: u128,
        batch_id: u128,
        base_data_slot: u64,
        base_chain_hash: [32]u8,
        result_chain_hash: [32]u8,
        payload_hash: [32]u8,
        payload_bytes: u64,
        transaction_count: u32,
        frame_count: u32,
    },
    read_barrier: struct { nonce: u128 },
};
```

The payload contains a versioned header, per-transaction session/sequence IDs and bounded results, transaction boundary markers, and the associated WAL frames. It contains page size, database identity, and count metadata. The fixed descriptor—not the payload—carries the parent chain and the SHA-256 content address that verifies the whole payload. A one-transaction write is represented as a batch of one. Values are content-addressed. An acceptor must not pass an incoming `accept` descriptor to `ReplicatedLog.step` until the referenced payload is present and durably synced locally. Consequently every counted vote has the bytes needed to recover the value, not merely its hash.

`base_chain_hash` and `result_chain_hash` are cumulative replicated-history identities, not hashes of the SQLite file. For a canonical descriptor encoding, define

$$C_i = H(C_{i-1}, \text{database-id}, \text{batch-id}, \text{base-data-slot}, \text{payload-hash}, \text{metadata})$$

where C_{i-1} is `base_chain_hash` and C_i is `result_chain_hash`. The inputs use a domain-separated, length-delimited canonical encoding; the displayed commas denote that unambiguous concatenation. Computing the chain step is $O(1)$ in database size. Computing `payload_hash` is $O(\text{payload\_bytes})$, which is already required to validate transfer; no transaction scans a gigabyte-scale database image. Snapshot-time `logical_content_hash` or page-digest verification remains a separate, infrequent corruption check. A history hash detects a dependency mismatch; it does not prove that the SQLite file was applied correctly.

Required Paxos-host invariant

A chosen descriptor without recoverable payload bytes is not a usable database command. Payload persistence is therefore a precondition to delivering `accept` into the Paxos core. During phase-one recovery, a leader that discovers an accepted descriptor fetches and verifies

its payload before reproposing it. This host-level rule needs fault tests in addition to the core protocol tests.

For the earliest vertical slice, an inline, strictly bounded transaction value is acceptable if it makes the invariant easier to prove. It is not acceptable as the long-term representation because `DurableState` retains bounded values per slot and database transactions can be large.

3.4. External payload transfer protocol

Normal Phase 2 uses ordered prestaging plus a receiver-side gate:

1. Queue `payload_data(hash, bytes)` immediately before the Paxos `accept` on the same ordered TCP/TLS stream.
2. The receiver reserves bounded space, verifies the digest, syncs and atomically installs the object before its read loop reaches the next frame.
3. `payload_stored(hash)` caches readiness for later sends; the normal adjacent accept does not wait for that round trip. Reconnects remain idempotent because payloads are content-addressed.
4. A reordered or separately arriving `accept` is retained only in the bounded missing-payload gate and is never stepped into Paxos until the object exists.

`Node.append` emits local durable writes and peer `accept` envelopes in one `Effects` batch. The host appends the local effects, releases only the accept requests classified by `preDurableMessages`, then runs its local barrier. The host mutex prevents an accepted reply from re-entering the core until that barrier completes. Every queue has explicit object and byte limits.

Phase-1 recovery has a stricter integration issue. In the current code, `onPromise` stores each reported descriptor and `maybeBecomeLeader` can emit recovery accepts in the same transition that completes the promise quorum. With the current API, the host must materialize and verify the payload of every value-bearing `promise` before passing that envelope to `ReplicatedLog.step`. The promise sender is a source because it was forbidden to report a durable accepted descriptor without its payload. Fetches are parallel but bounded.

After the first correct integration, add a general staged-recovery core API that reports the highest recovered descriptors, pauses activation, and resumes only after the host confirms their values are materialized. That avoids fetching lower-ballot candidates and keeps bulk transfer out of the final activation transition without changing Paxos value selection. The same materialize-before- step rule applies to inbound `commit` messages so a node never durably learns or releases a descriptor whose bytes it lacks.

A hybrid inline/external value is an optional measured optimization, not a safety requirement. If adopted, its inline array and threshold must be included in the compile-time `DurableState` memory formula; an IP MTU alone is not a sound limit for a durable application value.

3.5. Write path

1. Route sessions to the current leader and validate the session's next monotonic sequence number. Concurrent callers enter a bounded writer queue.
2. Execute one transaction, or a bounded group of queued transactions, in SQL order against a speculative SQLite branch. The custom VFS/WAL layer captures each commit boundary without publishing it to the canonical state machine.
3. Encode, checksum, and sync the payload in the leader's payload store.
4. Call `ReplicatedLog.append(command, &effects)`.

5. Append `effects.writesSlice()`. Release only classified phase-two accept requests while the leader barrier runs; then call `confirmWritesDurable()` and send all durability-bearing messages.
6. For each follower, queue the payload immediately before its accept. The follower stores it before reading the accept, then journals and syncs its vote before sending `accepted`.
7. When a quorum chooses the slot, treat the local commit marker as derived state: do not pay a second barrier. Apply `committedSlice()` strictly in contiguous slot order. Phase one reconstructs the choice after a crash.
8. Reply to each transaction only after the leader has applied the whole slot. Atomically retain the bounded result for each session's latest sequence so an ambiguous retry returns the same outcome.

The speculative SQLite work must be discarded if leadership changes or the descriptor loses its slot. Only committed frame batches touch canonical state. The descriptor's chain hashes turn an invalid dependency into a detected fatal invariant violation rather than silent page corruption. A chosen incompatible value must never be skipped or treated as a failed transaction; prevention is therefore required, as specified in the dependency theorem below.

3.6. Read path

The bring-up implementation appends a `read_barrier`, waits until that slot is committed and applied, and then opens a SQLite read transaction. This is the simple reference path and remains a fallback, but an fsync plus a Phase-2 round per `SELECT` is not an acceptable stable-release performance path.

The stable release adds a **Paxos quorum read fence**, not a wholesale copy of Raft ReadIndex. The prepared proposer captures its current `decided_through` k, sends a fresh `(ballot, nonce, k)` challenge, and counts a response only if the acceptor's durable promise is exactly that ballot. After replies from a Phase-1/read quorum, the proposer verifies it still has that ballot, waits until local SQLite is applied through at least k, and opens the read transaction. The fence changes no durable acceptor state and therefore requires no fsync. Concurrent reads that were registered before the fence completes may share the round; later reads need a new fence unless a separately proved lease is active.

Current `heartbeat` handling in `src/protocol.zig` cannot implement this: it has no nonce and no affirmative heartbeat ACK. Zaxonlite therefore implements a bounded host-level challenge/ACK frame keyed by a fresh fence ID, includes the captured slot, validates exact ballot equality, and counts distinct configured members. Reusing a periodic heartbeat response or a wall-clock "recent quorum" is unsafe. A generic core helper remains an optional API consolidation, not a safety dependency. Followers may serve local snapshot reads only when the caller requests stale consistency and receives `applied_slot` in the response.

3.7. SQLite integration decision

Use the SQLite amalgamation as a pinned build dependency and expose only the small subset of the C API needed internally. Never patch SQLite.

Phase 0 ADR — capture technique (decided, implemented)
The implemented capture layer uses public hooks only: the node runs one writer connection in WAL mode with automatic checkpoints disabled; `sqlite3_wal_hook` reports the committed frame count after every commit, and the exact frame bytes are read directly from the `-wal` file between the previous and new count. Rolled-back transactions never advance the

hook count, so abandoned frames are never captured. Checkpoints run only at snapshot boundaries, after which capture restarts at frame zero. Apply is deterministic page-level materialization: each frame's page image is written at `(page_number - 1) * page_size` and the file is truncated to the commit frame's page count. `zaxonlite/src/wal.zig` holds both sides; the spike test rebuilds a byte-identical database across DDL, triggers, BLOBs, savepoints, rollbacks, and nondeterministic SQL. A full custom VFS remains the recorded fallback if a future SQLite release changes WAL internals or if follower live-apply (a Phase 2 concern) cannot be built on offline apply plus connection reopen.

The capture layer must preserve:

1. page size and database identity;
2. transaction commit boundaries;
3. frame order and checksums;
4. rollback behavior and savepoints;
5. concurrent readers with one writer;
6. schema changes, triggers, nondeterministic functions, and large BLOBs;
7. checkpoint behavior without changing replicated history.

If public SQLite hooks cannot provide a maintainable capture/apply boundary, stop after the spike and record an ADR. Do not hide a private SQLite patch in the product.

4. Persistence and recovery design

4.1. File set

Each node owns one directory:

