

SurrealMX

An in-memory MVCC engine
with lock-free reads



SurrealMX: An In-Memory MVCC Engine with Lock-Free Reads

Tobie Morgan Hitchcock Emmanuel Keller Stu Schwartz Farhan Khan

SurrealDB

October 2026

Describes SurrealMX 1.0.0

Abstract

SurrealMX is an embedded, in-memory key-value engine written in Rust. This paper explains how it attains its throughput. Keys live in a concurrent adaptive radix tree, and each key holds a short chain of immutable versions, so a read takes no lock and writes no shared tree memory: its snapshot is a pair of counters in a slot that only its own transaction writes. Writers claim their place in a commit ring with one atomic increment, validate against the commits in their window, become visible in one order through a second ring, and are then applied to the tree. Garbage is trimmed by the writer that makes it. Transactions validate their reads at commit by default and can ask for plain snapshot isolation. Persistence is optional. On one 32-core machine with 15 million keys, taking the best of three runs of each benchmark, SurrealMX’s direct operations read more than 17 times faster than LMDB and more than 28 times faster than RocksDB, both running in memory. We set out each design decision, what it spares and what it costs, state the invariants the design relies on, compare it with Hekaton and Silo, and say what the measurement does and does not cover.

1 INTRODUCTION

An in-memory engine is judged on how many operations it completes as cores are added, and on what one transaction is allowed to see of another. SurrealMX is the in-memory storage engine of SurrealDB and also a standalone embedded engine, for a server or, through WebAssembly, a browser. Under SurrealDB every query reaches it, so its throughput is inherited by everything above it.

Thesis. SurrealMX’s throughput is the product of a small number of decisions, each of which removes a shared write from a hot path. Keys live in a concurrent adaptive radix tree [9], and each key holds a chain of immutable versions behind an atomic pointer, so readers neither lock nor write shared memory. A snapshot is two counters in a slot that only its own transaction writes, so starting and finishing a transaction touches no shared cache line. Commits are ordered by two fixed-size rings, so there is no commit lock. The writer that creates old versions removes them, so no pass has to scan the dataset. We describe each decision, say what it spares and what it costs, and state where the evidence is a measurement and where it is only the structure of the code.

Contributions. This paper makes five contributions.

- An account of the read path and the version structures that let a read take no lock and write no shared tree memory (Section 3).
- An account of transactions: the reader registry, conflict detection by a commit ring, visibility through a second ring, and the four invariants the design relies on (Section 4).

- A cost model and a table that sets each throughput-determining decision against what it spares and what it costs (Sections 6 and 7).
- A comparison of the design with two earlier in-memory transaction engines, Hekaton and Silo, and with LMDB and RocksDB, the two engines it is measured against in memory (Section 2).
- A statement of how the engine is tested, a measurement against in-memory LMDB and RocksDB that says what it does and does not show, and the limits of the design (Sections 9, 10, 11 and 12).

Scope. This paper describes SurrealMX 1.0.0. We state what the code does. The underlying techniques are established in the literature we cite; the contribution is how they are combined, what the combination costs, and what has been measured.

2 BACKGROUND AND RELATED WORK

Radix trees. The adaptive radix tree [9] sizes each node to the number of children it has, so a lookup costs time proportional to the key’s length and the tree stays compact. Its authors later showed how to synchronise it with optimistic lock coupling [10]. SurrealMX uses the `artmap` crate, a concurrent adaptive radix tree built that way. Its documentation states that readers never write to shared tree memory and never block, and that writers latch only the nodes they change.

Multi-version concurrency control and isolation. Multi-version concurrency control keeps several versions of a record so that a reader can see a consistent snapshot while writers proceed; the design space is wide [12]. Snapshot isolation [2] lets each transaction read from the snapshot taken when it began and aborts a transaction whose writes collide with a concurrent commit. It permits write

skew. Serializable snapshot isolation [3] closes that gap by tracking read-write dependencies between concurrent transactions. We call a read *linearizable* [8] when it returns a value that was current at some instant during the call.

