

SQLodin

Use, design and evidence

A replicated SQLite database in Odin. Writes can enter at any voter. Every voter applies the same ordered history.

Slot 1 Owner 1	Slot 2 Owner 2	Slot 3 Owner 3	Slot 4 Owner 1	Slot 5 Owner 2	Slot 6 Owner 3
--------------------------	--------------------------	--------------------------	--------------------------	--------------------------	--------------------------

Three owners. One application order.

Vikrant Rathore

With assistance from Ronak Rathore

25 September 2026

How to use this book

Read Chapter 1 to build SQLodin and run your first query. Read Chapter 2 and Chapter 3 for application code. The design begins at Chapter 4. The proof chapter, Chapter 9, explains which claims follow from induction, which are checked within finite models, and which depend on the implementation and its environment.

If you need to...	Start here
Run SQL	Chapter 1; command lookup in Chapter 12
Write a Python application	Chapter 2; search in Chapter 3
Understand multi-master writes	Chapter 4 and Chapter 5
Reason about crashes or retries	Chapter 6, Chapter 7 and Chapter 8
Inspect the mathematics	Chapter 9 and its links to executable specifications
Choose a workload or size a deployment	Chapter 10 and Chapter 11

The book describes the qualified fixed three-voter implementation. It uses the complete pinned paxos-odin library, SQLite 3.51.3, sqlite-vec 0.1.9 and OpenSSL 3.5.8. The native service and Python package have separate version numbers. Store format 5 and SQL policy 9 identify the current storage and SQL contracts.

A release decision has a scope. The Linux checks use three instances and durable local storage. They do not establish independent physical failure domains or physical power-loss behavior. Performance targets remain unmet. Large-database recovery time is not guaranteed. Chapter 11 gives the measurements and their limits.

The [release record](#) identifies the tested binary and accepted scope. The [documentation index](#) leads to operating guides, design decisions and historical records. Historical notes describe earlier states of the code; they do not override the contracts explained here.

Contents

1	Build, connect and run SQL	4
1.1	Build the native binary	4
1.2	Form a fixed cluster	4
1.3	Connect a client	5
2	SQL, Python and transactions	6
2.1	A reusable Python connection	6
2.2	Two kinds of transaction	6
2.3	Handle an unknown outcome	7
3	Full-text and vector search	8
3.1	What the scores mean	8
3.2	Combine ranks, not incompatible scores	9
3.3	Representations and bounds	9
4	One ordered history	10
4.1	What multi-master distributes	11
4.2	Separate the layers	11
5	Quorum agreement and progress	12
5.1	The intersection fact	12
5.2	Promises, votes and ballots	12
5.3	The agreement argument	12
5.4	Progress is a different claim	13
6	Durable execution	14
6.1	Two stores, one acknowledgement rule	14
6.2	Group commits without changing meaning	15
6.3	Why ordered SQL still needs a policy	15
7	Reads, retries and serial order	16
7.1	A quorum frontier bounds every completed write	16
7.2	An optimistic transaction validates its predecessor	17
7.3	A retry is a request identity	17
7.4	Reclaim sessions without reviving old writes	17
8	Snapshots, restart and replacement	18
8.1	Certify a logical prefix	18
8.2	Publish an old-or-new generation	18
8.3	Rejoin with state intact	19
8.4	Back up an application recovery point	19
8.5	Replace lost state in a new namespace	19
8.6	Upgrade and renew credentials deliberately	19
9	Formal verification	20
9.1	Begin with a state and an invariant	20
9.2	Durable-history induction	20
9.3	Prefix recovery with unbounded slots	21
9.4	Finite models explore the dangerous interleavings	21
9.5	Connect the model to running code	22
9.6	Reproduce the evidence	23
10	Work, memory and latency	24
10.1	Own the data that outlives a turn	24
10.2	Admission protects the quorum path	24
10.3	A simple cost model	25
10.4	What the resource checks show	25
10.5	Embed the durable host	25
11	Performance evaluation	26
11.1	The durable workload matrix	26
11.2	Targets remain visible	27
11.3	After SOD 0005: fewer sequential barriers	27
11.4	Earlier cross-system evaluation	27
11.5	Capacity, recovery and topology	28
12	Reference and source map	30
12.1	Native commands	30
12.2	Interactive client	31
12.3	Default service bounds	31
12.4	Terms used in the book	32
12.5	Find the implementation or argument	32

1 Build, connect and run SQL

SQLodin offers three entry points. The local shell opens an ordinary SQLite file. The cluster client talks to a running SQLodin voter. The embedded Odin library lets an application own the host loop. These paths have different responsibilities.

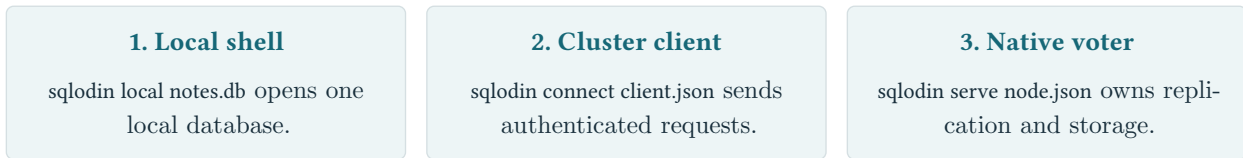


Figure 1: Choose the interface by who owns the database and replication.

Keep the two shells separate

sqlodin local bypasses replication. Use it for standalone files, never to modify an active voter's database. Use sqlodin connect for replicated SQL.

1.1 Build the native binary

Install Odin, Python 3, a C compiler and platform headers, ar, Make and Perl. The verified builds use macOS arm64 and Linux x86_64. SQLite, sqlite-vec and OpenSSL are C libraries, so the build also needs a C compiler.

```
git clone --recurse-submodules https://github.com/insanai/sqlodin.git
cd sqlodin
./build.sh
bin/sqlodin version
bin/sqlodin local notes.db 'SELECT 2 + 3;'
```

The build verifies pinned source hashes and links the database and TLS libraries statically. It also embeds the shell from the same SQLite release. The resulting binary needs normal OS runtime libraries; it is not a fully static libc binary. No installed SQLite or OpenSSL executable is needed to run the service.

build/downloads/ holds verified downloads. build/native/ holds compiled archives. Use python3 tools/build_cli.py --offline when those caches are already present. Keep the build manifest and license notices with the binary. A library security update requires a new pinned build; updating a system shared library cannot change code already linked into SQLodin.

1.2 Form a fixed cluster

Declare the same sorted members on every voter, with distinct IDs, directories and leaf credentials. This node-1 plan uses example addresses and absolute paths:

```
{
  "cluster": "orders", "node": 1,
  "listen": "10.175.52.19:7600", "data": "/srv/sqlodin/orders/node1",
  "certificate": "/srv/sqlodin/tls/node1.pem",
  "key": "/srv/sqlodin/tls/node1.key", "ca": "/srv/sqlodin/tls/ca.pem",
  "members": [
    {"id":1,"address":"10.175.52.19:7600","identity":"node1.sqlodin.test"},
    {"id":2,"address":"10.175.52.20:7600","identity":"node2.sqlodin.test"},
    {"id":3,"address":"10.175.52.21:7600","identity":"node3.sqlodin.test"}
  ],
  "clients": ["app.orders.sqlodin.test"]
}
```

Provision CA-signed certificates first. Each voter certificate needs the exact DNS SAN in its member entry and both serverAuth and clientAuth usage. A SQL client needs clientAuth and a SAN named in clients. Keep the CA private key off the voters. Use restrictive permissions for private keys and data directories. Fixture certificate generators in the test tools are not an enrollment service.

Create node-2 and node-3 plans by changing the node ID, listen address, data path and leaf key/certificate. Keep the cluster name, trust and member list consistent. Addresses currently use numeric IPv4. Paths should be absolute.

```
mkdir -p -m 700 /srv/sqlodin/orders/node1
bin/sqlodin serve node1.json --create # first start only
# After stopping the process:
bin/sqlodin serve node1.json # reopen existing state
```

--create refuses existing database state. Ordinary startup refuses missing state. Do not recreate an empty database under an old voter identity: the old promises and votes are part of the safety argument. Chapter 8 explains replacement.

1.3 Connect a client

A CLI client selects one endpoint. The expected DNS identity is checked against the server certificate; it does not replace CA verification.

```
{
  "cluster": "orders",
  "address": "10.175.52.19:7600",
  "identity": "node1.sqlodin.test",
  "certificate": "/srv/sqlodin/tls/app.pem",
  "key": "/srv/sqlodin/tls/app.key",
  "ca": "/srv/sqlodin/tls/ca.pem"
}
```

```
bin/sqlodin connect client.json
```

```
CREATE TABLE account(id INTEGER PRIMARY KEY, balance INTEGER NOT NULL);
INSERT INTO account VALUES (1, 100), (2, 0);
SELECT id, balance FROM account ORDER BY id;
```

SQL ends with a semicolon. Use .help for commands and .mode json for machine output. A script is not automatically atomic; Chapter 2 explains BEGIN and COMMIT.

A fresh query through each endpoint checks more than a listening socket. A lone minority voter can answer local status but cannot complete a new quorum-backed read.

Build details are in [the build guide](#). Certificate and protocol fields are in [the service guide](#).

2 SQL, Python and transactions

A successful write means that its expected request is durably chosen and applied at the responding voter. A query uses a fresh quorum barrier by default. Neither a proposal slot nor a local status response is a substitute for these conditions.