```
node/
  LOCK                    # exclusive advisory lock; one process per node
  identity                # node/database IDs, current configuration
  paxos-<config16hex>.log  # framed protocol journal, one per epoch
  payloads/aa/bb...       # immutable transaction payload by SHA-256
  snapshots/tmp-*         # incomplete, never selected on restart
  snapshots/<config>/     # db image + manifest per sealed epoch
  CURRENT                 # installed snapshot pointer
  current.db              # materialized SQLite image (rebuildable)
```

Every mutable metadata update uses write-temp, sync-file, atomic-rename, and sync-parent-directory. Every journal record includes a magic value, format version, record kind, byte length, monotonically increasing sequence, payload, and checksum. Recovery rejects corruption in the durable prefix and truncates only an incomplete final record. (Implementation note: the materialized image records the last applied `batch_id` in a replicated `__zaxon_meta` table updated inside every captured transaction, so recovery can validate the image's position against the committed descriptor sequence; SQLite's own `-wal`/`-shm` files are working artifacts deleted on open, never authoritative.)

4.2. Ordering rules

Step	Must happen first	May happen only afterward
1	Transaction payload reaches durable storage.	Deliver its descriptor to the local Paxos acceptor.

Step	Must happen first	May happen only afterward
2	All Paxos writes from one transition are appended and synced.	Call `confirmWritesDurable()` and send dependent messages.
3	A commit record is synced and earlier slots are available.	Apply the transaction to canonical SQLite and expose it to readers.
4	SQLite apply and client-session update complete for the slot.	Advance `applied_slot` and acknowledge the client.
5	A snapshot and manifest are synced and known recoverable on a quorum.	Seal/checkpoint the epoch and later reclaim covered journal/payload files.

Group commit may sync several independent Paxos transitions together. It may not send any transition's dependent message before the group sync succeeds. On I/O failure the node must stop voting and serving writes; a failed disk is safer than a node that advertises state it did not persist.

4.3. Restart sequence

1. Lock the node directory and validate `identity`.
2. Select the newest complete snapshot named by a valid `CURRENT` file.
3. Replay journal records after the snapshot into `ReplicatedLog.DurableState.apply`, validating monotonicity and checksums.
4. Verify every accepted or committed command descriptor has its payload. Mark the node unavailable and attempt authenticated peer repair if one is absent; never vote with missing bytes.
5. Restore `ReplicatedLog.Node`, reconstruct canonical SQLite from the snapshot, and apply the committed contiguous suffix.
6. Rebuild the bounded client-session table before accepting clients.
7. Join election/catch-up traffic; advertise ready only after local SQLite has reached the node's committed prefix.

4.4. Snapshot and log compaction

Use `ReplicatedLog.checkpoint(snapshot_metadata, effects)` to decide a stop sign whose metadata identifies the snapshot digest, database ID, last applied slot, and next configuration ID. Snapshot creation may run concurrently with new transactions by reading a stable SQLite snapshot, but installation and epoch transition must obey the decided boundary.

Keep the prior snapshot and covered journal until the new snapshot is complete, directory-synced, checksum-verified, and recoverable by a quorum. Retain at least one fallback generation. Add a background payload garbage collector only after it can prove that no live epoch, accepted descriptor, or retained snapshot references the object.

A timeout, lease expiry, or ballot change is not such a proof. If acceptor a has durably accepted (b, s, v), a later Phase-1 quorum containing a may find that vote to be its highest report and must re-propose v even when v was never known chosen. Deleting v's payload merely because it is old would make a legal Paxos recovery impossible. A payload is locally collectible only after a durable reachability scan proves all of the following:

1. no retained accepted or committed descriptor names it;
2. no retained snapshot/journal suffix or client result names it;
3. no proposal, recovery, or transfer is in flight; and

4. snapshot/epoch coverage permits deletion of the storage generation.

An orphan temporary or content object that was never durably referenced may be removed after an age threshold, but age is only a resource-management condition after the reference proof. When a higher-ballot accept durably replaces a lower-ballot value for the same local slot, the old payload becomes eligible only if the remaining conditions also hold.

4.5. Bounded client sessions and retry semantics

(Implemented mechanism: sessions live in a replicated `__zaxon_sessions` table inside the SQLite state machine itself. The session row update joins the user's statements in one SQLite transaction, so the captured frame batch carries the sequence advancement and bounded result atomically with the data change; replay and rebuild reproduce both together. Ordered expiry remains future work.)

Do not retain one result for every request forever. The replicated state machine allocates a never-reused `session_id` from a durable monotonic counter. A first release session permits one outstanding write and stores a fixed bound:

```
const Session = struct {
    next_sequence: u64,
    last_sequence: ?u64,
    last_result: BoundedResult,
    last_activity_slot: u64,
};
```

For live session c, sequence `next_sequence` is the only new operation that may execute. Its SQLite effects, incremented sequence, and result become one replicated application transition. A duplicate of `last_sequence` returns `last_result`; a lower sequence returns `ResultExpired`; a greater sequence returns `SequenceGap`. Neither error executes SQL. This bounds state to $O(\text{active sessions})$ for a fixed result limit. A future bounded result window may allow several outstanding requests at $O(\text{active sessions} \times W)$.

Session close or expiry is itself an ordered replicated command. The leader may propose candidates based on policy, but every replica applies the exact chosen set; replicas never expire sessions independently from local wall clocks. An unknown or expired session is rejected and cannot be implicitly recreated. Opening a replacement allocates a strictly newer ID, so a delayed packet for an old session cannot become a new write. Snapshots retain the allocation counter and live session table.

5. API sketch

The public surface should feel closer to SQLite than to a consensus framework:

```
var db = try paxos_sqlite.Node.open(allocator, .{
    .directory = "./data",
    .node_id = 1,
    .members = &.{
        .{ .id = 1, .address = "127.0.0.1:9401" },
        .{ .id = 2, .address = "127.0.0.1:9402" },
        .{ .id = 3, .address = "127.0.0.1:9403" },
    },
});
defer db.close();

try db.exec("create table if not exists items(id integer primary key, v text)", .
```

```
{});
var session = try db.openSession();
try db.execIdempotent(session.id, 1,
    "insert into items(v) values (?)", .{"tea"});

var rows = try db.query(
    "select id, v from items order by id", .{},
    .{ .consistency = .linearizable },
);
defer rows.deinit();
```

The same call with `.members = &.{.{ .id = 1, .address = null }}` opens durable one-node mode and does not create a listening socket. Keep configuration in a single options structure. Do not expose ballots, slots, or protocol effects in the normal SQL API; expose them through diagnostics.

6. CLI and process interaction

The library remains the product. The `zaxon` executable is built from the same public library and serves two use cases:

1. applications that embed `Node` use the CLI as a remote SQL and operations client;
2. applications that prefer a database process run `zaxon serve`, which is a thin reference host rather than a separate implementation.

The CLI must never open or modify a live node's journal, payload, or snapshot files directly. Online commands use authenticated client/admin RPC. Explicitly named offline recovery commands require the node to be stopped, acquire the data-directory lock, make a backup first, and default to inspection only.

6.1. Command surface

Command	Purpose	Default output
`serve`	Run one embedded node as a standalone process using the same `Node.open` path as an embedding application.	Structured startup log.
`sql`	Open an interactive SQLite shell through the cluster client.	Human-readable table.
`exec`	Execute SQL or a file; supports an explicit session ID and monotonic sequence for retry-safe writes.	Affected rows and status.
`query`	Run a read with `linearizable` or `stale` consistency.	Table, JSON, JSONL, CSV, or TSV.
`status`	Show node/cluster health, leader, ballot, configuration, committed and applied positions, queue depth, and snapshot.	Table; `--json` for automation.
`leader`	Resolve the current leader from one or more seed endpoints.	Endpoint and node ID.

Command	Purpose	Default output
`members`	List voters and reachability; add/remove follows the later reconfiguration milestone.	Table.
`snapshot`	Request and wait for a safe online checkpoint.	Snapshot ID, digest, slot, and coverage.
`backup`	Stream a consistent logical or snapshot backup to a new path.	Progress and manifest.
`integrity-check`	Run SQLite and replicated-state checks without mutating state.	Pass/fail plus findings.
`wait`	Wait for leader, quorum, applied slot, or healthy state with a deadline.	Final observed state.

`serve` accepts configuration from flags, one JSON file (`--config <path>` or `ZAXON_CONFIG`), or `ZAXON_*` environment variables; flags override environment values, which override file values. Secrets may be named by file or secret-provider handle and must not appear in `status`, process arguments in recommended examples, or logs.

6.2. CLI examples

One durable process:

```
zaxon serve \
  --node-id 1 \
  --data ./data-1 \
  --listen 127.0.0.1:9401 \
  --members 1=127.0.0.1:9401

zaxon sql --endpoint 127.0.0.1:9401
zaxon exec --endpoint 127.0.0.1:9401 \
  --sql 'insert into items(v) values (?)' --param tea
```

Three processes use the same immutable bootstrap membership and separate data directories:

```
zaxon serve --node-id 1 --data ./n1 --listen 127.0.0.1:9401 \
  --members 1=127.0.0.1:9401,2=127.0.0.1:9402,3=127.0.0.1:9403
zaxon serve --node-id 2 --data ./n2 --listen 127.0.0.1:9402 \
  --members 1=127.0.0.1:9401,2=127.0.0.1:9402,3=127.0.0.1:9403
zaxon serve --node-id 3 --data ./n3 --listen 127.0.0.1:9403 \
  --members 1=127.0.0.1:9401,2=127.0.0.1:9402,3=127.0.0.1:9403

zaxon wait --seed 127.0.0.1:9401 --for healthy --timeout 10s
zaxon status --seed 127.0.0.1:9401 --json
zaxon sql --seed 127.0.0.1:9401,127.0.0.1:9402
```

The interactive shell supports a small documented dot-command set: `.leader`, `.status`, `.consistency`, `.timeout`, `.mode`, `.headers`, `.read FILE`, `.backup PATH`, and `.quit`. SQL syntax and transaction commands remain SQLite's. Dot commands do not masquerade as SQL.

6.3. CLI routing and failure semantics

The CLI may connect to any seed. For a write or linearizable read, the contacted node either serves as leader, securely proxies the session to the leader, or returns a typed leader hint. The

client follows bounded redirects and preserves the session ID and sequence across reconnects. It never automatically retries a write under a new session or sequence.

Exit codes distinguish SQL failure, unavailable quorum, deadline, authentication, configuration, ambiguous write result, and integrity failure. `--json` writes one versioned result object to stdout; diagnostics go to stderr. This makes the CLI suitable both for a human shell and for health scripts without parsing decorative text.

7. Write concurrency and horizontal scaling

Direct answer
A cluster can accept concurrent write requests through every node, but one SQLite database still has one logical writer and one Paxos leader ordering commits. Multiple nodes provide high availability and more client ingress; they do not create independent writers for the same database. Horizontal write scaling requires sharding into multiple SQLite databases, each with its own Paxos group and preferably a different leader.

7.1. What "multi-write" means

Capability	Support	Semantics
Many concurrent client writers	Yes	All nodes accept ingress; requests are routed to one bounded leader-side writer queue.
Several requests in one replicated batch	Yes	The leader executes them serially under SQLite rules, preserves each transaction boundary/result, and chooses one ordered batch value.
Several consensus/storage operations in progress	Limited	Payload I/O, peer replication, apply, replies, and the next safe batch may overlap within bounded queues.
Two leaders writing one database	No	This would violate the single ordered state-machine model and SQLite's one-writer semantics.
Followers independently executing writes	No	Followers durably vote and apply the leader-produced WAL/page frames in chosen order.
Multiple independent databases/shards	Planned	Each shard has its own writer, log, quorum, snapshot, and failure domain.

SQLite permits concurrent readers but serializes writers. The product can hide consensus and sync latency with bounded batching: collect a small number of queued transactions for a short maximum delay, execute them serially on a speculative branch, and place all transaction boundaries/results in one payload and one Paxos slot. An individual transaction remains atomic. If the batch is not chosen, none of its transactions becomes visible; if chosen, all are applied in their captured order.

Do not initially pipeline state-dependent WAL batches in separate unchosen slots. A frame batch for slot $s + 1$ was produced from the state after slot s. Independent Paxos recovery could otherwise preserve the later slot while selecting a different earlier value, invalidating the page dependency. The safe first release waits until the predecessor is chosen before building the next slot, while overlapping non-ordering work and batching multiple transactions inside a slot. Any

later dependency-aware pipeline must make the parent/result chain hashes part of the replicated invariant and prove recovery cannot choose an incompatible chain.

A hash chain detects; it does not choose

Requiring only a durable local copy of slot s is insufficient. Suppose x is accepted but not chosen at s, and a WAL value y derived from x is accepted or chosen at $s+1$. A later Phase 1 may legally choose $x' \neq x$ at s while preserving y at $s+1$. `base_chain_hash(y)` detects the mismatch but cannot revoke the already chosen y; Paxos values cannot be "gracefully failed" after choice.

The first-release invariant is therefore $\text{propose}(s+1, y) \Rightarrow \text{chosen}(s, x) \wedge \text{locally-applied}(s, x) \wedge \text{parent}(y) = \text{chain}(x)$ for data-dependent predecessors. A future pipeline needs a new proof that consensus can choose only a compatible chain (or must replicate state-independent logical operations). Adding hashes alone does not provide Raft-style prefix semantics.

7.2. What adding nodes improves

Topology	Benefit	Throughput effect
One voter	Simple durable embedded SQLite; no network availability.	Lowest replication latency; no failover.
Three voters	Continues with one voter unavailable.	Default production shape; one remote durable vote is normally needed in addition to leader.
Five voters	Continues with two voters unavailable under majority quorums.	Usually more replication work and no extra write execution capacity.
Read replicas	More stale/local read capacity and geographic copies.	Does not increase single-database write capacity; voter-certified learners implement this without quorum inflation.
Many shards	Independent writers and failure/size domains.	Can scale aggregate writes when keys route cleanly; no cross-shard transaction in the first design.

Linearizable reads route to the leader. Bring-up barriers do not scale; the stable Paxos read fence avoids a log write but still consumes a read-quorum round, which concurrent reads may coalesce. Explicit stale snapshot reads can scale across followers. A later lease may remove per-read quorum traffic only under a separate clock-and-grant proof.

7.3. Horizontal scale roadmap

1. Completed: make one runtime-role cluster correct, bounded, and observable, including one through nine voters and non-voting learners.
2. Add multiple named databases in one process, each with an independent `ReplicatedLog`, SQLite state machine, directories, limits, and metrics.
3. Introduce a deterministic shard router from database/key to Paxos group; spread group leadership across physical nodes.

4. Add shard placement and movement using snapshot plus suffix transfer.
5. Completed: add bounded non-voting learners, role-aware standbys and read replicas, and stateless gateways.
6. Keep cross-shard transactions out of scope until there is a separate atomic commit design; do not imply that Paxos per shard provides them.

This scale-out model is similar operationally to partitioned databases: one shard remains serial, while independent shards write concurrently. Adding replicas to a shard scales durability and reads, not its SQLite writer.

8. Incorporation review of the existing Paxos library

The current library is a strong protocol core but deliberately not a database host. Most work belongs in the new `sqlite/` layer, preserving `src/` as a deterministic, bounded, I/O-free package.

8.1. Capability mapping

Existing capability	How Zaxonlite uses it	Work still required
`ReplicatedLog(Command, options)`	One ordered command/checkpoint log per database or shard.	Host lifecycle, command codec, epoch selection, and storage sizing.
`Node.append`	Propose a transaction-batch descriptor or read barrier at the leader.	Writer queue, session-sequence validation, payload preparation, and `NotLeader` routing.
`Node.appendBatch`	Potentially protocol no-ops or future independent commands.	Do not use for dependent WAL transactions until a chain-safety design exists; it also does not provide wire coalescing.
`Effects.writesSlice`	`journal` serializes and syncs every record in the emitted order.	Checksummed format, group-sync scheduler, I/O failure policy, torn-write recovery.
`confirmWritesDurable` then `messagesSlice`	Central host gate before peer send.	One event-loop path that makes bypass impossible and trace events that tests can audit.
`committedSlice`	Apply only its contiguous command prefix to SQLite.	Delay application until the associated commit effects are synced; persist `applied_slot` and results.
`DurableState.apply` and `Node.restore`	Reconstruct protocol state by journal replay after restart.	Record codec/versioning, snapshot boundary, corruption policy, payload presence validation.
`currentLeader`, `NotLeader`	Populate leader hints and route CLI/client sessions.	Authenticated proxy/redirect protocol and bounded retry.
`tick`, heartbeat, resend, `reconnected`	Driven by the host event loop and monotonic timers.	Clock ownership, randomized election staggering, socket reconnect, queue limits.
Current value-bearing `step` behavior	`promise`, `accept`, and `commit` envelopes can immediately	Materialize every referenced payload before `step`; later add staged recovery so only highest-

Existing capability	How Zaxonlite uses it	Work still required
	produce durable/recovery effects.	ballot winners must be fetched before activation.
Current heartbeat	Maintains leader hints and catch-up but sends no affirmative nonce ACK.	The product host supplies a bounded fresh-ID challenge/ACK with distinct-member counting; periodic heartbeat receipt is not a linearizable read proof. A core wrapper is optional.
`requestCatchUp`	Repair a lagging node within a retained epoch.	Payload transfer, snapshot fallback, rate limits, applied-state catch-up.