LMDB. LMDB [4] serves reads from a memory-mapped copy-on-write B+tree and admits one writer at a time, through a writer mutex. Its readers do not take that mutex. Each active reader owns a slot in a reader table of cache-aligned memory, acquired once per thread and reused, in which it records the transaction it reads from at the start of a read transaction and clears it at the end; in normal operation nothing blocks a reader. A writer copies a page before changing it, and reuses a freed page only when it is older than every snapshot recorded in the table, so one long-lived reader stops page reclamation and the writer allocates new pages meanwhile. For the comparison in Section 10 the harness runs LMDB from `/dev/shm` with the `NO_SYNC` and `NO_META_SYNC` flags set, so nothing is flushed, and with `WRITE_MAP` and `MAP_ASYNC`, so writes go through the memory map.

RocksDB. RocksDB [7] is a log-structured merge-tree: writes go to a memtable and a write-ahead log, and sorted tables in levels are merged in the background. Its commit pipeline groups concurrent commits into a batch group whose leader writes the group to the log, after which the batches are applied to the memtable concurrently and one bump of the visible sequence number makes the group visible [5]. Every record carries a sequence number, and those numbers give reads their snapshot views. In the harness’s in-memory mode RocksDB runs over its in-memory environment, so its files live in memory, and the adapter disables the write-ahead log on every write. The adapter uses optimistic transactions that take a snapshot, a write buffer of 256 MiB with up to 32 buffers, concurrent and pipelined memtable writes and level compaction.

Hekaton. Hekaton [6] is the memory-optimised engine of SQL Server. It uses latch-free data structures, hash indexes and Bw-trees, and an optimistic multi-version scheme in which every version carries begin and end timestamps drawn from a monotonically increasing counter. It offers snapshot, repeatable-read and serializable isolation. At commit, a serializable transaction re-checks that the versions it read are still visible as of its end timestamp and repeats its scans to find phantoms. Worker threads clean up obsolete versions cooperatively, and durability comes from logging and checkpointing: a transaction is irreversibly committed once its updates are logged.

Silo. Silo [11] is a single-version in-memory database built around an optimistic commit protocol that gives serializability while avoiding every shared-memory write for a record that was only read, and any centralised transaction identifier. Each worker chooses its transactions’ identifiers; a global epoch, advanced every 40 ms, orders transactions across epoch boundaries; and results are released to clients only when their epoch is durable. Phantoms are

detected by re-checking the version numbers of the index leaves a scan touched. Read-only transactions can run on a slightly stale snapshot, for which Silo keeps extra versions, and these never abort. Memory is reclaimed by epochs.

Comparison. Table 1 sets SurrealMX beside the two. Two differences stand out. SurrealMX has the central counters that Silo set out to remove: a commit claims a ring slot and a merge version by atomic increments, and visibility is published in claim order. In exchange its read-only transactions read the newest published state and never abort, where Silo gives them slightly stale snapshots to achieve that. A SurrealMX read-only transaction runs under snapshot isolation and is never validated; Silo’s are serializable. And SurrealMX makes durability an optional layer behind the commit, where both papers make it part of commit. The engine’s default mode validates reads and scans against concurrent commits’ write sets, which is closer to the validation in these two papers than to the dependency tracking of serializable snapshot isolation [3] from which its name is taken. We have not run the three engines against one another, and the papers’ throughput figures describe other workloads on other hardware, so we compare designs and not numbers.

Against LMDB and RocksDB. Table 2 sets SurrealMX beside the two engines it is measured against. SurrealMX’s reader registry is the nearest relative of LMDB’s reader table: each records, in a slot that only the reader writes, which snapshot the reader is using, so that reclamation can wait for the readers that might still see old data. LMDB’s slot belongs to a thread and the single writer reclaims pages when it allocates; SurrealMX’s slot belongs to a transaction object and every writer that makes garbage reclaims it. Against RocksDB the contrast is structural. SurrealMX has no memtable, no sorted tables and no compaction, and replaces the leader of a batch group with two rings. Its log, when enabled, is an optional layer behind the commit (Section 8), where RocksDB’s write-ahead log is part of the commit pipeline.

3 DESIGN

3.1 Structure

Figure 1 shows the parts. Applications reach the engine through direct operations on the database or through transactions, and both use the same concurrency control and the same datastore.

Each key in the tree maps to a version cell. A cell holds a lock that only writers of that key take, and a pointer to an immutable value that is one of three things: empty, a single version stored in place, or an ordered chain of versions. A writer takes the cell’s lock, copies the current chain, changes the copy and swaps it in. A replaced chain is freed by epoch-based reclamation once no pinned reader can still hold it. At rest a key usually holds a single version, because collection trims a chain as soon as no reader needs its older entries.