2.1 A reusable Python connection

Install from the repository with `uv add ./languages/python`. The synchronous base client requires Python 3.11 or later and has no runtime dependencies. Reuse a connection; opening one per write consumes new durable session capacity.

```
import sqlodin

nodes = [
    sqlodin.Endpoint("10.175.52.19:7600", "node1.sqlodin.test"),
    sqlodin.Endpoint("10.175.52.20:7600", "node2.sqlodin.test"),
    sqlodin.Endpoint("10.175.52.21:7600", "node3.sqlodin.test"),
]
tls = sqlodin.TLS(ca="ca.pem", cert="app.pem", key="app.key")
with sqlodin.connect(nodes, cluster="orders", tls=tls) as db:
    db.execute("UPDATE account SET balance=balance+? WHERE id=?", (20, 1))
    row = db.query("SELECT balance FROM account WHERE id=?", (1,)).one()
    print(row["balance"])
```

`execute()` returns a write result, not a row stream. `query()` returns immutable rows with column names, numeric positions, `one()`, `first()` and `scalar()`. Use aliases when a query repeats a column name. BLOB results become bytes. Write RETURNING is rejected by the current replicated SQL policy.

A connection serializes its calls. Use separate reusable connections for concurrent work. Do not share one across forked processes. The client has no native async API.

2.2 Two kinds of transaction

Interface	What it does
<code>db.transaction()</code>	Buffers writes, then sends one atomic body. It has no query method.
SQLAlchemy or CLI BEGIN	Reads a fresh revision, previews staged work, then validates at commit.

The next examples assume an open connection named `db`. A buffered write batch is useful when the client already knows every change:

```
with db.transaction() as tx:
    tx.execute("UPDATE account SET balance=balance-? WHERE id=?", (20, 1))
    tx.execute("UPDATE account SET balance=balance+? WHERE id=?", (20, 2))
```

A Python exception discards the unsent batch. A classified SQL error rolls back the whole request. Do not put BEGIN, COMMIT or ROLLBACK inside this body; the service owns its transaction boundaries. Plain `?` parameters remain bound values.

Install the optional dialect with `uv pip install './languages/python[sqlalchemy]'`. Declare tables or ORM models explicitly; general schema reflection is not supported. The following uses SQLAlchemy Core to show the complete transaction boundary:

```

from sqlalchemy import text
from sqlodin.sqlalchemy import create_engine

engine = create_engine(nodes, cluster="orders", tls=tls)
with engine.begin() as conn:
    balance = conn.execute(
        text("SELECT balance FROM account WHERE id=:id"), {"id": 1}
    ).scalar_one()
    if balance >= 20:
        conn.execute(text("UPDATE account SET balance=balance-20 WHERE id=1"))
        conn.execute(text("UPDATE account SET balance=balance+20 WHERE id=2"))

```

ORM Session transactions use the same protocol. Flush can obtain generated integer keys. Rollback discards staged work; nested savepoints edit that staged body. Releasing a savepoint does not durably commit anything. Two-phase/XA transactions and general DML RETURNING are outside the interface.

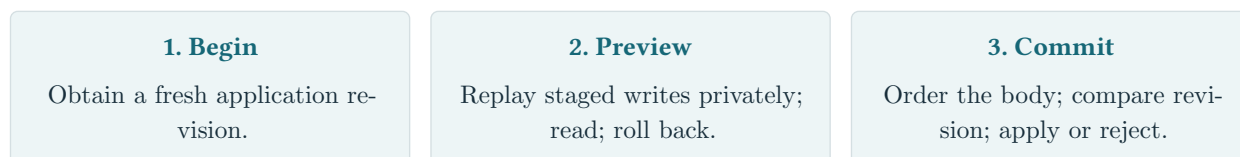


Figure 2: No SQLite writer lock is held across a client network round trip.

If another application write intervenes, the transaction conflicts, even when the rows are disjoint. SQLSTATE 40001 means retry the entire transaction, including its reads and decisions. SQLodin does not rerun application code automatically. The reason this conservative rule handles predicate reads is developed in Chapter 7.

2.3 Handle an unknown outcome

A lost response does not tell you whether a write committed. The Python connection retains the exact pending request and blocks a different write until it is resolved.

```

try:
    db.execute("UPDATE account SET balance=balance+? WHERE id=?", (20, 1))
except sqlodin.UnknownOutcome as exc:
    saved = exc.pending.to_json() # persist securely for process recovery
    # After connectivity is restored, on the same connection:
    result = db.resolve_pending()

```

To recover on another connection, decode the saved identity with `sqlodin.PendingWrite.from_json(saved)` and pass it as `pending=` to `connect()`. Then resolve it. Do not submit the SQL under a new identity. Saved pending requests contain SQL and application values; protect them like application data.

Python cannot recover a request identity lost in a process crash before the application saved it. Use application-level operation IDs and reconciliation where that window matters. The CLI instead saves its pending request in a private durable state file before sending it. Keep that file and use `.pending` and `.retry` after reconnecting.

Conflict and uncertainty require different actions

A definite conflict permits a new transaction after recomputing its reads. An unknown commit requires resolving the old identity first. Treating uncertainty as an abort can apply a payment twice.

The [Python reference](#) gives result types, exceptions and search methods. Chapter 12 collects the service limits.

3 Full-text and vector search

Search uses ordinary replicated content rows, FTS5 text indexing and sqlite-vec distance functions. The Python helper updates content, embeddings and FTS rows in one transaction. A search query uses one database snapshot after a fresh read barrier.

```
with sqlodin.connect(nodes, cluster="orders", tls=tls) as db:
    docs = db.create_search_index("documents", dimensions=3)
    docs.put(1, title="Consensus", body="Durable Paxos replication",
            vector=[0.9, 0.1, 0.0])
    words = docs.full_text("Paxos", limit=5)
    nearby = docs.nearest([1, 0, 0], metric="l2", limit=5)
    combined = docs.hybrid("durable", [1, 0, 0], limit=5, candidates=30)
```

This example uses `sqlodin`, `nodes` and `tls` from Chapter 2. Create an index once. On later connections, use `db.search_index("documents", dimensions=3)`. Opening a handle does not validate an existing schema. Direct SQL can break the relationship between content and FTS rows, so use the helper for index mutations.

3.1 What the scores mean

FTS5 ranks matching text with BM25; lower scores rank first in this API. Vector search ranks an exact distance scan. For vectors x and y of dimension d , Euclidean distance is

$$D_2(x, y) = \sqrt{\sum_{i=1}^d (x_i - y_i)^2}.$$

Cosine distance is one minus normalized similarity:

$$D_{c(x,y)} = 1 - \frac{\sum_{i=1}^d x_i y_i}{\sqrt{\sum_{i=1}^d x_i^2} \sqrt{\sum_{i=1}^d y_i^2}}.$$

Cosine requires nonzero vectors. These formulas compare geometry, not meaning by themselves. The embedding model determines what that geometry represents.

An exact scan over n candidate rows evaluates roughly nd components. Returning five results does not restrict the scan to five rows. SQLodin does not supply an approximate nearest-neighbor index through this API; writable vec0 tables are outside the replicated SQL policy.

3.2 Combine ranks, not incompatible scores

BM25 and vector distance have different scales. Hybrid retrieval instead combines positions in two ranked candidate lists. With rank constant k and ranks starting at one, document u receives

$$\text{RRF}(u) = \sum_{L:u \in L} \frac{1}{k + \text{rank}_{L(u)}}.$$

A document missing from a list receives no contribution from that list. Higher fused scores rank first; document IDs break ties. The helper computes both lists within one query, so they observe the same database snapshot.

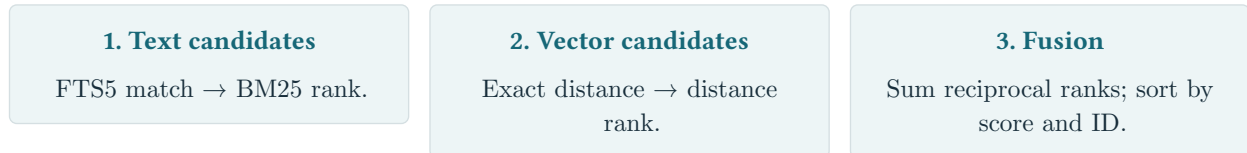


Figure 3: The two candidate lists feed one fused result. Neither score scale is reused directly. For example, with $k = 60$, a document ranked first and third scores $\frac{1}{61} + \frac{1}{63} \approx 0.03227$. A document appearing only at rank one scores $\frac{1}{61} \approx 0.01639$. Candidate truncation can therefore change the fused ranking.

3.3 Representations and bounds

`sqlodin.Vector` stores immutable finite float32 values. Components are rounded before encoding, so retries preserve the submitted values. Each vector has 1–384 components; the request has a shared 384-component budget. Embeddings occupy ordinary BLOB columns. Use `Vector.from_bytes()` to decode a returned embedding.

Titles, bodies and FTS expressions each retain the 256-byte UTF-8 parameter limit. This is a bounded chunk interface, not unrestricted document ingestion. Search results also obey the read instruction and result budgets. A too-expensive query fails without returning a partial success. Custom tokenizers and general FTS maintenance commands are outside the tested interface.

4 One ordered history

Suppose two clients update the same account through different voters. Each client needs a definite outcome. Every voter must eventually reach the same account state. SQLodin solves the ordering problem first, then applies that order to SQLite.