`checkpoint`, stop signs, `initFromStop`	Seal an epoch after a named SQLite snapshot and start the next.	Quorum snapshot coverage, transfer, installation, retention, and GC.
Flexible quorum options	Possible advanced stable-cluster policy.	Ship majority defaults first; validate intersections and failure behavior before exposing CLI flags.
Compile-time bounds	Explicit memory/backpressure envelope.	Epoch rollover before `max_entries`, small fixed descriptors, and external bounded payload storage.

8.2. Gaps that must not be mistaken for library defects

The protocol intentionally owns no files, codecs, sockets, clocks, threads, SQLite state, authentication, client sessions, or snapshot transfer. Paxos SQLite must implement these as a host. The existing in-memory three-node tests prove selected protocol schedules; they do not prove filesystem sync semantics or three operating-system processes.

`DurableState` retains accepted and committed `Value` instances in fixed arrays through an epoch. That is appropriate for a bounded protocol core but makes a small descriptor plus external payload store mandatory. `max_entries` is a hard capacity, so checkpoint/epoch rollover is required product behavior, not an optional maintenance task.

The current catch-up path repairs committed entries represented in the live epoch. A new or far-behind database node also needs snapshot discovery, authenticated bulk transfer, payload fetching, and post-snapshot suffix replay. Stop signs describe the safe epoch transition but do not perform those host operations.

8.3. Small general-purpose core additions to evaluate

Avoid changing consensus semantics for the SQLite product. Consider only general diagnostics or bounded mechanisms with standalone protocol tests:

1. a public read-only role/status snapshot so the admin API does not inspect internal fields;
2. counters or iterators needed to report accepted, decided, and capacity state without copying full values;
3. an explicit reserved-capacity query so the host checkpoints before `SlotLimitReached`;
4. a generic non-voting learner role only if read replicas become a committed roadmap item;

5. a staged Phase-1 activation API that exposes recovered winning values before `maybeBecomeLeader` emits Phase-2 effects;
6. a nonce-bearing read-fence request/ACK that counts a Phase-1 quorum only while the acceptors' durable promise equals the proposer's ballot;
7. measured wire-message batching only as a protocol-wide feature, distinct from `appendBatch`'s effect batching.

Payload transfer, SQLite batching, client sessions, CLI RPC, journals, and snapshots remain host concerns and should not enter `src/protocol.zig`.

8.4. Incorporation sequence

1. Add `sqlite/command.zig` with a tiny fixed descriptor and fuzzed codec.
2. Build `sqlite/journal.zig` around `Effects` and test durable-claim-before-send with an injectable disk and send trace.
3. Restore `ReplicatedLog` exclusively by replay through `DurableState.apply`; never serialize raw Zig struct memory.
4. Complete the one-node SQLite frame spike and state-machine apply path.
5. Add payload offer/storage-ACK gates around outbound accepts and materialize all value-bearing inbound envelopes before `step`, including Phase-1 recovery; then evaluate staged recovery to avoid fetching losing candidates.
6. Add TCP transport and the real three-process fixture.
7. Add client/admin RPC, then build the CLI entirely on those protocols.
8. Add snapshot/epoch rollover before load and scale testing.

9. Paxos safety proof for Zaxonlite

This section fixes the consensus identity of the product: the SQL surface and embedded deployment are inspired by dqlite, but ordering, recovery, quorum selection, and durability use this repository's Multi-Paxos protocol. SQLite does not participate in leader election and no Raft algorithm is introduced in the host.

Claim proved here

Under the assumptions below, two correct Zaxonlite nodes cannot commit different transaction descriptors for the same configuration and log slot. If the product-level payload and application invariants also hold, they cannot apply different SQLite transactions at that slot or diverge in their applied database prefix.

9.1. Why the design is Multi-Paxos rather than Raft

Paxos may have several competing proposers without violating safety. A stable deployment nevertheless selects one **distinguished proposer** so Phase 1 is amortized and the system makes progress. In this code that proposer is called the leader. Any member can replace it by completing Phase 1 with a higher ballot; leadership itself is not durable consensus state.

Mechanism	Zaxonlite	Raft mechanism not adopted
Durable authority	`promised` ballot and per-slot `accepted` ballot/value.	`currentTerm`, one `votedFor` value per term, and an election-vote record.

Mechanism	Zaxonlite	Raft mechanism not adopted
Leader replacement	Phase-1 quorum reports accepted values; proposer recovers the highest ballot independently for each slot.	Candidate must already have a sufficiently up-to-date prefix to win election.
Conflict handling	A higher ballot re-proposes the Paxos-safe value for the slot.	Leader uses `prevLogIndex`/`prevLogTerm` and deletes a follower's conflicting suffix.
Commit rule	One value is chosen when a Phase-2 quorum durably accepts the same ballot and slot.	Majority replication plus Raft's current-term commitment restriction and prefix implications.
Quorum policy	Phase-1 and Phase-2 quorum sizes may differ if every pair intersects.	Election and replication normally use majority quorums.
Normal operation	A prepared proposer assigns slots and skips repeated Phase 1.	A strong leader is the only source of log entries.

The CLI may accept a request at any node and forward it to the prepared proposer. That is client routing, not Raft consensus. Similarly, the words `leader`, `follower`, heartbeat, and timeout name common liveness mechanisms; they do not determine the safety algorithm.

9.2. System model and assumptions

Fix one configuration epoch with member set A and $N = |A|$. The proof is per slot; let s be an arbitrary slot. It assumes:

1. **Crash faults, not Byzantine faults.** A process may stop, restart, lose volatile state, duplicate messages, and delay messages indefinitely. A correct process does not forge a protocol message or lie about a durable vote. Peer identity is authenticated by the product transport.
2. **Durable acceptor state.** `promised`, `accepted`, and `committed` journal records survive restart. Every transition persists and syncs `Effects.writesSlice()` before it exposes any dependent `Effects.messagesSlice()`.
3. **Unique, totally ordered ballots.** A ballot is the tuple `(round, priority, node_id)`. A node owns its ballots and never deliberately issues two different values for the same `(ballot, slot)`.
4. **Cross-phase quorum intersection.** Every allowed Phase-1 quorum intersects every allowed Phase-2 quorum. The current uniform-size policy enforces $q_1 + q_2 > N$, which implies this intersection by counting.
5. **Complete Phase-1 replies.** A promise is counted only after all of that member's accepted entries and its `promise_done` marker have arrived.
6. **Self-contained decided value.** Paxos decides a fixed-size command descriptor, never a process-local pointer. Product rules below bind the descriptor to its immutable payload bytes.

Safety does not assume accurate clocks, a unique leader, FIFO delivery, or eventual message delivery. Those assumptions affect progress only.

9.3. State and definitions

For acceptor a:

1. P_a is its greatest promised ballot;

2. $V_{a(s)}$ is either no vote or its latest accepted pair (b, v) for slot s.

A proposal (b, s, v) is **chosen** when every member of some Phase-2 quorum has durably accepted that ballot, slot, and value. A `commit` record is durable knowledge of this already-established fact; the leader's commit broadcast does not make the value chosen.

The target agreement property is:

$$\text{forall s, v, w : “chosen“(s, v) and “chosen“(s, w) => v = w}$$

9.4. Preserved invariants

The proof relies on five invariants.

1. **Promise monotonicity.** P_a never decreases. After promising ballot b, acceptor a never accepts a ballot smaller than b.
2. **Vote durability.** An `accepted` reply is evidence that the corresponding vote survived a crash at the sender before that reply was transmitted.
3. **One value per ballot and slot.** All Phase-2 requests for fixed (b, s) contain one value. Duplicate requests are harmless; a conflicting duplicate is rejected as corruption.
4. **Phase-1 recovery rule.** After a complete Phase-1 quorum, the proposer uses the value belonging to the greatest accepted ballot reported for s. It is free to use a new value only when no quorum member reports a vote.
5. **Cross-phase intersection.** If R is a Phase-1 quorum and W is a Phase-2 quorum, then R and W share at least one acceptor.

For uniform quorum sizes the fifth invariant follows immediately. If R and W were disjoint, they would contain $q_1 + q_2$ different members. But $q_1 + q_2 > N$, while the configuration has only N members, a contradiction. Notice that two Phase-2 quorums need not intersect in Flexible Paxos; the intersection needed by the induction is between the earlier choosing quorum and a later recovery quorum.

9.5. Agreement theorem

Theorem. If value v is chosen for slot s in ballot b, every proposal issued for s in any higher ballot contains v. Consequently, two different values cannot both be chosen for s.

Proof. Use induction over the total ballot order. Suppose v was chosen by Phase-2 quorum W in ballot b, and consider a higher ballot b' that issues a proposal after completing Phase 1 with quorum R.