Keys and values are byte slices of 24 bytes, of which up to 20 bytes are stored in place on 64-bit targets. Longer

Table 1: SurrealMX beside Hekaton and Silo. The first two columns are as their papers describe them.

	Hekaton	Silo	SurrealMX
Scope	An engine inside SQL Server, with T-SQL and compiled procedures	A research main-memory database serving one-shot requests	An embedded key-value engine with direct operations and transactions
Index	Latch-free hash index and Bw-tree	Masstree-inspired B+-tree	Concurrent adaptive radix tree; writers latch nodes, readers do not block
Versions	Several per record, each with begin and end timestamps	One per record and a transaction identifier; extra versions only for read-only snapshots	An immutable chain per key behind an atomic pointer, usually one version at rest
Isolation	Snapshot, repeatable read, serializable	Serializable, plus stale read-only snapshots	Snapshot isolation, or the default mode that also validates reads and scans
Commit check	Re-checks that the versions read are still visible at the end timestamp, and repeats scans for phantoms	Re-checks the identifiers of records read, and leaf version numbers for phantoms	Checks its read keys and scan ranges against the write sets of the commits in its window
Ordering	A global monotonically increasing timestamp counter	No central counter: identifiers chosen per worker, global epoch every 40 ms	Central atomic counters for ring slot and merge version; visibility in claim order
Reads	Lock-free	No shared-memory writes for records only read	No writes to tree or chain; only the thread’s own epoch record
Old versions	Cleaned cooperatively by worker threads	Epoch-based reclamation, per-core garbage lists	Trimmed by the writer, plus a per-thread tracked sweep
Durability	Logged and checkpointed; committed once logged	Results released when their epoch is durable	Optional; a commit is visible before it is logged; off by default

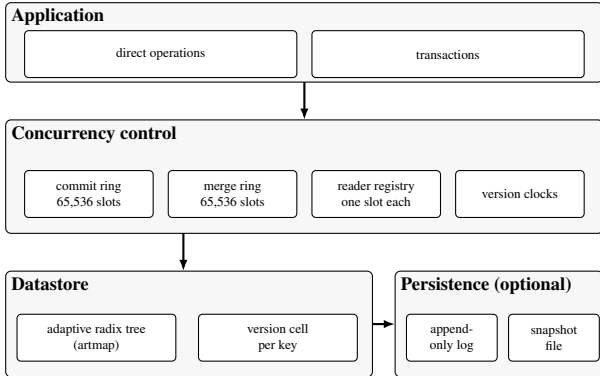


Figure 1: The parts of SurrealMX. The log is written after a commit has been applied to the tree, and a snapshot is written from the tree.

ones are allocated and reference counted. Two clocks order everything. A merge version is allocated with one atomic increment. A second counter, the published version, advances only in allocation order and says which merge versions are visible.

3.2 Reading

A read in a transaction costs an epoch pin, an atomic load of the cell’s pointer and a search of the cell’s chain for the newest version at or before the transaction’s snapshot. Short chains are searched linearly and longer ones by binary search. The read writes no shared memory in the tree or the chain. Its only writes are to the calling thread’s own epoch record. A read-only transaction tracks no reads and never aborts.

Commits that are visible but not yet applied to the tree live in the merge ring. A read first checks whether any such commit is newer than the last one retired, and if so probes the ring for the newest unretired write of the key. The cost of that probe grows with the number of unretired commits.

3.3 Direct operations

The database also offers operations that skip transaction setup: `get`, `exists` and `with_value` for reads, and `set`, `put` and `del` for writes. A direct read returns a value committed before the call began or while it ran and takes no snapshot, so two direct reads do not make a consistent view. `with_value` lends the stored bytes to a closure instead of copying them. A direct `set` or `del` is a one-key transaction that reads nothing and retries a write conflict internally. `put`, which fails if the key exists, does not retry.

Scans come in the same two forms. `scan_with` and `keys_for_each` lend each key and value to a closure, and the per-entry loop allocates nothing. A direct scan pins a snapshot in a slot owned by the calling thread. If that slot is already in use, or a visible commit has not yet been retired, the scan uses a pooled read transaction instead.

4 TRANSACTIONS

4.1 Snapshots and the reader registry

Every transaction owns a cache-padded slot in a registry for its whole life, including while it waits in the transaction pool. The slot holds two values, the published version and the commit watermark the transaction started from. Starting a transaction pins them in an order that a

Table 2: SurrealMX beside LMDB and RocksDB. The LMDB and RocksDB columns are as the LMDB paper [4] and Pebble’s notes [5] describe them, and the last row is how the harness ran them.