A log slot is a place in this order. Paxos chooses at most one value for each slot. A value may contain a SQL transaction, a read marker, a skip or a host control. SQLite applies chosen values as a contiguous prefix. Choosing slot five does not permit applying it while slot three is unresolved.

Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6
Chosen	Chosen	Missing	Chosen	Chosen	Open
Applied	Applied	Wait	Wait	Wait	Wait

Figure 4: The applied frontier stops at two. Slots four and five are chosen, but their effects must wait for slot three.

Let S_0 be the initial logical database state and v_s the value chosen for slot s . A deterministic application transition F defines

$$S_s = F(S_{s-1}, v_s).$$

If two voters start at the same state and apply the same prefix, they reach the same state by induction. The induction is short. Its premises are substantial: agreement must survive crashes, execution must be deterministic, and every prefix update must be atomic. The following chapters examine those premises separately.

4.1 What multi-master distributes

Every configured voter may admit writes. Ownership rotates by the position in the sorted membership list:

$$\text{owner-index}(s) = (s - 1) \bmod N.$$

Member IDs need not be consecutive. The formula selects an index, not an arbitrary numeric node ID. For three voters, the first six owners look like this:



Figure 5: Ownership partitions proposal slots. It does not partition application data.

A healthy owner can propose at its special round-zero ballot without a preparatory phase-one exchange. Another voter needs a higher ballot to recover that slot. Idle owners fill needed gaps with skips; a skip advances order without changing user rows.

Every voter still applies every chosen SQL transaction. Three voters therefore do not give three independent SQLite write engines for disjoint shards. Replication adds durability and fault tolerance, with coordination and storage costs.

4.2 Separate the layers

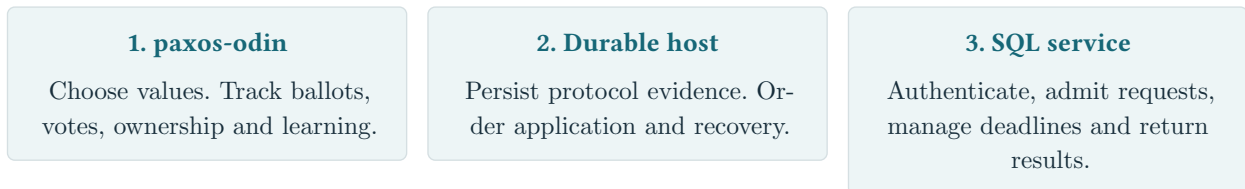


Figure 6: SQLLodin imports the complete pinned Paxos library. It does not copy fragments into a second protocol.

The pin is `c3d197016c1f938db23fdf7f1fe87fbd8bb86ac1c`. The adapter in `src/paxos.odin` supplies SQLLodin values and enables rotating ownership. The library owns consensus. `src/durable/` owns its storage contract. `service/` supplies the native network loop; `cli/` and the Python package are clients of that service.

The low-level embedded API exposes these responsibilities to its caller. Returning from `propose` means a proposal was admitted. The caller must drive messages and check the durable outcome. The native service performs that work before returning successful completion to an application.

5 Quorum agreement and progress

A voter can crash after sending a reply. A message can arrive twice or arrive late. Paxos must preserve a chosen value through these events. The key is not that every voter always agrees; it is that a later successful proposer cannot choose a different value for the same slot.

5.1 The intersection fact

For N voters, a majority has size $q = \lfloor \frac{N}{2} \rfloor + 1$. Any two majority sets Q_1 and Q_2 satisfy

$$|Q_1 \cap Q_2| \geq 2q - N > 0.$$

The inequality follows because their union contains at most N voters. For three voters, every quorum has two members. Any pair of such quorums shares at least one.



Figure 7: The shared voter carries evidence from an earlier choice into a later recovery.

Intersection alone is insufficient. The shared voter must retain its promise and accepted vote across a crash. It must also report that evidence during recovery.

5.2 Promises, votes and ballots

A ballot orders competing attempts. A proposer uses a unique higher ballot to recover a slot. An acceptor promises not to accept lower ballots and reports its accepted value and ballot, if any. Both promise and vote are durable facts.

Phase	Rule
Prepare	Obtain a majority of promises for the new ballot.
Select	If replies contain votes, use the value at the highest accepted ballot. Otherwise choose a value.
Accept	Obtain a majority of durable votes for that ballot and value.
Learn	Disseminate the chosen value; apply it only in prefix order.

A ballot must not propose two different values for one slot. An acceptor rejects conflicting values at one ballot and values below its applicable promise. The unique slot owner may use round zero; acceptors check this ownership rule. Round zero is not permission for another voter to skip recovery.

5.3 The agreement argument

Assume value v was chosen by quorum Q at ballot b . Consider a later successful prepare quorum Q' . It intersects Q . At least one reply therefore contains a vote for v or a later accepted vote.

Why must that later vote also name v ? Induct over higher ballots. The first higher proposer that could obtain a quorum must carry forward the earlier accepted value. At each subsequent successful ballot, highest-vote selection carries the same value forward. A lower ballot cannot collect a new conflicting quorum after the higher promises: the quorums intersect again. Thus two chosen values for one slot must be equal.

This is an argument about protocol actions under durable, truthful acceptors. It does not prove that arbitrary code implements those actions. The model and implementation checks in Chapter 9 test that correspondence. Byzantine voters, lost durable identities and a device that reports a false successful synchronization violate the premises.

5.4 Progress is a different claim

Safety says conflicting choices do not occur. Progress says pending work eventually finishes. A network that drops every message can preserve safety forever while doing no useful work. SQLodin needs a surviving majority, eventual successful communication, fair scheduling and storage/application work that eventually completes.

For a finite offered frontier h , consider the lowest unapplied slot $s \leq h$:

1. A live owner proposes work or fills a required gap with a skip.
2. If the owner is absent, a surviving voter recovers the slot at a higher ballot.
3. Under an eventually non-preempted exchange, a majority chooses a value.
4. Learning and application advance the contiguous prefix.

The distance $h - a$ decreases as applied prefix a advances. Repeating the argument reaches h . New arrivals create new frontiers; the argument does not promise an individual request freedom from starvation under perpetual interference or overload.

Healthy gap filling is event-driven. A regression advances the prefix with no timer ticks, guarding against the earlier 100 ms slot-selection delay. Missing-owner recovery still uses a failure detector: the default stall interval is ten 100 ms ticks. A timer is evidence that recovery should be attempted, not proof that a voter has failed.

One voter down

With three voters, either survivor can admit a write when the two survivors communicate and make durable progress. One isolated voter cannot acknowledge a new write or a fresh quorum read. Multi-master does not mean each voter can commit alone.

See [the composition argument](#) for the mapping to upstream ownership, revocation, promise and acceptor functions.

6 Durable execution

A network message can outlive the process that sent it. If a vote is sent before it is durable, another voter may choose a value using evidence that disappears after a crash. The recovered acceptor could then vote incompatibly. Persistence must therefore precede visibility, not merely follow it soon afterward.

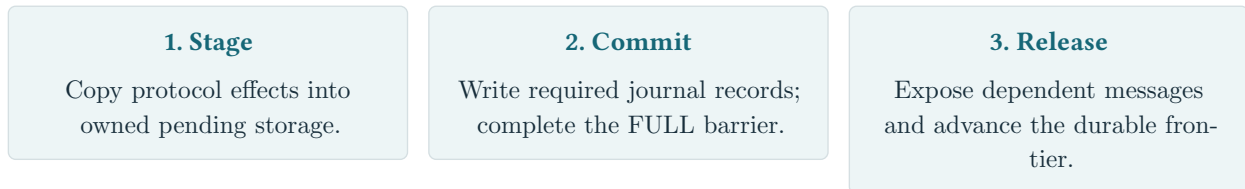


Figure 8: A successful synchronization separates private work from externally usable evidence.

6.1 Two stores, one acknowledgement rule

Format 5 separates application data from consensus state. The application database contains user rows, outcomes, retry fences, revision and the applied watermark. The consensus database contains node-local promises, accepted/chosen evidence, reservations and generation metadata. A donor's application image can be transferred; its acceptor identity cannot be copied onto another voter.

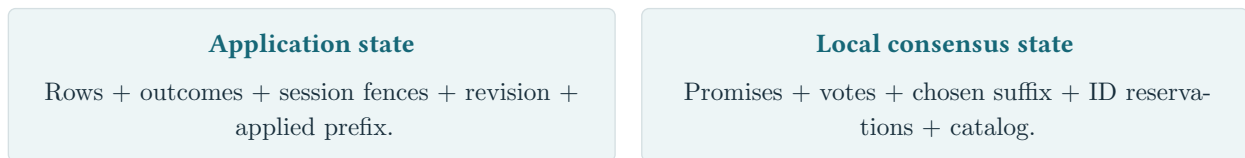


Figure 9: State transfer must preserve the recipient's acceptor facts.

Consensus evidence is durable before dependent effects are released. Chosen application work then commits its data and metadata atomically in SQLite. Only afterward can the service acknowledge the matching request. These separate transactions need no distributed transaction between the two files: ordering and replay bridge the possible crash gap.

Crash point	Recovery consequence
Before durable choice	No successful write response is justified. Retry the same identity.
After choice, before application	Replay the chosen suffix in order.
After application, before reply	The result exists; an identical retry recovers it.
After reply	Recovery must preserve the acknowledged outcome and data.