Because R intersects W, choose an acceptor a in both. Before promising b', acceptor a had accepted (b, v) or later replaced it with a vote in some ballot c satisfying $b < c < b'$. In the first case its relevant value is v. In the second case the induction hypothesis says the proposal in c also contained v.

The proposer for b' selects the value attached to the greatest accepted ballot reported by all of R. If some report has a ballot greater than a's report, that ballot is still below b' and, by the same induction hypothesis, also carries v. Thus the greatest report carries v, so the Phase-1 recovery rule forces ballot b' to propose v.

Therefore every higher proposal carries v. Every chosen value first appears in some proposal, so a value w chosen in a later ballot must equal v. Within the same ballot, the one-value invariant already gives equality. Hence agreement holds. □

9.6. Mapping the theorem to `paxos-zig`

Proof obligation	Library mechanism	Product obligation
Total, unique ballots	`Ballot.order`; `startCampaign` chooses a round above local, promised, and observed rounds and includes node identity.	Persist identity; reject duplicate node IDs; authenticate envelope sender.
Promise monotonicity	`onPrepare` rejects ballots below `durable.promised`; `DurableState.apply` rejects regression.	Journal and sync the emitted promise before sending its replies.
Complete Phase 1	`promise`, per-member counts, and `promise_done`; leadership waits for `readQuorum()` complete members.	Codec preserves counts and message identity; transport bounds cannot silently discard part of a reply.
Highest-vote selection	`onPromise` retains the greatest accepted ballot per slot; `maybeBecomeLeader` re-drives it and fills true gaps with no-op.	Fetch and verify the payload for every recovered descriptor before the recovered accept is delivered or sent.
One value per ballot/slot	`sendAccept` and `onAccept` reject conflicting values; `proposals[slot]` fixes the leader's value.	Never mutate payload bytes stored under a descriptor hash.
Phase-2 choice	`onAccepted` counts distinct members in a bit set and calls `recordCommit` only at `writeQuorum()`.	An acceptor is counted only after its protocol record and referenced payload are synced.
Cross-phase intersection	`Membership.init` validates `read_quorum_size + write_quorum_size > member_count`.	Expose only validated configurations; majority quorums are the first-release default.
Crash recovery	`DurableState.apply`, `Node.restore`, and a new campaign discard volatile leadership and reconstruct acceptor state.	Replay framed records in order; halt on durable-prefix corruption.

The library's `role == .leader` check is a Multi-Paxos optimization and a liveness policy. It does not add a Raft election invariant. Any node may call `campaign`; after a higher Phase-1 quorum it becomes the prepared proposer and the older proposer is rejected by durable promises.

9.7. Lifting one-slot agreement to the SQLite log

The theorem applies independently to every slot in one epoch. Therefore all correct nodes that know a decision at slot s know the same command descriptor. `committedSlice()` releases only the contiguous prefix, so if nodes apply released entries in increasing slot order, they apply the same descriptor sequence.

Consensus agreement on a descriptor is necessary but not sufficient for a database. Zaxonlite must preserve six additional invariants:

1. **Payload availability.** Before an acceptor passes an `accept` descriptor into the Paxos core, the referenced payload is present, checksum-verified, and durable. Hence a choosing quorum also durably covers the payload bytes.
2. **Content identity.** `payload_hash` names one immutable byte string. A mismatch is corruption and the node stops voting. The proof assumes the selected hash function's collision resistance; an on-disk length and checksum catch accidental corruption independently.
3. **Chain compatibility.** A transaction batch records `base_data_slot`, `base_chain_hash`, and `result_chain_hash`. The state machine applies it only to the named history chain. The hashes are cumulative descriptor identities, not whole-database hashes.
4. **Dependency-before-proposal.** A proposer constructs a state-dependent batch only after its parent transaction slot is known chosen and locally applied. This prevents an incompatible batch from becoming a Paxos value; merely checking its hash during application would be too late.
5. **Deterministic application.** Given the same SQLite snapshot and exact WAL frame bytes, each supported SQLite/VFS implementation produces the same database transition or reports corruption. Version, page size, database ID, and feature compatibility are validated before apply. The chain hash does not substitute for these checks.
6. **Exactly-once prefix application.** The durable applied position and request session records advance atomically with the SQLite transaction effects, or the SQLite image remains explicitly rebuildable from snapshot plus committed log.

Use induction on the applied slot to prove database agreement. The base case is a checksum-identified snapshot shared by the epoch. For the induction step, assume two correct nodes have the same canonical SQLite state and chain identity through slot $s-1$. Paxos agreement gives them the same descriptor at s; content identity gives the same payload; chain compatibility validates the same predecessor; deterministic application gives the same next SQLite state; and the $O(1)$ chain formula gives the same next chain identity. Thus their states agree through s.

Why dependent WAL slots are not freely pipelined

Paxos agreement chooses one value independently per slot. It does not prove that a frame batch chosen at slot $s+1$ was generated from whichever value recovery ultimately chooses at slot s. The first release therefore batches several SQLite transactions inside one Paxos value and waits for the predecessor to be chosen and locally applied before constructing the next dependent value. A future cross-slot pipeline needs a separate chain-selection invariant. A cryptographic parent link is necessary for detection but is not sufficient for prevention because Paxos choice is final.

9.8. Successful-write durability theorem

The API may report success for request r only after:

1. the payload containing r is durable wherever a counted vote is recorded;
2. a Phase-2 quorum has accepted the descriptor;
3. the leader's local commit knowledge is journaled and synced;
4. the committed prefix through that slot is applied to SQLite; and
5. the session's sequence advancement and bounded result are durable with the application transition.

At step 2, the Paxos theorem makes the descriptor immutable across all future ballots. A future leader's Phase-1 quorum intersects the choosing quorum and is forced to recover it. The payload-before-vote rule makes the corresponding bytes recoverable from that quorum. Steps 3–5 make

the responding node able to return the same result after its own restart. Therefore a successful request cannot be replaced or acknowledged and later forgotten under the stated quorum and storage assumptions.

A timeout before success remains ambiguous: the command may already be chosen. Retrying the same live-session sequence is safe only when sequence state and the bounded result are part of the replicated state machine, not merely an in-memory response cache. Expired/older sequences are rejected rather than executed.

9.9. Linearizable read arguments

9.9.1. Committed-barrier reference path

The default read first routes to a prepared proposer, appends a unique `read_barrier`, waits until that slot is committed and locally applied, and only then opens the SQLite read transaction.

Consider a write whose success response completed before the read began. Its descriptor was already chosen. If leadership changed, Phase 1 forces the new proposer to preserve that chosen value. The barrier obtains a later available slot in the recovered log. Because application releases only a contiguous prefix, applying the barrier implies that every earlier chosen slot, including the acknowledged write, has been applied locally. The subsequent SQLite snapshot therefore includes that write.

This proof requires request ordering at the client/leader boundary: a stale follower cannot manufacture a barrier locally, and the read must not execute before the barrier's committed effect is synced and applied.

9.9.2. Paxos quorum read-fence path

Let R be the acceptors returning fresh nonce ACKs for ballot b, with $|R| \geq q_1$. Each ACK is emitted only when the acceptor's durable promise is exactly b. Let k be the proposer's decided prefix captured before it sent the challenge. It serves the SQLite read only after receiving all ACKs and applying locally through k.

Take any write w whose success completed before this read was invoked. If w was completed by this same prepared proposer before it captured k, the successful-write rule and contiguous application put w at or before k. If w was chosen by another/higher ballot, let W be its Phase-2 quorum. Because R is a Phase-1 quorum and $q_1 + q_2 > N$, choose $a \in R \cap W$. Before voting for the higher-ballot write, a durably promised at least that higher ballot, so it cannot subsequently ACK the older ballot b. This contradicts $a \in R$. Therefore an older proposer cannot complete the fence after a previously completed higher-ballot write.