	LMDB	RocksDB	SurrealMX
Structure	Memory-mapped copy-on-write B+tree	A memtable and a write-ahead log over sorted tables in levels	Concurrent adaptive radix tree; an immutable version chain per key
Writers	One at a time, through a writer mutex	Concurrent commits grouped under a leader, which writes the group to the log when the log is on; the batches are applied concurrently	Concurrent; places claimed in two rings; versions published in claim order
Readers	A slot in the reader table, written at the start and end of a read transaction; nothing blocks a reader in normal operation	A sequence number gives the snapshot view	A snapshot is two counters in a slot its own transaction writes; a direct read takes no snapshot
Old data	Freed pages reused once older than every recorded reader snapshot	Removed as sorted tables are compacted	Trimmed by the writer that makes it, plus a tracked per-thread sweep
As run in memory	Files in <code>/dev/shm</code> , no flushing, writes through the memory map	RocksDB’s in-memory environment, write-ahead log disabled, optimistic transactions	No persistence

Table 3: The checks made at commit, in order.

Check	Applies to	On conflict
Write against write	every transaction that wrote	write conflict
Locked reads	reads made with <code>get_for_update</code> , in either mode	read conflict
Point reads	the default mode, transactions that wrote	read conflict
Scan ranges	the default mode, transactions that wrote	read conflict

concurrent scan of the registry can detect: the slot is first marked as pinning, a fence follows, and only then are the two counters read and stored. A scan that meets a slot in the pinning state treats the oldest active snapshot as unknown and skips reclamation for that pass. The oldest active snapshot is the minimum over all slots. It bounds both the collection of old versions and the release of committing entries. The registry holds up to about a million transactions at once.

4.2 Conflicts

A transaction runs under one of two modes. The mode that validates reads is the default: the engine calls it serializable snapshot isolation, and it adds read and scan validation to snapshot isolation. Snapshot isolation alone is available per transaction. The checks apply to a transaction that wrote; a transaction that only reads is never validated. At commit, a transaction that wrote is checked against every commit in its window, which is the commits claimed after its snapshot and before its own slot. Aborted commits are skipped, and so are commits already visible in its snapshot. Table 3 lists the checks.

The first check is decided exactly. A range-bound test and a small Bloom filter of 4,096 bits only skip work: a hit is always confirmed against the commit’s sorted key list. A scan records the range it asked for, not the keys it visited, so a limit does not narrow it. A rollback to a savepoint rewinds writes only, and reads made inside the

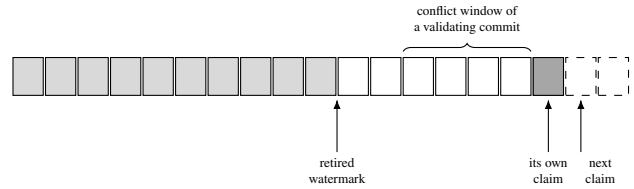


Figure 2: The commit ring. Slots below the retired watermark are released. A commit’s window runs from its snapshot to its own claim. Slots at and above the next claim are free.

rolled-back scope stay tracked, because a surviving write may depend on them. A transaction whose window has been retired past, or whose slot has been overwritten by a later lap of the ring, aborts with a write conflict, so a write transaction left open across more than 65,536 commits can meet this.

4.3 The two rings

Figure 2 shows the commit ring. Both rings have 65,536 slots indexed by sequence number. The commit ring holds, for each recent commit, its sorted key list, a Bloom filter when it has more than two keys, and a state: in flight, aborted, or the merge version at which it became visible. A claim is one fetch-add on a counter. Publishing a claim waits if the previous lap’s occupant is not yet published and complete, so a claim is bounded by the duration of one in-flight commit and never by a reader. A cleanup worker advances a retired watermark up to the oldest active snapshot and releases the entries below it. A validating transaction reads only the slots between its snapshot and its own claim.