SQLite uses WAL and `synchronous=FULL` for the consensus journal, catalogs, images and the embedded engine. The separated service store treats its application database as a cache of the journal (SOD 0005): it commits with WAL `synchronous=NORMAL`, which keeps every crash state a committed prefix, and recovery replays the retained chosen suffix. An entry is applied only after the journal barrier containing its decision. Required standalone files and directories also cross explicit synchronization barriers. These rules assume the filesystem and device honor successful synchronization. SIGKILL and injected syscall failures test software boundaries; they are not physical power-cut certification.

6.2 Group commits without changing meaning

A FULL commit has a cost even when the transaction is small. SQLodin can group up to sixteen adjacent application requests in one outer transaction. A savepoint separates each request. Its SQL either succeeds or rolls back before its outcome is recorded.

Let R_i be the reference state after i individually committed requests. Let G_i be the private grouped state after staging the same prefix. The useful invariant is

$$G_i = R_i.$$

It holds initially. At the next request, both paths inspect the same retry fence and revision, execute the same deterministic SQL, and record the same outcome. A classified error rolls back that request in both paths. This establishes the induction step.

Deferred foreign keys need a check at every request boundary. Otherwise a later request could repair an earlier violation that the reference path would have rejected. SQL ROLLBACK conflict actions can abort the outer transaction; the implementation discards that unacknowledged attempt and retries through individual reference transactions. Unknown storage or execution failures are not semantic fallback: they fail closed.

Savepoint release does not acknowledge a write. The outer commit is the publication point for the group; its durability comes from the FULL journal barrier that preceded it. Journal grouping combines persistence for every Paxos transition of a service turn while preserving the same durable-before-send rule. Owned copies keep effect payloads valid when the next protocol transition reuses its working storage.

6.3 Why ordered SQL still needs a policy

A shared order does not make `random()` deterministic. Every voter might agree to execute the same expression and store a different value. SQLodin therefore restricts replicated functions, schema operations and extension behavior.

Wall-clock and random writer functions are rejected, including uses hidden in defaults. The service owns transaction boundaries and internal metadata. Peers require matching policy and engine fingerprints. Startup checks the pinned vector extension identity. The qualified cluster uses matching Linux x86_64 builds; local macOS checks do not qualify heterogeneous replication. Local read functions have a different role: their results affect future replication only when a client submits concrete values in a write.

Policy 9 also keeps at least one hidden-rowid alias available in an ordinary rowid table. A schema that shadows rowid, _rowid_ and oid together is rejected; certified logical snapshots need access to row identity. WITHOUTROWID tables do not need that alias. Schema rejection rolls back the entire request.

The deterministic-execution argument assumes identical logical state, admitted schema, parameters, collations and pinned engine behavior. It includes planner inputs and generated keys, not just a function allowlist. Physical-layout/cache variation tests supplement that argument. Unknown I/O, allocation or execution failures stop acknowledgement; a local timeout cannot invent a different replicated rejection or skip a chosen transaction.

The full [SQL policy](#) and [grouping argument](#) name the code and regressions.

7 Reads, retries and serial order

A read can return an old but internally consistent SQLite snapshot. That is not enough for an application that expects to observe writes completed before the read began. SQLodin's default read path establishes a fresh ordering point before opening the snapshot.

7.1 A quorum frontier bounds every completed write

After accepting a read, the service closes a cohort of already waiting reads. It then observes its own highest seen slot and asks peers for theirs. Once a read quorum has answered, including this voter, the largest reported slot H is the read's frontier. The cohort is answered when the local contiguous applied prefix reaches H . No application transition slips between that check and the snapshot acquisition.

A write acknowledged before the read's invocation was chosen, so a write quorum durably voted for its slot s . That quorum intersects the queried read quorum. Every voter's highest seen slot is at least every slot for which it holds a durable vote, and it never decreases, even across restart. Hence $H \geq s$, and the snapshot includes the write. The same argument orders reads: a read that returned a prefix A only did so after every slot up to A was chosen, so a later read obtains $H \geq A$. The actual snapshot may also include later concurrent writes. That places the read later within its invocation/response interval, which linearizability permits. The barrier writes nothing and needs no sync. This is Paxos Quorum Reads (Charapko, Ailijiang and Demirbas, 2019), whose "rinse" phase is the wait for the applied prefix.

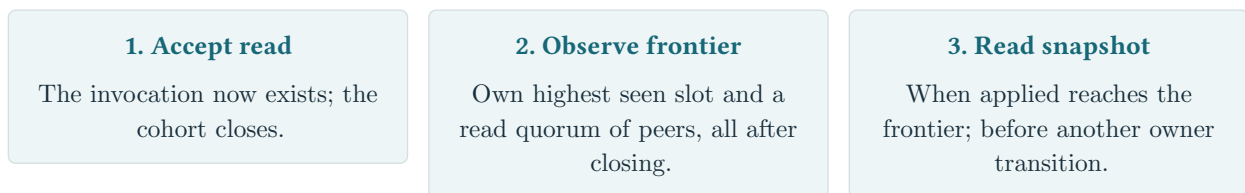


Figure 10: The frontier establishes order. The SQLite snapshot supplies the rows.

Already accepted reads share one frontier as a closed cohort. Every member must arrive before the frontier is observed. A later read cannot join, even while the cohort waits. Suppose a write finishes after the observation but before that later read arrives: reusing the frontier could return a snapshot older than the completed write.

Cancelling one cohort member leaves the other members' wait intact. Cancelling the last member retires the cohort. A reply for another cohort is ignored, and each peer counts once. Unanswered requests are repeated every 200 ms; a minority cannot complete a frontier. The embedded durable host keeps its single-use marker barrier (`begin_read/poll_read`) for callers without a peer transport.

`consistency="local"` explicitly omits this barrier and permits stale state. `status()` also reports only local state. Neither can establish that a quorum is available.

7.2 An optimistic transaction validates its predecessor

Let r denote the application revision. Begin obtains r after a fresh read barrier. Each preview crosses another fresh barrier, checks that the current revision is still r , replays the staged body privately, reads its effects, and rolls back.

At the ordered commit slot, a writing transaction checks

$$r_{\text{current}} = r_{\text{begin}}.$$

If the equality fails, it returns a conflict before executing the body. A successful application write advances the revision atomically with data and outcome. Read barriers, duplicate retries and rejected requests do not. Session retirement does advance it.

Thus every successful preview sees one committed predecessor plus the transaction's own staged writes. A successful commit can be placed at its applied log slot. A read-only transaction can be placed at one of its successful snapshots; a changed-revision read would instead fail. This yields a serial order respecting completed-before-invoked edges, under the consensus and deterministic-execution premises. The validation is database-wide. It covers predicate reads and phantoms without tracking read sets, but unrelated writes can conflict. That is a concurrency cost, not a hidden row-level lock. No preview workspace or SQLite writer lock survives between network requests.

7.3 A retry is a request identity

The service binds a request to the authenticated client principal, epoch, session, sequence and content. Reusing the same identity and body returns the recorded outcome without applying the body again. Reusing an identity with different content is an error. Changing voter endpoints does not create a new request.

The application transaction commits its data, outcome, session fence and applied prefix together. A lost reply leaves uncertainty at the client, but no gap between the durable application effect and the durable fence. Retrying the original request resolves that gap in knowledge. It does not require guessing whether the first connection reached the server.

7.4 Reclaim sessions without reviving old writes

There are at most 65,536 session rows in the current epoch. Local eviction would be unsafe: an old request could arrive after its fence vanished and execute again. Instead, a chosen retirement command advances epoch e to $e + 1$ and deletes old session rows atomically.

Every request is checked against the current epoch before session lookup. A request from an older epoch is expired even when its row is gone. Inductively, because the epoch never decreases or wraps, erased requests remain fenced at every later prefix. Only one scalar must survive, rather than one tombstone per old request.

Retirement is an operator action. Quiesce clients and resolve pending outcomes first. Use `.retire-sessions E --quiesced` OR `retire_sessions(expected_epoch=E)` deliberately. An expired uncertain request must not be relabelled into a new epoch. Retain its identity and reconcile the business operation. The CLI preserves its used state file's epoch; it does not silently adopt a newer one.

The [ordering argument](#) maps reads and previews to code. The [retirement argument](#) explains the epoch invariant and its crash tests.

8 Snapshots, restart and replacement

A bounded in-memory consensus window does not bound disk history. A long-running voter needs a way to replace an old application prefix with a verified image, preserve the remaining acceptor facts, and delete only history that recovery no longer needs.

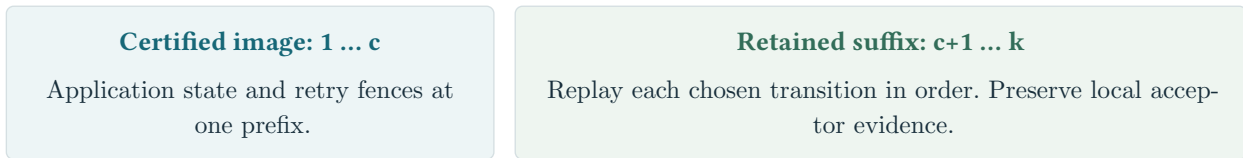


Figure 11: An image replaces an applied prefix, not the local acceptor's identity.

8.1 Certify a logical prefix

A snapshot worker pins an application read transaction at prefix c . It copies the image, checks its contents, and computes a logical digest. The image includes user state, schema, application revision and retry fences. Two valid SQLite files may have different page layouts while representing the same logical state; logical and physical digests serve different purposes.