A higher-ballot write concurrent with the read may occur after the fence's quorum point; linearizability may order the read before that write. Thus the snapshot opened after local apply through k contains every write that must precede it. The proof needs fresh nonces, distinct-member counting, equality with durable `promised`, read/write quorum intersection, and request-to-fence ordering. The current one-way periodic heartbeat satisfies none of the ACK requirements. Reusing a completed fence for a later-arriving read requires a separate lease proof and is forbidden by default.

9.10. Snapshot and reconfiguration proof obligation

A snapshot manifest binds `(database_id, configuration_id, applied_slot, chain_hash, snapshot_content_hash, session_hash)`. The old epoch decides a stop sign containing that manifest identity. `ReplicatedLog` prevents commands after a pending or decided

stop sign, so the epoch has one sealed suffix. A node starts the next epoch only from the exact verified snapshot named by the decided stop sign and then applies the new epoch's log from slot one.

The intended cross-epoch invariant is that every correct node beginning epoch $e + 1$ starts from the same state produced by the decided prefix of epoch e. The stop-sign mechanism establishes the ordering boundary, but snapshot quorum coverage, transfer, atomic installation, and garbage collection are host operations. They require their own TLA+ extension and crash tests before dynamic reconfiguration is claimed as proved.

9.11. Safety versus liveness

Agreement remains true with no leader or several nodes that temporarily believe they are leader: higher promises reject stale ballots and Phase 1 preserves chosen values. Progress is conditional. In a fully asynchronous system, a deterministic failure detector cannot guarantee that competing proposers stop preempting one another. The implementation therefore uses ticks, heartbeats, priority, and an eventual-leader assumption.

Once one proposer with a sufficiently high ballot can communicate with the required quorums without continual preemption, new values make progress. This is the standard Multi-Paxos liveness condition, not Raft's safety-dependent leader election restriction.

9.12. Formal evidence and its boundary

`specs/Paxos.tla` models durable `promised`, `accepted`, and `committed` state plus a monotonic message set. Its action mapping is Prepare/Promise/Accept/Vote/ Decide/Learn to the corresponding Zig transitions. On 20 July 2026, the checked configuration was independently rerun with TLC 2.14. The exact state counts for the upgraded voter-plus-learner fixture are recorded in `specs/README.md`:

1. 85,515,700 states generated and 3,986,355 distinct states;
2. complete breadth-first search to depth 20 with zero queued states;
3. no violation of Agreement, CommitUniqueness, PromisedDominatesVotes, or Validity;
4. no learner emitted a promise or accepted vote.

The checked configuration has three voters, two learners, one slot, one client value plus no-op, and one round owned by each voter. This is bounded exhaustive evidence, not an unbounded theorem and not a mechanical refinement proof of the Zig program.

Obligation	Current evidence	Required before stable claim
Abstract Paxos agreement	Inductive argument above; bounded TLC.	Encode and check the invariant proof with TLAPS or an equivalent proof assistant.
Zig transition conformance	Manual action mapping, unit tests, simulator.	Maintain a line-level refinement map; generate conformance traces; review every safety-relevant change.
Flexible quorums	Counting proof; validation tests.	TLC configurations with representative asymmetric Phase-1/Phase-2 sizes.
Durability ordering	`Effects` debug assertion and durable benchmark.	Real journal crash tests and a production path that cannot send or apply before successful sync.

Obligation	Current evidence	Required before stable claim
Payload-before-vote	Design invariant only.	Three-process failpoints, storage-ACK gating, accept-before-payload bounds, Phase-1 recovery fetch, corruption tests, and trace oracle.
SQLite state agreement	Induction above; Phase-0 design.	Frame-equivalence property tests across supported SQLite versions and restart schedules.
Linearizable reads	Barrier and quorum-fence arguments above.	Core nonce/ACK tests plus black-box concurrent history tests including leader changes and partitions.
Bounded retry state	Session transition argument above.	Sequence duplicate/gap/expiry tests, leader-before-reply crash, and snapshot-size bound under churn.
Snapshot epoch transition	Stop-sign library tests.	Formal host model, snapshot-transfer tests, atomic-install and GC crash matrix.

Permitted release wording
Today it is accurate to say: "the protocol core implements the standard Multi-Paxos safety rule under crash faults, durable write-before-send, unique ballots, authenticated members, and intersecting quorums; a bounded TLA+ model and deterministic simulator find no invariant violation." It is not yet accurate to call the complete Zaxonlite product formally verified.

10. rqlite incorporation review

rqlite is a useful product benchmark, not a consensus template: it replicates SQLite through Raft, while Zaxonlite must preserve Paxos ballots, quorum intersection, and Phase-1 recovery. The following review uses rqlite's current official feature, clustering, and read-only-node documentation.

rqlite capability	Zaxonlite decision	Evidence / limit
Voting HA cluster	Implemented with Multi-Paxos data voters and optional acceptor-only witnesses.	Three-process election, loss, catch-up, total restart, and leader-death-before-reply tests.
Read-only non-voter	Implemented as voter-certified `standby` and `read-replica` learners.	Local `any` reads, optional `freshness_ms`, lag rejection, and a real six-node role test.
Any-node client access	Implemented by leader hints, transparent client redirection, and stateless gateways.	Learners remember the voter that certified their latest chosen entry; no learner becomes a proposer.
Nodes/discovery API	The `members` RPC exposes every configured address, role, voter capability, self, and leader.	Bootstrap remains explicit and static; DNS/Consul/etcd discovery and automatic join are roadmap work.

rqlite capability	Zaxonlite decision	Evidence / limit
Dynamic join/removal	Not copied into the first release.	Safe voter replacement requires stop-sign epoch transition, learner catch-up proof, and operational fencing; timeout-based voter reaping would be unsafe.
Hot backup	Implemented as authenticated, digest-verified streaming to an atomically installed plain SQLite file.	Cloud scheduling and restoring an external SQLite image into a live cluster remain separate product work.
TLS and authorization	Mutual TLS 1.3, per-node certificate binding, node-ID-pinned mTLS redirects, seed-only PSK redirects, and optional in-TLS PSK/HMAC are implemented.	A deliberately configured issuer automates short-lived, single-use token/CSR enrollment for nodes already in the static registry; initial CA and issuer credentials remain operator-provisioned. Revocation is a node-ID denylist. There are intentionally no database-user/RPC roles: the embedding application is the one authority.
HTTP, CDC, cloud backup	Not required for the embedded first release.	The Zig/C APIs and compact TCP JSON protocol remain the supported surface.

This incorporates the correctness-relevant lessons—non-voters, bounded stale reads, transparent routing, observable roles, and online backup—without wholesale adoption of Raft's leader/log rules. It also keeps the operational limit honest: rqlite itself recommends small odd voting groups because large consensus groups add coordination cost; Zaxonlite similarly bounds voters while allowing runtime-sized non-voters.

11. Better-than-dqlite feature targets

The baseline inspiration is dqlite: embedded SQLite, a network client/server path between application nodes, frame/page replication, and quorum durability. The official architecture and replication notes are linked in the references. The product should differentiate only where tests establish a real property.

Area	Target	Evidence required
Read consistency	Linearizable reads by default; explicit stale local mode.	History test that forbids a read from missing an earlier acknowledged write.
Write concurrency	Concurrent intake, bounded SQLite writer queue, multi-transaction values, and group sync without unsafe dependent slots.	Throughput/latency curves by batch size, queue delay, and transaction size.

Area	Target	Evidence required
Quorums	Majority defaults; optional validated flexible read/write quorum configurations.	Intersection validation plus failover tests for every supported topology.
Retry safety	Replicated sessions, monotonic sequences, bounded retained results, and exactly-once apply for retries.	Kill leader before reply, retry, and observe one row/result; churn sessions without unbounded state.
Recovery clarity	Journal is authoritative; SQLite is reproducible from a snapshot plus suffix.	Delete materialized image, rebuild, and compare logical dump/hash.
Portability	Portable threaded/blocking backend first; platform-specific async backends optional.	CI on every claimed OS and filesystem.
Operations	Online backup, integrity check, snapshot, and health APIs with no external daemon.	Documented drills with machine-checkable results.

Comparison boundary
Do not advertise a dqlite or rqlite advantage until the benchmark uses equivalent durability, transaction sizes, concurrency, network topology, SQLite options, hardware, warm-up, and failure behavior. Report p50/p95/p99 commit latency beside throughput and fsync counts; a throughput win that hides unbounded latency is not a product win.

12. Delivery plan

12.1. Implementation status (21 July 2026, first-release audit)

Status: first-release scope implemented
