

R1.4 real-time read and transaction order

This argument assumes the agreed immutable Paxos log, durable acknowledgement boundary, and deterministic SQL transition contract checked separately by R1/R2. It establishes the read/transaction ordering consequence of those assumptions; it is not a proof of the compiler, network or arbitrary SQLite execution.

Fresh reads (service): quorum frontier

Since SOD 0005 the native service crosses a quorum-read barrier. `begin_read_cohort` closes a cohort of already accepted reads, then observes this voter's `highest_seen` and sends a peer-only frontier request. Replies from `read_quorum - 1` distinct peers, each for this cohort's token and each produced after the cohort closed, give the frontier H (their maximum with the local value). `finish_read_cohort` answers the members once the contiguous applied prefix reaches H , before any further application or consensus transition.

Let write w be acknowledged before a member's invocation. It was chosen at slot s , so a write quorum holds durable votes for s . The queried set is a read quorum and intersects it. `highest_seen` is at least every slot a voter has durably voted for or decided, never decreases within a process, and resumes above the durable ledger after restart. Every observation follows invocation, so $H \geq s$ and the snapshot includes w . If an earlier read returned a prefix A , every slot up to A was chosen before the later invocation, so $H \geq A$. The snapshot may include later concurrent decisions; that only moves its linearization point forward within the invocation/response interval. No journal write or sync barrier is involved. `QuorumRead.tla` checks this argument, with negative controls that report applied prefixes instead of `highest_seen` or answer without a peer. Unanswered requests repeat every 200 ms, and a minority cannot complete a frontier.

Fresh reads (embedded durable host): markers

`durable.begin_read` allocates a new durable marker identity after invocation. The service may share that marker among a closed cohort of already accepted reads; every member invocation precedes marker allocation. Later arrivals wait for another marker, even if the preceding cohort has not returned yet. `poll_read` checks the chosen value equals that marker and the contiguous local application prefix includes it. A displaced proposal cannot authorize a read. The owning host consumes the ticket once, including when the local query fails. An acknowledged write completed before invocation already occupies an immutable slot in the log. Quorum intersection and contiguous application ensure that a successful fresh marker cannot bypass that write.

The serialized service owner processes no other state transition between checking the marker and opening/executing the actual query. The read snapshot may include later concurrent decisions already applied locally; that only moves its valid linearization point forward within its invocation/response interval. A minority cannot complete a new marker. Explicit local reads intentionally omit this guarantee and are labelled as such in the API.

Optimistic SQL and predicate reads

`transaction_result` obtains the revision after a fresh marker. Every preview also crosses a fresh marker, opens the active generation's application database, and checks its revision equals the transaction's original revision. The service owner is serialized: no application commit can interleave the revision comparison and preview snapshot acquisition. The preview replays the staged body privately, then reads its effects, and always rolls back. No mutable preview workspace or SQLite transaction is retained between requests.

An applied application transaction advances the global revision atomically with its data and request outcome. For a separated store, NORMAL application commits are recovered from the retained FULL

journal under SOD 0005 J1–J3. Duplicate/rejected requests and read markers do not advance it. Explicit session retirement also advances it. At the ordered write slot, `engine_run_outcome` compares the original revision before executing any SQL. If any application transition intervened, the transaction rejects with Conflict. This conservative whole-database validation covers predicate reads and phantoms as well as the keys a transaction eventually changes. It trades concurrency for a simple serializability boundary; disjoint writes may conflict.

For a successful writing transaction, all preview reads therefore describe the same committed predecessor, combined with that transaction's own staged effects. Place it at its applied log slot. For a read-only transaction, all successful previews use the same revision, so place it at a successful read snapshot. Its client-side commit need not create another write. A subsequent changed-revision read aborts the transaction. Savepoint operations only change the staged body; release does not publish it or acknowledge a durable commit.

Consequently successful transactions admit a serial order respecting completed- before-invoked real-time edges. Writes that overlap a read-only transaction after its last read may follow that transaction in the serial order. Unknown commit outcomes retain their request identity and require resolution under R1.3; they must not be treated as either a definite abort or a new transaction.

Executable evidence

`tools/check_transaction_history.py` records monotonic invocation/response times, successful predicate observations before and after staged changes, conflicts, contacted voters and restart/rejoin reads. For each bounded concurrent batch it exhaustively searches serial executions of a separate four-row set model, respecting every response-before-invocation edge. A final nonconcurrent predicate read anchors the resulting state between batches. It never accepts a batch merely because the final row count matches. Raw histories and a serial witness are saved.

Negative controls reject a stale predicate after a completed write and committed write skew. Native checks cover each voter absent, every surviving entry point, rejoin, full restart, and minority/stale-read rejection. Existing ORM regressions cover savepoints, repeatable reads, disjoint-write conflicts and pending commits. The bounded ReadFence model supplies stale-marker and early-snapshot negative controls, with concrete host regressions. These tests supplement the argument; a finite history search does not prove all possible executions.

The Linux reports `transaction-history-linux.json` and `transaction-history-readonly-linux.json` each contain sixteen concurrent batches and their serial witnesses, plus rejoin/restart reads. They pass two seeds on the recorded disk-backed filesystem. Light local reports check four batches each. `source-manifest-sql-contract-linux.json` identifies the native candidate and source. Together with the existing model and native/ORM regressions, this closes R1.4; release-candidate integration remains R7. No elapsed soak requirement is imposed.

Closed-cohort refinement

`service/read_batch.odin` closes membership in `begin_read_cohort` on the serialized owner, with no connection dispatch between the membership scan and the frontier observation. `finish_read_cohort` executes each member snapshot before another application/consensus transition. A reply for another cohort token, or a second reply from the same peer, is ignored. `release_read` removes one waiter without cancelling other members; only the last member retires the cohort. Connection admission bounds cohort size. Generation publication retains the host's active ticket and applied-prefix checks.

The ReadCohort model checks 180,443 reachable states under the immutable-log premises above. Three required negative controls expose late membership, reading before marker application, and

cancellation that invalidates another waiter. `formal-read-cohort-local.json` records all four results. Concrete regressions in `tests/test_service_read_batch.odin` require quorum, preserve another waiter after cancellation, and require a later read following a completed write to use a newer marker and observe that write. Cohorts change amortized barrier cost, not the freshness contract. The same regressions now cover the quorum frontier: a reply for another cohort does not count, and a voter that missed a write acknowledged by the other two waits for the frontier before answering (`test_read_cohort_waits_for_frontier_above_write_acknowledged_elsewhere`).

Reconnect identity boundary

A read quorum counts distinct configured voter identities, including the requester, not distinct sockets. Cohort-level membership survives connection replacement. `QuorumReadReconnect` checks this five-voter seam; `QuorumReadConnectionCount` reproduces a stale read when a stale peer reconnects and is counted twice. The implementation regression uses two connection objects for the same peer. This extends the host-seam evidence, not the qualified deployment membership.