A snapshot key binds configuration, engine, logical digest, generation and prefix. A receipt binds that key, the reporting voter, physical file hash and size. The voter must durably retain its verified image before issuing a receipt. The authenticated transport binds the receipt to the configured sender.

A certificate contains a majority of distinct, matching receipts. Duplicate voters do not increase its weight. The host then chooses a seal binding the complete certificate through Paxos. A locally copied file or a receipt alone is not permission to trim.

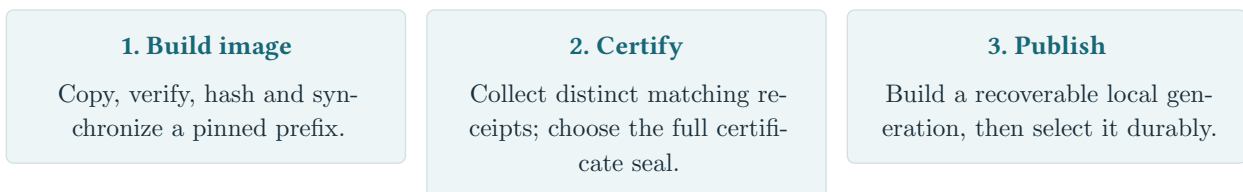


Figure 12: Each arrow carries a stronger condition. A partial image cannot skip ahead to publication.

8.2 Publish an old-or-new generation

A private replacement combines the certified application image with the recipient's local consensus suffix. It preserves promises, accepted values, chosen work, durable ID reservations and request fences. A background worker builds the base; the service then copies bounded deltas while the live voter continues to run.

Each owner turn copies at most 128 journal records or 1 MiB, then applies at most one chosen SQL transaction to the private replacement. One small SQL record can generate many data pages, so a record-count limit alone is not an application-work bound. The final publication checks that the replacement matches the required live frontiers and identity. A FULL root-catalog transaction selects the new generation.

Crashing before publication leaves the old generation selected. Crashing after it selects the complete new generation. The exact predecessor is retained. Retirement uses a durable ownership inventory, refuses active/predecessor or unowned paths, deletes eligible files, synchronizes the directory, then forgets their inventory entry. An interrupted deletion can retry missing files; it must never infer ownership from a filename alone.

Image retirement separately protects images needed by the active generation, predecessor and latest certificate. These rules matter because a correct new snapshot does not imply that every old file is immediately dispensable.

8.3 Rejoin with state intact

A temporarily offline voter first tries retained history. Beyond the retained prefix, it can fetch a certified image in authenticated bounded chunks and install a generation. The healthy majority can continue serving. Transfer cannot import the donor's promises or accepted state as if they belonged to the recipient.

Automatic maintenance uses one job, with a 256 MiB tail or fifteen-minute dirty-history trigger. Retained consensus history has an 8 GiB cap and admission reserves. Application files, predecessor images and staging space are separate disk costs. `maintenance: "manual"` disables automatic initiation/publication for deliberate offline workflows.

Maintenance bounds do not preempt blocked kernel I/O. Recovery retains its integrity checks; Chapter 11 reports the observed startup times.

8.4 Back up an application recovery point

Stop the source voter; the other two may continue serving. Use a new destination:

```
sqlodin backup source-node.json /backups/orders-cut
sqlodin verify-backup /backups/orders-cut
```

The backup contains a verified application image and manifest. It is not a quorum certificate and does not include later acknowledgements. To take a lossless recovery cut, first quiesce clients, resolve in-flight outcomes, obtain a fresh quorum read on the source, stop it and take the backup. Keep the old deployment fenced afterward.

8.5 Replace lost state in a new namespace

A lost disk cannot be repaired by creating a blank acceptor under its old identity. The supported procedure restores the fixed group from one verified backup into a globally unused cluster namespace and fresh directories. Stop every old process and its restart automation. Restore the same backup on every new voter before starting them:

```
sqlodin restore /backups/orders-cut new-node1.json --new-cluster
sqlodin serve new-node1.json
```

Repeat the restore for the other new voter plans. Check fresh quorum reads through all endpoints. The backup prefix and retry fences survive; donor promises, votes and ID reservations do not become the new acceptor's state. A genesis digest binds peers to the same restored initial image. Incomplete restores refuse startup.

8.6 Upgrade and renew credentials deliberately

For a supported format-4 source, quiesce and stop all voters, explicitly configure `storage_format: 4`, and run `sqlodin migrate OLD.json NEW-DIRECTORY` separately for each voter. Preserve original files. Activate the compatible format-5 group together. Rollback to the originals is safe only before activating the migrated group; after new acknowledgements, a stale source is not a rollback target. No rolling upgrade is promised.

Certificate renewal also uses coordinated shutdown. Replace trust and leaf credentials on voters and clients, preserve configured principals, restart, and check fresh reads. Existing TLS sessions do not terminate merely because a certificate expires. Removing old trust and closing old sessions are both needed. Dynamic enrollment and membership changes are outside this fixed-voter procedure.

Follow [the operating contract](#) for the full backup, restore, migration and certificate conditions.

9 Formal verification

The previous chapters separated agreement, persistence, application order, reads and recovery. That separation also structures verification. A small model can expose a missing guard. An inductive proof can establish a property for arbitrary history lengths. A code-level test can check that an actual crash follows the intended storage boundary. None of these alone supplies all the others.

The retained evidence contains 59 model configurations, including required negative controls, and 48 discharged inductive proof obligations. Those counts identify completed work; they are not a measure of how much of the executable is proved. The [composition argument](#) records the connections between the claims and their implementation.

9.1 Begin with a state and an invariant

A specification names the state variables and allowed transitions. An invariant I is a property that every reachable state must satisfy. Induction has two obligations:

$$\begin{aligned} \text{Init}(s) &\Rightarrow I(s), \\ I(s) \wedge \text{Next}(s, s') &\Rightarrow I(s'). \end{aligned}$$

The first proves the initial state is safe. The second proves that each allowed step preserves safety. Together they cover any finite sequence of steps, not merely a long test run. A proof that omits a possible implementation transition does not cover that transition; the mapping from code to model is therefore part of the argument.

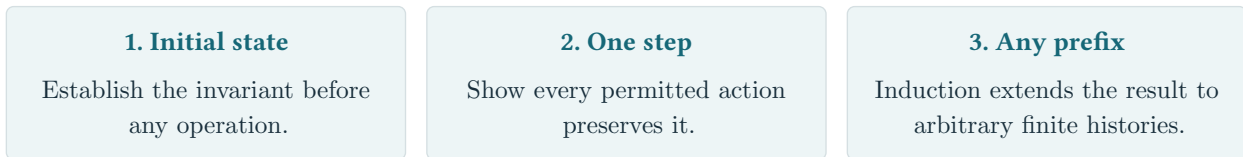


Figure 13: The number of executed test hours does not appear in an inductive safety argument.

9.2 Durable-history induction

DurableHistoryProof.tla distinguishes pending evidence, stable evidence, released evidence, chosen slots, applied slots and acknowledged slots. The invariant includes:

$$\begin{aligned} \text{released} &\subseteq \text{stable}, \\ \text{acknowledged} &\subseteq \text{applied} \subseteq \text{chosen}. \end{aligned}$$

It also requires every chosen slot to have a quorum whose evidence is stable. Here released/stable elements identify a slot and voter; applied/chosen elements identify slots. The different sets should not be confused with identical physical files.

The actions enforce the chain. Stage adds private evidence. Sync makes it stable. Send requires stable evidence. Choose requires a quorum of released evidence. Apply requires choice; Ack requires application. Crash removes pending evidence but not stable facts. Checking each action preserves the invariant gives the induction. TLAPS discharged 12 obligations for this abstraction.

This proof assumes one immutable value per chosen slot, correct quorum evidence and atomic application. It does not establish Paxos agreement or SQL determinism itself. Those premises come from Chapter 5, Chapter 6 and their model/code correspondence.

These proofs concern the abstractions, not the complete executable, compiler, OS or device.

9.3 Prefix recovery with unbounded slots

PrefixRecoveryProof.tla adds the part a set of chosen slots cannot express: a contiguous application prefix. Let c be the certified image cut, a the current replay/application cursor, k the durable committed prefix, h the acknowledged prefix, and T the retained suffix. Its invariant includes

$$0 \leq c \leq a \leq k, \quad 0 \leq h \leq k, \\ (c, k] \subseteq T, \quad \text{ready} \Rightarrow a = k.$$

The interval contains integer slot numbers. Every slot through k is chosen, and retained tail entries are chosen. After a crash, the cursor returns to c and readiness is false. Replay advances it one slot at a time through retained evidence. Opening the service requires $a = k$.

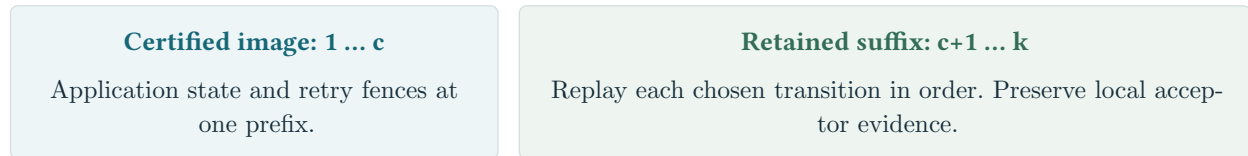


Figure 14: Trimming below a certified cut must leave the whole replay suffix above it. Publishing a new cut never moves beyond the committed prefix. Trimming removes only entries covered by the cut. Each action preserves the inequalities and suffix inclusion. TLAPS discharged 36 obligations for arbitrary natural-numbered slots.