The one-node product, runtime-role cluster, transport-owning Zig and C facades, authenticated process host, CLI, recovery paths, and deterministic adverse-network schedules are implemented and tested. The explicitly deferred 10,000 crash schedules, 100 consecutive cluster runs, 1 GiB recovery, and publication of Linux-only dqlite numbers remain post-release stress work. A local installed-rqlite comparison is recorded separately and is not silently generalized beyond its host.

Implemented and covered by the current automated suites:

1. WAL-hook capture and offline page apply, including DDL, DML, triggers, BLOBs, rollback/savepoint, and nondeterministic SQL, with a byte-identical rebuild oracle;
2. fixed descriptors, cumulative chain validation, a checksummed/fsynced Paxos journal, verified content-addressed payloads, and a 64 MiB-minus-73-byte maximum payload policy;
3. journal-authoritative recovery that always discards the working image, restores a digest-verified snapshot, replays the suffix, and resumes both interrupted checkpoint rollover and interrupted snapshot installation;
4. bounded per-peer payload gates; a payload is queued immediately before its dependent envelope on the ordered TCP/TLS stream and drive-flushed before the receiver reads that

envelope. Reordered/missing values remain gated, and `payload_stored` is a readiness cache rather than a serial round trip;

5. one writer, dependency-before-proposal, bounded replicated retry sessions, follower offline apply, leader-change resynchronization, and fatal stop on journal/payload durability failure;
6. a real authenticated three-process test covering wrong-secret rejection, election, writes through every endpoint, one stopped voter, catch-up, leader death before reply, exactly-once retry, snapshot transfer/rollover, image deletion, and total restart;
7. default-linearizable reads using fresh host-level fence IDs, captured slots, exact-ballot replies, and distinct-member quorum counting; explicit `leader` and `any` modes remain available;
8. mandatory mutual TLS 1.3 for production TCP, exact `zaxon-node-<id>` peer certificate binding, node-principal server-certificate validation by clients, optional in-TLS PSK/HMAC, an explicit PSK-only development mode restricted to numeric loopback, hard downgrade rejection, and a failpoint-gated insecure transport used only by deterministic repository tests;
9. prepared bindings and bounded explicit multi-call transaction builders in both Zig and C, plus restart/integrity and compiled C smoke coverage;
10. a transport-owning embedded facade and matching C API; runtime role registry; three data-voter, witness, standby, read-replica, and stateless gateway implementations; voter-certified learner replication; and process-safe connection-handler shutdown;
11. digest-verified remote streamed backup, JSON/environment/CLI configuration precedence, provider-file secrets, static-membership inspection, an offline recovery command, and a standalone format/upgrade contract;
12. a five-boundary real one-process `_exit` crash matrix in addition to the leader-before-reply cluster crash;
13. a real adverse TCP schedule covering deterministic frame loss, semantic duplication, pair reordering, seven-byte partial-frame writes, and delayed journal sync, plus a journal-only 1 MiB payload recovery test;
14. an exact pinned three-node durable-state comparison harness for dqlite 1.18.7/go-dqlite 3.0.4 with equal payloads, warmup, verification, and JSON output, using dqlite's supported default materialization rather than its removed disk-mode option (dqlite itself requires Linux);
15. a three-voter rqlite comparison harness using the system `rqlited` and `rqlite` tools, equal one-row autocommits, no queued writes, CLI membership evidence, strong exact-payload verification, percentiles, and JSON output; the latest TLS/full-sync transport run observed 63 Zaxonlite writes/s at p50 15.94 ms, while the separately recorded same-host rqlite v10.2.7 baseline observed 44.5 writes/s at p50 21.95 ms. These are host-specific executions, not a paired statistical trial or a Paxos-versus-Raft theorem;
16. a deterministic four-client order-processing comparison with 70% linearizable reads, 30% idempotent writes, abrupt follower and leader loss, per-node catch-up, total-cluster restart, accounting/inventory invariants and integrity checks; two local runs retained correct identical state throughout, while the book reports the observed throughput and recovery-time ranges;
17. CLI/RPC operations with Elm-style boundary/explanation/`Hint:` diagnostics, snapshots, integrity inspection, the C ABI, seeded fuzz/soak/benchmark harnesses, and the Zaxonlite book build.

Deferred stress and portability work, not blockers for this first release:

1. 10,000 seeded crash schedules and 100 consecutive cluster runs, explicitly deferred by the release owner;
2. a 1 GiB recovery gate (the current release gate is 1 MiB), long-duration overload measurements, and publication of hardware-specific dqlite numbers;

3. Windows remains unsupported. The explicit first-release matrix in `docs/zds/records/0004-zaxonlite-format.typ` is POSIX-only because parent-directory sync is required and is not silently weakened.

Pipelined dependent WAL writes, group sync, automatic runtime voter reconfiguration, discovery-provider integrations, and sharding remain roadmap work. Static configured voters are not mislabeled dynamic; runtime-sized learners and gateways provide read/routing scale without quorum inflation.

12.2. Phase 0 — prove the risky boundary

SQLite replication spike
Outcome: Link pinned SQLite, capture one committed WAL transaction without a private SQLite patch, apply its frames to a second in-process SQLite image, and show identical schema/data for DDL, DML, BLOB, trigger, rollback, savepoint, and nondeterministic-function cases.
Exit criterion: A written ADR selects the VFS/WAL technique; the spike passes under Address Sanitizer where supported and survives 1,000 randomized transaction cases.

Also define the command/payload wire format, cumulative chain-hash formula, database identity rules, maximum transaction policy, and the exact point at which speculative SQLite changes become visible. These decisions precede public API work.

12.3. Phase 1 — durable one-node vertical slice

Single-process product
Outcome: One application process embeds SQLite and a one-member `ReplicatedLog`, writes a checksummed journal, applies committed transactions, snapshots, closes, and reopens without data loss. The reference CLI can host the node and execute SQL against it.
Exit criterion: `cd zaxonlite && zig build test-single` passes normal restart, SIGKILL failpoints, torn tail recovery, bounded-session retry, and rebuild-from-snapshot tests.

This phase proves the host ordering contract before networking makes failures harder to diagnose. It must use the public Paxos effects API, never mutate `Node.durable` directly.

12.4. Phase 2 — real three-process cluster

Three voters over real transport
Outcome: Three separately spawned OS processes, three data directories, and three TCP endpoints elect a leader, replicate transactions, tolerate one stopped node, catch it up, and recover after all processes restart.
Exit criterion: `cd zaxonlite && zig build test-cluster` passes the mandatory scenario below 100 times without flakes and leaves a reproducible seed/action trace on failure.

Implement peer handshake, version negotiation, payload offer/stream/storage-ACK gating, accept-before-payload rejection, Phase-1 materialization, framing, reconnect, bounded queues, deadlines, and clean shutdown. Start with static three-member bootstrap. Reconfiguration follows only after snapshot transfer is reliable. Add `serve`, `status`, `leader`, and `wait` CLI commands so the integration controller and operators use supported protocols rather than a test-only back door.

12.5. Phase 3 — consistency and client behavior

Usable embedded API
Outcome: Sessions, prepared statements, explicit transactions, leader routing, the committed-barrier reference path, quorum read fences, bounded client sessions and sequences, deadlines, stable error semantics, the interactive SQL shell, scripted output modes, and CLI exit-code contract.
Exit criterion: A black-box history suite validates acknowledged-write visibility, one-writer serializability, ambiguous retries, leader changes, and stale-read labeling.

12.6. Phase 4 — snapshots and operations

Bounded storage and recoverability
Outcome: Online snapshots, epoch checkpointing, payload garbage collection, backup, integrity inspection, metrics, and a documented disaster-recovery command.
Exit criterion: A node with no SQLite image joins from snapshot plus suffix; repeated checkpoint/crash schedules never lose an acknowledged transaction or select an incomplete snapshot.

12.7. Phase 5 — performance and release

Evidence-backed release candidate
Outcome: Bounded transaction batching and group sync are tuned without weakening ordering; quorum read fences replace per-read log appends on the benchmark path; multi-database/shard baselines are measured; the CLI, C ABI, formats, and compatibility policy are documented.
Exit criterion: Soak, fault, upgrade, fuzz, and fair dqlite comparison gates pass on supported platforms. Any performance claim includes the reproducible result file.

13. Mandatory integration tests

13.1. Three-node durability test

This is a real multi-process test, not three `Node` values in one test process. The controller allocates three temporary directories and loopback ports, spawns three copies of a small fixture executable, and drives it through a control API.

