

Session retirement under R1.3

A replicated monotonically increasing epoch fences all request identities from older epochs. Within the current epoch, the existing 65,536-session bound and strict per-session sequence rule remain. The application stores one epoch scalar with its transaction revision. A chosen retirement command atomically increments that scalar and deletes the old session rows. It is a consensus operation, never a local eviction or a wall-clock decision.

Every transaction carries its epoch as part of its request identity and content hash. Before session lookup, an older epoch is rejected as Expired; a future epoch is rejected without consuming a sequence. No absent row can revive an old request. The rejection applies after restart, snapshot installation and restored genesis. The memory/disk bound is the current epoch's bounded session table plus one scalar, not an accumulating set of retirement tombstones.

Retirement is an explicit administrative action with an expected epoch, not an automatic reaction to a temporary capacity spike. Operators first quiesce clients and resolve outstanding results.

Advancing epoch e to $e+1$ is idempotent while the current epoch is $e+1$; a stale command cannot advance it again. A client with an unresolved old request must retain that identity and receive Expired. It must never silently relabel that write into the new epoch. A new session may discover the epoch through a fresh quorum read before its first write.

A retirement command also advances the application revision, invalidating old optimistic transactions. Application state, epoch, session deletion, outcome and applied prefix commit together, including grouped application. Concurrent old requests are ordered either before retirement (at most one effect) or after it (Expired); new-epoch requests execute only after the fence. Epoch overflow refuses retirement. The model and executable tests must cover both orderings, duplicate retirement, full capacity, restart and stale retries after row reclamation.

This implements an existing criterion. The evidence and limits below distinguish the session lifecycle argument from the remaining SQL and consensus criteria.

Public operations and compatibility

Python connections discover the current epoch through a fresh quorum barrier before their first write. `PendingWrite.epoch` is serialized and immutable during recovery; saved pre-epoch requests decode as epoch zero. Optimistic transaction begin returns the epoch with its fresh revision. An unavailable discovery is an unsent request, not an unknown write. `session_epoch()` reads the current fence; `retire_sessions(expected_epoch=e)` explicitly advances it. After retirement, use a new connection for new work and retain expired pending identities for business-level diagnosis. A future epoch is also rejected as Expired: only the current epoch is admissible.

The CLI exposes `.session` and `.retire-sessions E --quiesced`. It persists the epoch with its private request state before sending a write. A used state file never adopts a newer epoch automatically. Its Expired diagnostic directs the operator to retain the saved identity and avoid replaying it under a new one.

Policy 7 and peer wire version 3 identify this behavior. The value codec has an explicit epoch word. Recovery can decode and preserve old policy-6 journal bytes; ordinary startup refuses a policy mismatch. Explicit format-4 migration accepts policy 6 and builds canonical policy-7 metadata in a new destination, preserving the source. A policy-6 format-5 backup can be restored into a fenced fresh namespace using the existing recovery procedure. Epoch-zero content hashes remain unchanged, so retained original retry outcomes still match. Every newer epoch has a distinct hash domain. There is no rolling upgrade promise.

Safety argument and test map

Induct on the ordered applied prefix. Before retirement, a session's latest sequence fences smaller sequences and returns the same stored outcome for an identical retry. Retirement deletes those rows only in the same transaction that advances the epoch. The epoch never decreases or wraps. Every erased request thus fails the epoch check at all later prefixes, irrespective of how many subsequent epochs exist. A new epoch deliberately defines different request identities. Crash recovery and snapshots preserve the scalar in the same application image as the session rows; restore preserves both. Grouped application applies these same transitions in order before its one durable commit. Storage failures halt without acknowledging incomplete state. This argument does not require retaining one tombstone per historical request.

`SessionRetirement.tla` checks this epoch seam for two sessions and three epochs (84 states). Negative controls remove the guard, lose the durable epoch, repeat a stale retirement or keep retired rows. It abstracts a completed SQLite transaction as one durable step; process-kill tests cover the implementation boundary. It does not replace R2's consensus refinement argument or prove the storage device.

Implementation maps to `engine_session_epoch.odin`, `engine_sessions.odin`, the existing individual/group outcome commits, and `service/sessions.odin`. Tests cover all 65,536 session rows, reclamation/retry boundaries, grouped/reference ordering, restart, cross-voter operations, an absent voter, compaction, CLI recovery state, backup/restore and the old binary's migration path. Raw reports and failed fixtures are retained under `benchmarks/results/verification-20260924/`. R1.3 is complete with the evidence recorded in the fixed release checklist. R1.1, R1.2 and R1.4 remain separate existing criteria; no additional release gate is introduced.