The image is assumed to represent exactly its certified prefix. This proof does not inspect SQLite pages or authenticate receipts. Snapshot identity, logical verification, file durability and generation publication discharge those separate implementation obligations. Keeping the premise explicit prevents a short proof from appearing to prove more than it does.

9.4 Finite models explore the dangerous interleavings

TLC exhausts reachable states within a model's declared bounds. Positive cases must finish without the checked violation. A timeout is not success. Each negative control removes a specific condition and must produce the expected counterexample, rather than merely fail to parse or run out of resources.

Model	Property checked	Negative control
OwnedSlot	One value despite ownership and revocation ballots.	Lose durable votes.
RotatingWindow	Prefix progress and safe ring reuse under fair actions.	Remove skip, recovery or release; reuse a held slot.
RecoveryProgress	Repair advances through bounded history chunks.	Remove probes or stop after one chunk.
DurableEffects	Evidence is durable before it is sent.	Release before the durability gate.
ReadFence	Fresh marker is applied before the snapshot.	Reuse a marker or read early.
ReadCohort	Every shared read belongs to its fresh barrier.	Admit late readers or cancel another waiter's barrier.
SessionRetirement	Reclaimed sessions cannot replay old writes.	Lose or omit the epoch fence.

SOD 0005 adds three models and one inductive proof for the performance mechanisms that post-date the qualified release candidate. They are reproduced by the same runners:

Model	Property checked	Negative control
OwnedSkip	A no-op learned from the owner's Accept, or a value learned from this voter's vote plus the owner's, never disagrees with a decision.	Revoker offers any value; owner forgets its vote; learn without voting.

Model	Property checked	Negative control
JournalCache	An unsynchronized application database recovers every acknowledged outcome from the journal.	Apply before the journal barrier; trim without a durable image.
QuorumRead	A quorum frontier read observes every write completed before invocation.	Peers report applied prefixes; answer without a peer.
OwnedSkipProof	41 TLAPS obligations: an owner's round-zero no-op fixes the slot for unbounded ballots and values.	—

The rotating-window model explores independent delivery, learning and applied frontiers. Fairness represents eventual service and successful retransmission. It does not prove a millisecond deadline or progress during endless contention. The read models assume an immutable ordered log; they examine the barrier boundary, not the entire network protocol.

Model	Property checked	Negative control
SnapshotIdentity	A certificate has a distinct matching durable quorum.	Mix keys, count duplicates or issue volatile receipts.
ManifestDurability	Successful file publication survives modeled crashes.	Omit file or directory sync.
SnapshotPublication	Trimming follows recoverable publication.	Trim without its guard.
GenerationCatalog	Published state preserves promises, suffix and reserved IDs.	Publish early or lose local facts.
GenerationRetirement	Delete only eligible owned generations.	Delete current, predecessor or unowned paths.
ImageRetirement	Keep active and certificate-required images.	Drop predecessor or successor protection.
RestoredGenesis	A new namespace opens one durable initial state.	Reuse old identity, mix images or open early.

9.5 Connect the model to running code

A refinement map says which implementation event represents a model step. For example, a successful FULL journal commit advances stable evidence; releasing the owned message queue represents sending; the catalog commit selects the published generation. A failed or incomplete operation must not be treated as that successful abstract step.

Implementation tests exercise these boundaries. The final candidate passes 43 targeted SIGKILL cases for generation publication, retirement, backup and restore. Other retained campaigns inject ENOSPC, actual short writes and synchronization errors at exact WAL paths. They check that the affected voter does not acknowledge fabricated success, that survivors resolve the original identity, and that repaired state converges.

Transaction-history tests go beyond final row counts. They record invocation and response times, predicates and outcomes. A separate bounded reference model searches for a serial execution respecting every real-time edge. Negative histories include stale predicates and write skew. A final balance alone could miss both errors.

The final three-instance campaign also removes each voter in turn, writes through both survivors, rejects minority reads/writes, rejoins the missing voter and restarts the whole group. These tests connect protocol, storage and service code under actual process failure. They do not turn a finite experiment into a universal liveness theorem.

9.6 Reproduce the evidence

```
python3 tools/check_formal.py --jar /path/to/tla2tools.jar \  
  --output build/formal/new-model-run.json  
python3 tools/check_proofs.py --tools build/proof-tools \  
  --output build/formal/new-proof-run.json
```

The runners check pinned tool hashes and require fresh evidence paths. The source-bound [formal manifest](#) reuses only unchanged successful specifications. It is labelled as evidence reuse, not a new TLC or TLAPS execution. Exact commands and historical failures remain in the reports.

10 Work, memory and latency

Odin gives the host control over data layout and allocation. That control removes some overhead; it does not remove the need to persist votes, wait for a quorum or execute SQLite. To improve performance, first identify which resource limits progress.

10.1 Own the data that outlives a turn

The service uses a fixed connection array and a serialized owner for host state. Configuration lives in its own arena. Short-lived decoding and results use a temporary allocator. A payload borrowed from a decoder or protocol effect cannot remain borrowed after that storage is reused. Pending protocol and durable work therefore own their copies.

The consensus window is a fixed array of 64 active slots. With window size W and bounded value size B , its value storage grows as $O(WB)$, independent of lifetime history length. Slot reuse is tied to the applied floor; modulo indexing alone would overwrite a still-needed slot. SQLite, TLS, queues and worker state add allocations outside this array. Fixed capacity is not zero memory cost.

The mutation representation shares vector storage across columns instead of reserving a full embedding per scalar column. Small requests still carry and copy a bounded value through several layers. Those copies and JSON encoding remain measured costs, even when the consensus transition itself performs no heap allocation.

10.2 Admission protects the quorum path

The 32-connection pool admits at most 24 authenticated clients and four incoming handshakes/rejected-client connections. Fixed peers retain capacity. Each connection has a bounded frame buffer and at most 256 queued frames, also capped at 2 MiB total. JSON nesting is checked before recursive decoding.

The owner works in bounded turns. A turn steps the peer packets that arrived (up to 128, in groups of sixteen), admits up to sixteen already waiting writes, runs the timer and ownership progress, and crosses one journal barrier for all of them (SOD 0005). Already waiting reads share one quorum frontier, which needs no barrier. A peer link drains its output until the socket would block, and receives until the turn's packet buffer is full. Responses are flushed before the turn's barrier. This amortizes work without waiting for a batching timer to collect requests. A full consensus window keeps unproposed writes pending until their own timeout; history-space pressure still answers Busy at once. Slow-peer traffic beyond the byte bound can be dropped for retransmission rather than accumulated without limit.

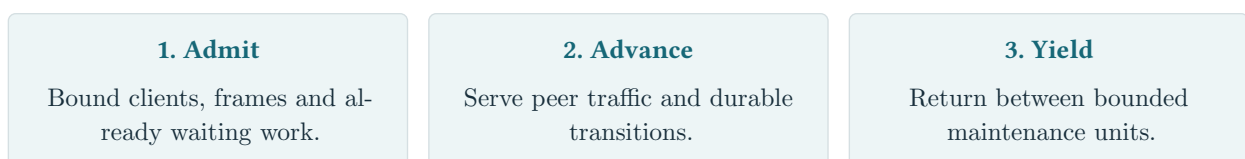


Figure 15: Bounds keep one queue from consuming all service turns. They do not preempt a blocked system call.

Maintenance uses one worker/job with bounded chunks and explicit cancellation. The final generation catch-up applies one transaction per owner turn. A transaction itself can still perform substantial I/O. A SQLite progress callback counts VM work; an optimized B-tree operation can do disk work inside one VM opcode. Neither count is a hard deadline.

10.3 A simple cost model

Let f be the time spent at a durability barrier and b the number of useful requests sharing it. That barrier contributes roughly $\frac{f}{b}$ per request when amortized. This is not a throughput formula for the entire database: a write crosses multiple ordered stages, and quorum waits, application work and queueing can dominate elsewhere.

Client latency includes admission, protocol progress, durable voting, earlier-slot completion, local application and the response path. A healthy owner may choose a value in one quorum round trip. That does not make the client operation a one-RTT transaction.

In a stable workload, Little's law relates average in-flight work L , completed rate λ and average response time T :

$$L = \lambda T.$$

Increasing clients can expose more batching. Beyond the bottleneck it mainly grows waiting time. This relation assumes a stable measurement interval; it should not be used to infer capacity from a growing backlog or a failed run.

Low CPU utilization is compatible with poor latency when threads wait on storage. High throughput with weaker synchronization is a different durability contract. Chapter 11 reports both measured rates and persistence boundaries, so an optimization cannot appear successful merely by changing what is being measured.

10.4 What the resource checks show

The corrected bounded load test exercises full client admission, slow consumers with 120,000-byte responses, excess clients, stalled handshakes, writes and live maintenance. It records 3,840 operations and roughly 29–35 MiB peak ordinary voter RSS. Instrumented runs measure allocations separately because instrumentation changes timing.

These observations support the tested short-query profile. They do not establish a maximum database size or immunity to arbitrary hostile SQL. The service fails closed on unknown replicated execution/storage errors. If all voters lack resources for the same chosen write, quorum alone cannot make that write executable.

The [resource contract](#) defines bounds and fault handling. The [batching argument](#) records why the optimizations preserve ordering and where performance remains below its goals.