```
test "three-node SQLite cluster preserves acknowledged transactions" {
  start N1, N2, N3 with independent directories
  CLI wait for exactly one leader and one configuration ID

  CLI create schema; submit concurrent requests 1..100 through all three nodes
  verify linearizable count = 100 and every session sequence occurs once
  stop one follower
  insert requests 101..150 through the leader; verify count = 150
  restart follower; wait until its applied slot equals the leader
  compare logical database hashes on all three nodes

  arm leader failpoint: after quorum choice, before client reply
  submit session sequence 151; kill occurs; retry that sequence at new leader
  assert session sequence 151 appears exactly once
```

```
  stop all nodes without graceful shutdown
  restart all from the same directories
  wait for election and catch-up
  assert every acknowledged session sequence is present exactly once
  run SQLite integrity_check and compare logical hashes
}
```

The fixture reports its node ID, role, ballot, committed-through, applied-slot, snapshot ID, and last synced journal sequence. The test waits on observable conditions rather than sleeps. Every wait has a deadline and prints all node states plus recent logs when it fails.

13.2. Required crash matrix

Crash point	Expected client result	Recovery oracle
Before payload sync	Failure/unknown; never success.	No vote or applied transaction may reference the incomplete payload.
After payload sync, before local accept sync	Unknown.	Payload may be garbage; transaction is not inferred chosen from payload alone.
After accept append, before sync	Unknown.	Torn tail is truncated and the node never claims that vote survived.
After accept sync, before `accepted` send	Unknown.	Vote survives and is available to a later leader.
After quorum choice, before commit marker	Unknown.	Election recovery must preserve and eventually commit the chosen value.
After chosen, before SQLite apply	Unknown or delayed.	Phase-one recovery reconstructs the choice from durable accepts; replay applies the transaction exactly once.
After SQLite apply, before reply	Unknown.	The same live-session sequence returns the stored bounded result and does not apply twice.
During snapshot install	Existing traffic follows policy.	Restart picks old complete or new complete snapshot, never the temporary one.

Run the matrix in one-node and three-voter modes. In cluster mode, vary whether the killed process is leader, quorum follower, or lagging follower. Add packet drop, duplicate, reorder, reconnect, partial frame, and slow-disk schedules.

13.3. Persistence assertions

The integration suite must establish all of the following for its tested schedules:

1. two nodes never apply different transaction payloads at one Paxos slot;
2. `applied_slot` is monotonic and never passes a missing committed slot;
3. an acknowledged request remains after any single-node loss and total restart;
4. a permitted session sequence affects SQLite at most once;

5. duplicate, gap, old, and expired session sequences never execute SQL outside the one permitted next-sequence transition;
6. a value-bearing envelope delivered before `payload_stored` is never stepped; it is retained only within the configured object/byte bounds (or dropped for retransmission), and Phase-1 promises obey the same rule;
7. an accepted payload is never garbage-collected because of age or ballot change alone, and becomes collectible only after durable reference removal;
8. a quorum read fence never completes for an old ballot after a higher-ballot write has completed, and every completed fence read includes all preceding acknowledged writes;
9. every sent Paxos message whose meaning depends on a write has a preceding successful sync event in the trace;
10. a node with missing/corrupt durable-prefix data refuses to vote;
11. a torn final journal record is recoverable, while interior corruption is reported rather than silently skipped;
12. removing the SQLite materialized image and replaying snapshot plus log yields the same logical database hash;
13. with only one of three voters available, no write is acknowledged.

These tests demonstrate the implementation against selected failures. The Paxos agreement argument and the existing simulator/model remain the basis for the general safety claim.

14. Verification and performance gates

14.1. Test layers

Layer	Focus	Command
Unit	Journal codec, checksums, payload format, SQLite frame capture/apply, session sequence/result state.	`zig build test`
Deterministic host simulation	Disk-prefix crashes, payload ordering, message faults, restart, snapshot epochs.	`zig build test`
One-process integration	Real files, SQLite, sync, SIGKILL, reopen.	`cd zaxonlite && zig build test-single`
Three-process integration	TCP, election, quorum loss, catch-up, failover, total restart, CLI routing and output.	`cd zaxonlite && zig build test-cluster`
CLI contract	Interactive/script SQL, JSON schema, exit codes, redirect, auth failure, timeout, and stopped-node offline safety.	`cd zaxonlite && zig build test-cli`
Fuzz/property	Record parser, peer decoder, random SQL transactions and frame equivalence.	`cd zaxonlite && zig build fuzz`
Soak	Hours of concurrent reads/writes, snapshots, restarts, disk delay.	`cd zaxonlite && zig build soak`

Layer	Focus	Command
Benchmark	Throughput, tails, fsync count, CPU, allocations, bytes, recovery time.	`cd zaxonlite && zig build benchmark`

14.2. Initial release gates

1. Zero lost acknowledged requests across 10,000 seeded crash schedules.
2. Zero duplicate applies across leader-before-reply failures.
3. Three-process mandatory scenario passes 100 consecutive runs in CI stress.
4. Concurrent writes submitted through all three endpoints are applied exactly once in one serial order; overload returns bounded backpressure.
5. Recovery from a 1 GiB database and retained suffix meets a published target chosen after the Phase 2 baseline.
6. Memory and queued payload bytes remain within configured bounds under a slow follower and overload.
7. Performance comparison publishes configuration and raw results; no regression over 10% at p95 or throughput is accepted without review.
8. SQLite `integrity_check` and a logical-content hash pass after every destructive recovery scenario.
9. Representative majority and flexible-quorum TLA+ configurations complete without violating agreement, and the proof-obligation table contains no unowned stable-release requirement.
10. Every three-process trace satisfies payload-data-before-dependent-accept on each ordered stream, payload-before-vote, durable-claim-before-send (except the explicitly classified local accept pipeline), chosen-before-apply, contiguous-apply, and acknowledge-after-session-update ordering.
11. One million sequential writes across bounded session churn leave storage proportional to active sessions plus the configured result window, not total historical requests.
12. The linearizable-read benchmark path performs no consensus-log append or disk sync per read; the committed barrier remains available as a differential correctness oracle.

15. Risks and decisions still required

Risk	Mitigation	Decision point
Public SQLite integration cannot safely intercept commit frames.	Time-box Phase 0; test supported SQLite versions; record an ADR and stop rather than ship a hidden patch.	Before API design.
Large values conflict with the bounded in-memory Paxos core.	Descriptor + content-addressed payload protocol, strict payload-before-vote rule, epoch compaction.	Phase 0 format ADR.
Two durable representations drift.	Journal/snapshot is authoritative; SQLite image is validated materialized state and rebuildable.	Phase 1.
Chosen payload exists only on failed voters.	Persist payload before every counted vote; recovery fetches by digest; refuse to vote when missing.	Phase 2.

Risk	Mitigation	Decision point
Dependent WAL slots are pipelined unsafely.	Batch transactions inside one chosen value; wait for a chosen predecessor before building a dependent slot; validate cumulative base/result chain hashes. Hash mismatch is fatal, not a way to discard a chosen value.	Phase 1 and Phase 5.
“Horizontal scaling” is interpreted as multi-leader writes to one file.	Document one writer per database; scale aggregate writes with independent database/Paxos shards and no implicit cross-shard transaction.	Before public preview.
Read barrier limits read throughput.	Keep it as bring-up/oracle fallback; require the nonce-bearing Paxos quorum fence and its history proof before stable release. Leases remain later work.	Phase 3 and release gate.
Payload transfer races or blocks Phase 1.	Per-peer storage-ACK gates, bounded offer/stream state, pre-step materialization with the current core, and staged recovery after correctness.	Phase 2.
Retry metadata grows without bound.	One outstanding sequence per never-reused replicated session, bounded last result, ordered expiry, and no implicit session recreation.	Phase 1 and Phase 3.
Group sync increases tail latency.	Bound both group size and timer; expose profiles; measure tails.	Phase 5.
Snapshot GC deletes recoverable state.	Quorum snapshot coverage, two generations, durable-reference payload GC, and crash matrix; age/lease alone never deletes an accepted payload.	Phase 4.
C ABI freezes immature semantics.	Keep Zig API experimental through Phase 3; version handles/errors; freeze only at RC.	Phase 5.
Bounded model checking is advertised as a complete proof.	Keep the unbounded invariant argument, bounded TLC evidence, Zig refinement mapping, and product host obligations explicitly separate.	Every release.

15.1. ADR backlog

1. SQLite capture/apply mechanism and supported SQLite versions.
2. Authoritative storage model and exact client-ack point.

3. Transaction descriptor, payload format, cumulative chain hashing, optional inline threshold, and compile-time/on-disk size limits.
4. Journal framing, checksum, sync primitive, and directory durability per OS.
5. Paxos quorum read-fence API/proof, barrier fallback, later lease boundary, and stale-read API.
6. Session allocation, sequence/result bounds, ordered expiry, and error semantics for old or expired retries.
7. Snapshot construction, quorum coverage, installation, and garbage collection.
8. Transport security, protocol negotiation, and certificate rotation.
9. Static bootstrap and the later stop-sign reconfiguration workflow.
10. Compatibility policy for on-disk, wire, Zig, and C APIs.
11. Writer queue, multi-transaction batch boundaries, maximum delay, and dependency-chain invariant.
12. Multi-database/shard identity, placement, routing, and cross-shard limits.
13. CLI RPC, configuration precedence, output schema, exit codes, and offline recovery safety.

16. Repository layout proposal

Zaxonlite lives in the monorepo as its own Zig package, `zaxonlite/`, with a path dependency on the parent `paxos` package and the pinned SQLite amalgamation as a `build.zig.zon` dependency. The layout as implemented:

```
src/                          # existing Paxos library remains independent
zaxonlite/
  build.zig                   # package build: library, zaxon CLI, tests
  build.zig.zon               # paxos (path) + sqlite amalgamation (pinned)
  src/root.zig                # public embedded API and exports
  src/embedded.zig            # transport-owning runtime-role facade
  src/capi.zig                # local and transport-owning C facades
  src/server.zig              # authenticated peer/client transport host
  src/client.zig              # redirecting RPC client and streamed backup
  src/gateway.zig             # stateless client stream router
  src/roles.zig               # node-role capability matrix
  src/node.zig                # host: lifecycle, write path, recovery,
                              #   sessions, snapshots, epoch rollover, GC
  src/main.zig                # zaxon CLI: shell, exec, query, status, ops
  src/sqlite.zig              # narrow SQLite bindings + wal hook
  src/wal.zig                 # WAL frame capture, payload codec, page apply
  src/command.zig             # fixed-size descriptor codec + chain hashes
  src/types.zig               # the shared ReplicatedLog instantiation
  src/journal.zig             # framed, checksummed protocol journal
  src/payload_store.zig       # content-addressed immutable payloads
  src/durability.zig          # parent-directory sync after link/rename
  src/integration_test.zig    # single-process durability/recovery suite
  src/cluster_test.zig        # three-process failover and catch-up suite
  src/role_cluster_test.zig   # voter, witness, standby, replica suite
  src/fault_cluster_test.zig  # adverse transport/storage schedules
  benchmarks/                 # reproducible rqlite and pinned dqlite comparisons
docs/zds/records/0002-zaxonlite-product-plan.typ
docs/zds/records/0004-zaxonlite-format.typ  # frozen on-disk/wire format contract
```

The state-machine and snapshot lifecycle remain together in `node.zig` because they share the image-rebuild invariants. Wire encoding, transport authentication, durability primitives, payload storage, prepared bindings, and configuration loading are separate narrow modules.

Keep the consensus package free of SQLite, filesystem, sockets, threads, and allocators. Product-specific host code consumes its effects. If a core change is needed, first express it as a general bounded Paxos capability with independent protocol tests.

17. Definition of done

The product is ready for a first stable release when an application can link one library, use the same API in one-node or runtime-role cluster mode, execute ordinary SQLite transactions, lose any one voter, and recover every acknowledged write after restart. It can also run the reference process, SQL shell, and operations commands from one documented CLI. The repository contains a reproducible real three-process test, crash-point tests that enforce durable-claim-before-send, format/version documentation, an operator recovery guide, and fair benchmark evidence. The proof-obligation matrix must identify evidence for consensus agreement, host durability, payload availability, SQLite state agreement, barrier and quorum-fence reads, bounded retry sessions, and snapshot epoch transitions; bounded model checking must remain labeled as bounded evidence.

The release notes must state remaining limits plainly: one writer, supported SQLite features and platforms, maximum transaction/batch settings, static or dynamic membership status, read modes, sharding status, and what an ambiguous timeout means. "Horizontal scale" must distinguish voter availability, follower read scale, and independent-shard write scale.

18. References

1. `paxos-zig` protocol contract: `src/root.zig`, `src/protocol.zig`, and `src/replicated_log.zig` in this repository.
2. dqlite architecture: official architecture notes.
3. dqlite replication and custom VFS/WAL approach: official replication notes.
4. dqlite consistency model and its documented stale-read behavior: official consistency notes.
5. rqlite feature overview: official features.
6. rqlite read-only/non-voting nodes and freshness controls: official read-only node guide.
7. rqlite cluster sizing and failed-node operations: official clustering guidelines.
8. rqlite automatic discovery and bootstrap: official automatic clustering guide.
9. rqlite linearizable, strong, and local read semantics: official read-consistency guide.
10. SQLite VFS interface: SQLite VFS reference.
11. SQLite WAL format and concurrency: SQLite WAL documentation.
12. SQLite atomic commit considerations: SQLite atomic commit documentation.
13. Leslie Lamport, "Paxos Made Simple": Phase-1 recovery, agreement induction, distinguished proposer, and Multi-Paxos state-machine construction.
14. Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman, "Flexible Paxos": only cross-phase quorum intersection is required.
15. Diego Ongaro and John Ousterhout, "In Search of an Understandable Consensus Algorithm": the Raft mechanisms explicitly excluded by this design.
16. Repository formal artifacts: `specs/Paxos.tla`, `specs/Paxos.cfg`, and `specs/README.md`.