The merge ring holds the whole write sets of commits that are visible but whose application to the tree has not been retired, indexed by merge version. Inserting waits if the slot’s previous occupant is not yet retired. A read probes it only when its snapshot is above the last retired

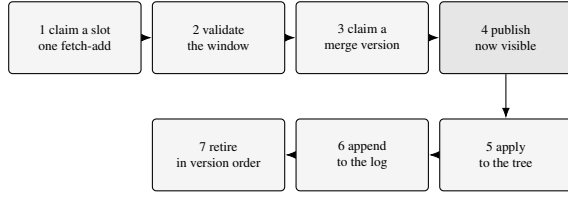


Figure 3: The commit path. The commit becomes visible at step 4. Steps 5 to 7 follow it. Step 6 happens only when persistence is on.

version, and a scan merges it lazily with the tree when an unretired write overlaps the range.

4.4 The commit path

Figure 3 shows the path of a commit that wrote.

A commit claims the next slot of the commit ring with one atomic increment and publishes its sorted key list and Bloom filter there. It validates against the commits in its window. It then claims a merge version and inserts its whole write set into the merge ring. It publishes its version. Validation is in the order of the commit ring’s slots. Publication is strictly in the order versions were claimed, so a commit that stalls delays the ones behind it, and from this point new snapshots see the commit through the merge ring. It applies each key to the tree, under the key’s lock, and trims the key’s older versions. If persistence is on, it appends to the log. Finally it marks its merge entry applied, and entries are retired from the ring in version order.

The two rings are what keep committers from queuing behind a single lock. Validation reads only the ring slots in its window, and visibility is an ordered counter. The cost of a commit is paid mostly by the writer: validation grows with the number of commits that ran during the transaction’s life, and application with the number of keys written. A transaction can also set stackable savepoints and later roll back to the latest or release it. Setting one records a position in an undo journal, every write after it records the key’s previous state, and a rollback replays the journal backwards.

4.5 Invariants

Four invariants carry the argument that these structures give snapshots that are consistent in version order, and safe reclamation. We state them as the code establishes them; Section 9 says how far they are tested.

Pin before read. A transaction publishes its snapshot to its slot in two steps. It marks both fields of the slot as pinning, executes a sequentially consistent fence, and only then reads the commit watermark and the merge clock and stores them. A concurrent scan of the registry therefore sees either the pinning sentinel or the final pair, never a stale pair that would let reclamation overtake a snapshot that has begun. A scan that meets the sentinel treats its bound as unknown and skips that pass.

Publish before validate. A commit enters the commit ring and is published there before it validates, and a validator waits for any slot in its window that is claimed but

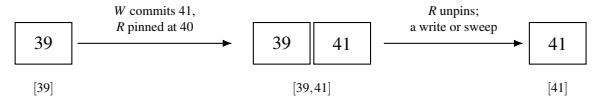


Figure 4: The chain of key k in the worked example. Version 39 is kept while R is pinned at 40 and trimmed after.

not yet published. Of two overlapping commits, the later therefore sees the earlier.

Visibility is a prefix of claim order. Merge versions become visible through a counter that advances strictly in the order they were claimed, and a commit does not return until its own version is covered. A snapshot therefore never sees a commit while missing one whose version was claimed before it.

A bound that excludes only its owner. The inline collector bounds its trimming by the minimum over every registered slot except the committer’s own. Excluding any other slot is forbidden, because any other slot may hold a reader of the very versions being trimmed.

4.6 A worked example

Consider a reader R and a writer W on a key k whose chain holds a single version, 39 (Figure 4). R begins and pins published version 40. W writes k and commits: it claims a commit slot, finds no conflicting commit in its window, claims merge version 41, inserts its write set into the merge ring and publishes 41, so new snapshots now see it.

R reads k . The merge ring holds a commit newer than R ’s snapshot, which R ignores, and the tree’s chain is [39], so R reads 39. W then applies k under the cell’s lock: it copies the chain and appends 41. The oldest active snapshot is R ’s 40, so version 39, the newest visible to it, is kept, and the chain becomes [39, 41]. R reads k again and still gets 39. When R finishes and unpins its slot, the next commit that writes k , or the tracked sweep, finds no snapshot older than 41 and trims 39, and the key returns to a single version.

5 GARBAGE COLLECTION AND MEMORY

Old versions are removed in three ways.

Inline. A commit that writes a key trims that key’s chain while it holds the key’s lock. It keeps the newest version visible to the oldest active snapshot and everything newer. If that newest visible entry is a delete marker it removes the marker too, and a key left empty is unlinked from the tree. The committer’s own slot is excluded from the bound; no other slot may be.

Tracked sweep. A key whose chain could not be fully trimmed is recorded in a buffer owned by the committing thread. A background worker drains the buffers every ten seconds by default and trims those keys. Its cost scales with the amount of pinned garbage, not the size of the dataset.