10.5 Embed the durable host

An Odin embedding owns the responsibilities that `sqlodin` serve normally supplies. Serialize access to each host. Drive peer delivery and ticks, consume outgoing packets, and stop serving when an unknown storage failure poisons the host. Do not call SQLite through a second writer or mutate its metadata outside the ordered host.

Host operation	Caller responsibility
<code>open_store</code> , <code>close</code>	Own the locked format-5 directory and release resources explicitly.
<code>propose</code> , <code>propose_batch</code>	Keep the request identity; a returned slot is not success.
<code>step</code> , <code>step_batch</code> , <code>tick</code>	Drive protocol progress and handle copied output before reuse.
<code>outcome</code> , <code>acknowledged</code>	Match the expected chosen request, not merely a slot number.
<code>begin_read</code> , <code>poll_read</code>	Use a fresh single-use ticket for a quorum-backed read.
<code>next_id</code>	Use durable reservations when IDs must survive restart.

The concrete entry points are in `src/durable/`; `tests/test_durable.odin` exercises their lifecycle. The in-memory example under `examples/` teaches message flow but has a different policy and no disk durability. Do not use its completion condition as the storage contract for an embedded production host.

11 Performance evaluation

A benchmark answers a question about a particular workload, build and machine. It does not rank database languages. This chapter separates SQLodin's durable workload matrix from an earlier comparison with other systems. Tables read the retained JSON directly. Failures stay in the evidence even when a chart shows only completed repetitions.

11.1 The durable workload matrix

The designated benchmark instance is Linux host .18. Three native mTLS voters use separate directories on persistent ZFS storage. The matrix uses SQLite FULL durability for application and consensus state, and fresh barriers for reads. It covers 1/8/32/64 clients; 70/30, 50/50, 95/5 and pure-write mixes; 1–8 statements; 1–4 row changes; 256-byte and 4 KiB values; skew; and entry through every voter.

The 32 cases verify 91840 operations. The source reports identify the binary, workload, seeds, individual latency distributions and resource counters. The matrix predates the final maintenance-budget and one-transaction replay scheduling fixes. It is performance evidence for that recorded candidate, not a fresh timing measurement of the final artifact.

The SQLite reference runs the matched SQL and schema with FULL synchronization and bounded grouping of up to sixteen already waiting requests. It is one engine with no TLS or replication. That difference is the baseline's purpose and must remain visible.

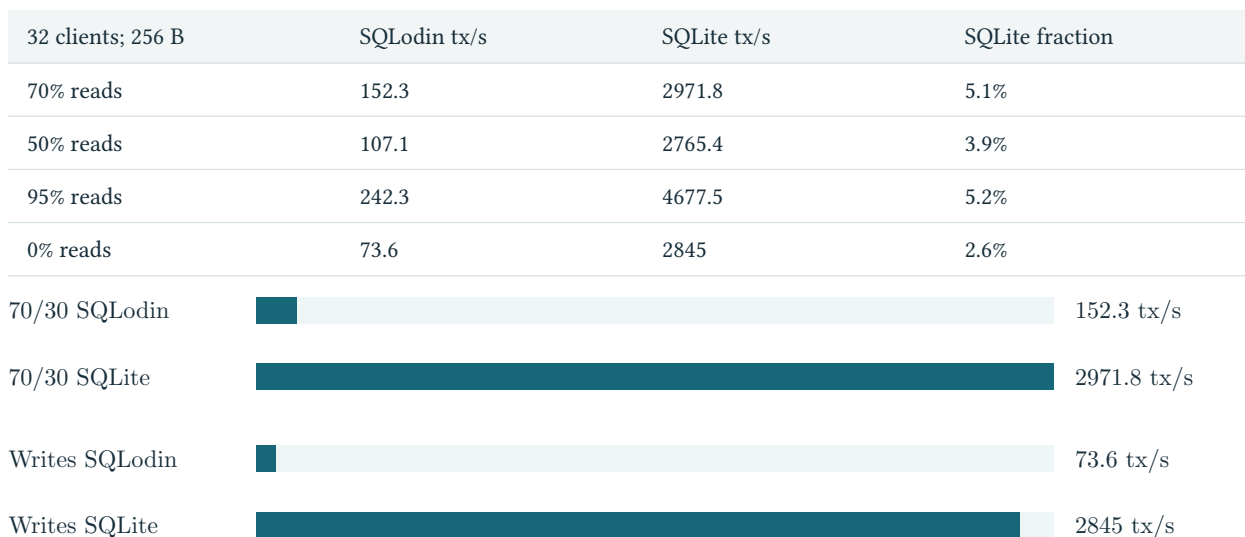


Figure 16: Matched durable SQL, different system boundaries. Both pairs use one shared linear scale. The 70/30 case completes 46.4 successful writes/s within 152.3 total transactions/s. Its read/write p99 latencies are 397.1 / 471.8 ms. The pure-write case completes 73.6 writes/s with 1230.8 ms write p99. These are measured cases, not maxima.

Closed-loop latency starts at actual invocation. Scheduled-arrival cases include waiting from the offered arrival time. Do not mix the two distributions: a closed-loop client stops generating arrivals while it waits. The reports name the latency basis explicitly.

11.2 Targets remain visible

Original provisional goal	Disposition
3,000 mixed tx/s; 900 writes/s	Not achieved; future improvement goal.
1,000 pure writes/s	Not achieved; future improvement goal.
Read/write p99 $\leq$ 20/50 ms	Not achieved; future improvement goal.
At least 25% of matched SQLite rate	Not achieved in the matrix.

The owner accepted release after correctness qualification with these shortfalls disclosed. The original numbers were not changed into lower passing thresholds. This release should not be selected on the assumption that those rates or latencies have been delivered.

Whole-process mean synchronization times in the matrix range from roughly 5.5 to 21.2 ms. Low CPU use alongside these waits points toward durability and coordination costs, not proof of an efficient or inefficient compiler. The observations include setup, validation and shutdown where the report says so. They are not isolated SQL-statement measurements.

The earlier healthy-path 100 ms ownership wait has a deterministic no-tick regression. Missing-owner recovery still uses failure detection. Changing one timer or language cannot remove storage barriers, prefix dependencies and application work at the same time.

11.3 After SOD 0005: fewer sequential barriers

Review note: the original three-host worker could swallow thread failures. Its historical throughput figures below are provisional; they are not corrected-harness results. The separate calibration harness propagates worker failures. SOD 0005 records the correction and its evidence.

The measurements above describe the qualified release candidate. SOD 0005 traced up to seven sequential sync barriers per write and per fresh read. It then introduced one barrier per service turn, one-message learning of owner no-ops (and, with three voters, of values this voter also voted for), a WAL NORMAL application cache of the FULL journal, and quorum-frontier reads. The same calibration matrix, with SQLite measured in the same run on .18, moved from 1.7–10.5% to 6.8–40.6% of SQLite; the absolute gain is 1.9–10.5 $\times$ per case. On three separate hosts, 32-client pure writes rose from 95 to 784 per second (medians), and a sequential fresh read fell from 44.6 to 0.51 ms. Thirty-two-client pure writes remain far below the 25% goal on the shared-disk host. The reports, including failed attempts, are in `benchmarks/results/sod-0005/`, and SOD 0005 records the method and the remaining gaps.

11.4 Earlier cross-system evaluation

The retained comparison ran three voter processes per system on .18, using native clients and persistent directories. It measured SQLodin format 4 / policy 6, Zaxonlite v0.7.0, rqlite v10.2.7 and the stock go-cowsql v1.22.0 demo with libcowsql v1.15.9. These are historical builds; the chart is not a comparison of today's releases.

The mixed workload uses order, inventory and ledger changes with indexed reads and dashboard queries. Each measured phase has 400 operations after 100 warmups, with 4 client workers and 3 attempted repetitions. SQLodin admits through different voters. The cowsql demo has no matching arbitrary-SQL endpoint, so it is not forced into this mixed profile.

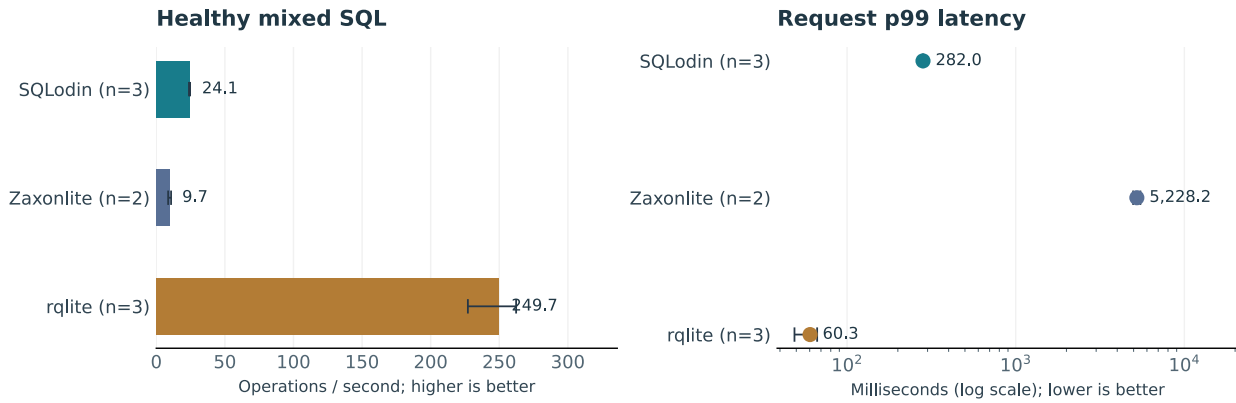


Figure 17: Historical mixed-SQL comparison. Successful sample counts are shown in the chart.

Whiskers are observed ranges, not confidence intervals.

One Zaxonlite mixed repetition failed during setup with a malformed database error. Stopped-database checks confirmed the error on two replicas; its cause was not established. Two successful mixed samples appear in its chart, versus three for SQLodin and rqlite. This is neither an all-attempt success rate nor a general reliability ranking.

Persistence differs. SQLodin uses FULL application and Paxos commits. That Zaxonlite build uses a full-sync Paxos log and SQLite WAL NORMAL. rqlite uses persistent Raft state and on-disk SQLite. TLS/HTTP transport and product batching remain part of each result. A common SQL statement does not make the persistence paths identical.

The common sequential workload inserts 256-byte values. It includes cowsql only through its supported HTTP PUT/GET demo. The demo keeps its SQLite image in memory while persisting Raft logs and snapshots on disk. That is durable replicated state, but not an on-disk SQLite application image like SQLodin's.

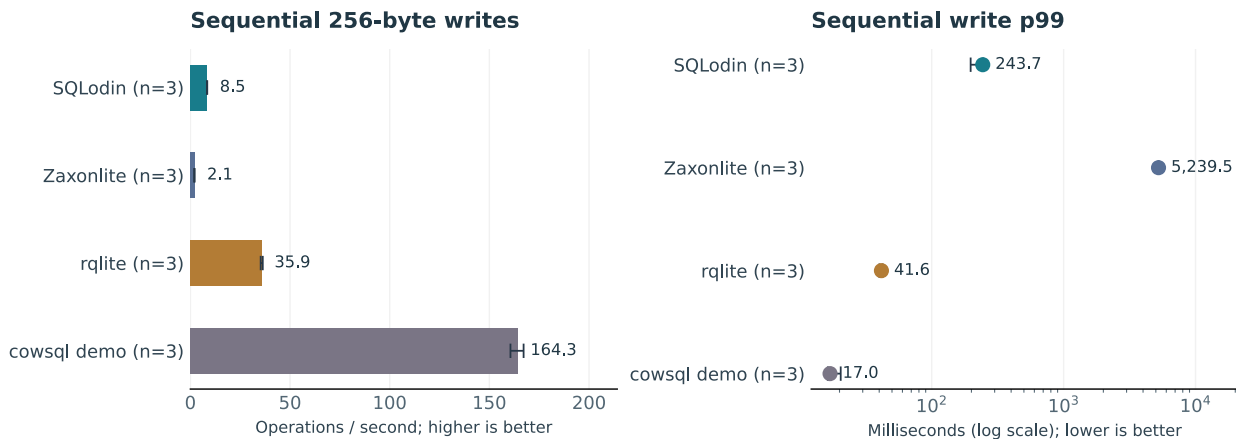


Figure 18: Historical sequential writes. Product-specific persistence remains part of the result; the latency axis is logarithmic.

Cowsql's full-cluster restart verification succeeded in 3 of 3 repetitions. Those restart results are separate from throughput. Other completed runs also check acknowledged state after voter loss and restart. Raw failures and sample hashes remain in `benchmarks/results/linux18-native-comparison-complete.json`.

11.5 Capacity, recovery and topology

Lightweight current-candidate checks complete snapshot publication, restart and exact key counts. The 64 MiB fixture restarts in 1.32 seconds. A prior candidate completed a 10 GiB payload run and restarted in 14.89 seconds. Retained larger-store continuations took 114 and 137 seconds to become ready, exceeding the original 60-second goal. Integrity scanning dominated those observations.

The later growth campaign was stopped at the owner's request at its last saved 32 GiB checkpoint. It is interrupted evidence, not a passed 64 GiB test. Large-capacity campaigns and fixed soak durations

are not release requirements. Targeted proofs and fault tests support the stated release scope; they do not imply an unmeasured capacity or recovery SLA.

The three-instance fault checks use .19, .20 and .21. Their first writes after removing each voter take about 2.37, 1.24 and 2.48 seconds. This meets the five-second goal in that campaign, not under every disk or network delay. Physical failure-domain independence, exclusive hardware and device power-loss protection were not established. Shared ZFS I/O pressure was observed during large-data work.

Use [the benchmark guide](#) for reproduction and the [release manifest](#) for the exact evidence binding. Historical memory-only and embedded runs remain in the archive; they must not be combined with network/durable measurements into a single ranking.

12 Reference and source map

This chapter collects names and limits used earlier. The operating guides contain the full command syntax. The specifications define the contracts; the evidence reports identify which source and binary were checked.

12.1 Native commands

Command	Purpose
sqlodin local FILE	Open a standalone SQLite file; bypass replication.
sqlodin connect CLIENT.json	Open the interactive or scriptable cluster client.
sqlodin serve NODE.json	Reopen and serve one durable voter.
serve NODE.json --create	Create new state once; refuse an existing store.
request CLIENT.json REQUEST.json	Send one explicit authenticated protocol request.
backup NODE.json NEW-DIR	Take a verified offline application backup.
verify-backup DIR	Check manifest, bytes and application integrity.
restore BACKUP NODE.json --new-cluster	Restore into a fenced fresh namespace.
migrate NODE.json NEW-DIR	Perform supported offline format-4 migration.
compact NODE.json	Perform offline certified generation compaction.
version, help	Inspect the binary and available syntax.

12.2 Interactive client

SQL ends with a semicolon. BEGIN, COMMIT, ROLLBACK and savepoints use the optimistic transaction protocol. Without BEGIN, each write is its own replicated request.

Command family	Use
.help	List the supported dot commands.
.mode, .headers, .nullvalue	Choose presentation and null formatting.
.tables, .schema, .indexes	Inspect the SQL catalog.
.read, .output, .once	Run scripts or direct output.
.timeout, .consistency	Set request waiting and explicit read consistency.
.status, .nodes	Inspect local state and fixed membership.
.pending, .retry	Inspect or resolve the exact saved uncertain request.
.reconnect CLIENT.json	Change endpoint within the same cluster and identity.
.session, .retire-sessions E --quiesced	Inspect session state or explicitly advance its epoch.

The shell stores pending writes durably before sending them. --state PATH selects its private recovery file. Do not share that file between simultaneous shells or delete it to escape an uncertain result. .pending can reveal application values.

Diagnostics name the error and suggest a correction. Data goes to stdout or the selected output; errors go to stderr. Terminal color is disabled for redirected streams, TERM=dumb or NO_COLOR. CSV and JSON preserve values; terminal display escapes control bytes. The [CLI guide](#) covers editing keys, scripts and output modes.

12.3 Default service bounds

Resource	Default bound
Write transaction	8 statements; 4,096 SQL bytes; 16 parameters.
Text parameter	256 UTF-8 bytes.
Vectors	384 float32 components per request; at most 384 per vector.
Read result	4,096 rows and 256 KiB accounted storage; whole-result failure on overflow.
Read work	Approximate one-million SQLite VM instruction budget.
Client request / response	64 KiB / 1 MiB wire limit.
Connections	32 total; 24 admitted clients; peer capacity reserved.
Queued output	256 frames and at most 2 MiB per connection.
Consensus window	64 active slots, reused only after safe floor advancement.
Application/journal grouping	16 requests per application group; one journal group per service turn, flushed early at 1,024 records or effect capacity.
Sessions	65,536 rows in the current epoch; explicit replicated retirement.
Maintenance	One job; 1 MiB chunks; 32 MiB transfer buffers.
Consensus history	8 GiB cap with reserves; separate from application/staging capacity.

Bounds describe the default build and admitted API. They do not bound arbitrary SQL's wall-clock cost or the operating system's buffers and cache. Arbitrary BLOB parameters, attached databases, extension loading, dynamic membership and rolling upgrades are outside the supported interface. Vector BLOB results and typed vector inputs are supported.

12.4 Terms used in the book

Term	Meaning
Slot	One position in the replicated total order.
Ballot	An ordered proposal attempt; distinct from a log slot.
Chosen	Accepted by a quorum under the protocol.
Applied prefix	The contiguous chosen history reflected in local application state.
Acknowledged	The service has verified durable choice and application of the expected request.
Read frontier	The largest highest-seen slot of a read quorum, observed after a read's invocation; the snapshot waits for it.
Read marker	A fresh ordered value used by the embedded durable host to authorize a read snapshot.
Revision	An application-state counter used for optimistic validation.
Epoch	A replicated fence that prevents reclaimed retry sessions from reviving.
Generation	A recoverable application/consensus pair selected by the root catalog.
Certificate	Distinct matching durable snapshot receipts from a quorum.

12.5 Find the implementation or argument

Concern	Source	Argument
Consensus adapter	src/paxos.odin and deps/paxos-odin/	Multi-master refinement
SQL and grouping	src/engine*.odin	SQL policy grouping
Reads and transactions	src/durable/reads.odin, service/read_batch.odin	Ordering
Durable lifecycle	src/durable/, src/snapshot/	Publication retirement
Operator recovery	cli/backup.odin, cli/restore.odin	Recovery procedures
Networking and bounds	service/, transport/mtls/	Resource contract
Application clients	cli/, languages/python/	ORM contract

The [SOD index](#) records design decisions. The [release record](#) records qualification and known limits. The [documentation index](#) separates current guides from historical notes. Use those records to follow a claim to its source rather than treating the book's prose as a substitute for executable evidence.