Table 4: Memory structures, derived from the types and constants.

Structure	Size or count
Key and value	A 24-byte slice each; up to 20 bytes stored in place, longer ones allocated and reference counted
Version cell	An 8-byte lock and an 8-byte pointer per key
Version value	A heap value per key: 16 bytes of counts and an enum stated to take 40 bytes
Commit ring	65,536 slots of about 24 bytes, about 1.5 MiB
Merge ring	65,536 slots of about 16 bytes, about 1 MiB
Registry	1,024 chunk pointers, and a chunk of 1,024 cache-padded slots on first use
Pooled transaction	Two Bloom filters, one of 512 bytes for reads and one for locked reads; the pool holds 512 by default

Cleanup. A second worker releases commit-ring entries up to the oldest active snapshot, every second by default.

Both workers can be disabled, and `run_cleanup`, `run_gc_tracked` and `run_gc` do the same work on demand. The workers do not run on WebAssembly targets, where the application calls them. The oldest active snapshot bounds all three. A transaction or a scan that stays open keeps every key written in the meantime tracked and stops the commit ring releasing entries. Three options bound memory, and none is set by default. A memory limit refuses a commit that would take the estimated bytes of live keys and values past it, with an error that leaves the transaction open so that it can retry or cancel; the estimate counts each version’s key and value and not the engine’s own structures. A maximum transaction age makes the transaction’s next operation fail and releases its reader slot. Limits on the depth and the bytes of the asynchronous log’s queue refuse a commit before it is applied (Section 8). A statistics call reports the counters behind them: live keys and versions, estimated bytes, tracked garbage candidates, the oldest pinned version, active readers, both rings’ occupancy and the queue’s depth.

What is allocated. Table 4 lists what the structures cost, as the types and constants give it. These are derived from the code and have not been measured.

6 A COST MODEL

Table 5 states what each operation’s cost grows with. It describes the loops in the code, not measurements, and it is the basis for the predictions in Section 7.

7 WHERE THE PERFORMANCE COMES FROM

Table 6 sets each design decision against what it spares and what it costs. These are properties of the design as read from the code. We have not run ablations that remove one decision at a time, so the measurement in Section 10 does not apportion the speed-up among them.

8 PERSISTENCE

SurrealMX holds no persistent state unless the application asks for it. With persistence on there are three independent settings: the log mode (never, synchronous on

Table 5: What each operation’s cost grows with. k is the key length, c the length of a key’s chain, u the number of unretired commits, w the number of commits in a window, n the keys written and r the registered transactions.

Operation	Cost grows with
Read in a transaction	A load of the stopped flag; k for the tree lookup; c for the chain search, linear up to four and binary above; u for the merge-ring probe
Direct read	As a read, without the snapshot and its slot
Scan	The entries returned; a lazy merge when an unretired write overlaps the range
Start a transaction	A constant: a few atomic stores and one fence
Commit: validate	w range tests and filter probes, and a binary search of the commit’s keys on each hit
Commit: apply	n tree insertions, each copying a chain of length c , and a trim of each chain
Commit: account	Two relaxed atomic adds to the thread’s shard of 32 cache-padded counters; with a memory limit set, a sum over the 32 shards first
Oldest snapshot	r , including pooled transactions

commit, or asynchronous after commit), the fsync mode (never, after every append, or at most once per interval) and snapshots (never, or at a fixed interval, with optional LZ4 compression). Every default is off. Table 7 gives what each common configuration survives.

A commit is applied to the tree and made visible to other readers before it is appended to the log, so a reader can see a commit that a crash then loses. If the append fails, the commit returns an error that says it was not persisted, it stays applied and visible in memory, and the database is stopped: every later operation, reads included, fails with an error that gives the reason. A panic while a commit is being applied, or in the asynchronous log’s thread, stops the database in the same way.

Log write path. Under a synchronous log with fsync after every append, committers share a group-commit leader: the committers enqueue, one becomes the flusher, drains the queue, appends every frame, issues one fsync and wakes the others. Under a synchronous log with other fsync modes, each commit takes the file’s lock and writes its own frame. Under an asynchronous log, the commit hands its write set to a queue and returns, and a background thread drains the queue in batches of up to 100 frames. The queue is unbounded unless a depth or a byte limit is set; a commit that would pass a limit is refused before it is applied, with an error that leaves the transaction open, and the refusal does not stop the database. An error in the background thread stops the database, but the commits that thread had not yet written had already returned success.

Formats. Table 8 gives the two files. The log holds one frame per transaction, and a frame is applied at recovery only after the whole frame has been read and verified, so a transaction is either present or absent.

Recovery. When the database opens, an incomplete frame at the end of the log is cut off before any new commit is appended. A damaged frame that is followed by an

Table 6: The design decisions that determine throughput, and their prices.

Decision	What it spares	What it costs
Immutable version chains behind an atomic pointer	A reader never waits for a writer and never writes tree memory, so reads scale with cores.	Every write copies the key’s chain, and replaced chains are freed only after readers move on, which raises peak memory.
A snapshot is two counters in the transaction’s own slot	No shared reader count to update: starting and ending a transaction touches one cache line the transaction owns.	Finding the oldest snapshot scans every registered slot, pooled ones included.
Direct operations	No transaction checkout and no snapshot, no copy of the value with <code>with_value</code> , no allocation per scanned entry.	A direct read has no snapshot, so it cannot give a consistent view across keys.
Two rings instead of a commit lock	A commit claims its place with one atomic increment, validates against the slots in its window only, and becomes visible through an ordered counter.	Publication is in claim order, so a stalled commit delays those behind it, and a window past 65,536 commits aborts.
A cheap test before the exact one	A range-bound test and a 4,096-bit Bloom filter rule out most commits that do not conflict without comparing keys.	A hit costs a binary search of the commit’s keys.
Collection at write time	The writer that makes garbage trims it under a lock it already holds and records only what it could not trim, so no pass scans the dataset.	An open transaction pins every key written while it is open.

Table 7: What each configuration survives. A process crash leaves the operating system’s buffers intact. A system crash or power loss does not.

Configuration	Process crash	System crash
Snapshots only	loses data since the last snapshot	loses data since the last snapshot
Asynchronous log	queued commits may be lost	queued commits and unflushed buffers may be lost
Synchronous log, no fsync	survives	buffers not yet flushed by the system may be lost
Synchronous log, fsync on every append	survives	survives

intact one fails the open with a corruption error rather than dropping what follows, though the check looks only at the next frame. Recovery loads the snapshot, replays the log on top of it, seeds the clocks from the largest version seen, and collapses the replayed chains with one full collection.

Snapshots. One snapshot runs at a time. It reads the log’s length, pins a reader slot as a transaction does, so that the versions it reads stay protected from collection, and writes the newest version at or below the pinned version of every key to a uniquely named temporary file. It syncs that file, renames it into place and syncs the directory. It then replaces the log: the part after the length it read first is copied behind a fresh header into a new file beside the log, which is synced and renamed over it, so a crash leaves either the old log or the new one. The cutoff is read before the version is pinned, so that every frame before it is at or below the pinned version.

9 VERIFICATION

The engine is checked by six kinds of test.

Table 8: The persistent files.

File	Layout
Log	An 8-byte header (a marker byte, the letters SMXAOL, a version byte), then frames. A frame is a 4-byte little-endian payload length, a 4-byte checksum (the low 32 bits of xxh3 seeded with the length) and the payload: the key, version and optional value of every write of one transaction. A log written before frames existed is still read, and the header is added after its last record.
Snapshot	A stream of each key with its newest committed version at or below the version the snapshot pinned. A key whose entry at that version is a delete marker is omitted. It is optionally compressed with LZ4, detected from the stream’s magic number.

Deterministic simulation. A seeded generator drives the engine and a sequential reference model with the same actions: transactions in both modes, reads, scans, save-points, direct operations and maintenance. Every read, scan result and commit verdict must match, or the run fails. The continuous integration job runs 5,000 seeds of 5,000 steps, 25 million steps in all, in memory over a pool of 30 keys with up to 12 transactions open at once. Two further runs with fixed seeds use the append-only log, and restart the engine from it. Each run is reproducible from its seed: the generator is seeded, one thread drives the engine, and the background workers are disabled so that collection and cleanup run when the driver asks. This is deterministic simulation testing in the manner of FoundationDB [13], applied to the interleavings of open transactions. It does not simulate thread scheduling, so it does not exercise races between threads, and the runs that use the log write real files. The model repeats the engine’s own conflict rules, so it checks the optimised implementation against a simple one of the same algorithm and not against an independent definition of serializability.

Concurrency models. Five models of the commit ring run under the `loom` permutation checker with a bound on pre-emptions. They check that a read never sees another lap’s entry and that the completed prefix never covers an entry in flight. The merge ring, the registry, the epoch cell and the version chains are not modelled, so the pin-before-read and visibility invariants of Section 4.5 rest on the simulation and the stress tests, not on a model.

Isolation anomaly tests. A suite of hand-built schedules, named for the isolation phenomena of the literature [1], runs against the default mode. It covers dirty writes, aborted and intermediate reads, circular information flow, lost update, read skew, and write skew over keys and over scanned ranges, and each schedule must commit or abort as the mode prescribes. Further tests exercise each conflict class on its own: write against write, read against write, a deleted key, a scanned range and a locked read. Snapshot isolation does not track reads, and a test checks that it permits the read-write anomaly that the default mode prevents.

Recovery tests. A child process is killed while committing and the log is checked for whole transactions in order, and the log is cut at every byte offset for six mode combinations. These tests kill the process, not the machine, so power loss with unflushed data is not simulated.

Isolation, snapshot and bound tests. Three small suites in the repository run in a fraction of a second. One runs eight threads of bank transfers while a read-only transaction audits that the balances still sum to their total, and checks write skew and concurrent increments. One takes snapshots while writes continue, and checks that each truncates the log and that replaying the tail restores the state. One checks the memory limit, the transaction age, the queue limits, the statistics and the stopped state. They pass, and each is small.

Memory safety. The library tests and the direct-operation tests run under Miri on a nightly toolchain, and the crate denies undocumented unsafe code. The only unsafe code in the crate is the epoch cell.

These tests show that the engine handles the standard schedules and that its verdicts equal its model’s. They do not prove serializability.

10 EVALUATION

10.1 Method

The measurements use `crud-bench` on one bare-metal machine: an AMD Ryzen Threadripper 9970X (32 cores, 64 threads, up to 5.48 GHz) with 128 GB of DDR5 memory. They load 15 million keys across 48 worker threads and 128 clients and run 5,000,000 iterations of each range scan. Each figure is the best of three runs of that benchmark, taken separately for each benchmark and each engine, so that the figures for one engine come from different runs. The comparators are LMDB and RocksDB, each in memory and described in Section 2. Hekaton and

Table 9: What each measured operation calls in the harness’s SurrealMX adapter.

Operation	Call	Includes
Read	<code>with_value</code>	A linearizable borrowed read: no snapshot, no copy
Create, update	<code>set</code>	A one-key transaction that reads nothing: pool check-out, commit ring, merge ring, apply
Delete	<code>del</code>	As a write, for a delete marker
Scan	<code>scan_with,</code> <code>keys_for_each,</code> <code>total</code>	A range of up to 100 rows, borrowed, no allocation per entry

Table 10: Operations per second, 15 million keys, 48 worker threads, with LMDB and RocksDB each in memory. Scan is `select(*) limit(100)`. Ratios are computed from the figures shown.

Operation	SurrealMX	LMDB	RocksDB	vs LMDB	vs RocksDB
Point read	21,506,711	1,226,897	759,060	17.5×	28.3×
Create	1,887,149	232,805	616,649	8.1×	3.1×
Update	1,595,009	241,431	572,246	6.6×	2.8×
Delete	1,814,037	177,198	705,773	10.2×	2.6×
Scan	18,031,692	1,188,365	836,666	15.2×	21.6×

Silo are compared by design only: we have not run them, and their papers report other workloads on other hardware.

The machine runs Ubuntu 24.04 with kernel 6.8.0-146, the `powersave` frequency governor, transparent huge pages in `madvise` mode and an 8 GB swap file. The harness uses the `mimalloc` allocator. The figures are for SurrealMX 1.0.0. LMDB is `heed 0.22.1` over `lmdb-master-sys 0.2.6`, and RocksDB is the `surrealdb-rocksdb` crate 0.24.0-surreal.5, which wraps RocksDB 11.0.0. The harness is `crud-bench`, and the adapters for the three engines are those described here.

10.2 What each operation measures

The numbers are for the direct operations, not for default transactions. Table 9 says what each harness operation calls and what that includes. The harness runs LMDB from a memory-backed file system with flushing off and a writable memory map, and RocksDB over its in-memory environment with the write-ahead log disabled and optimistic transactions. SurrealMX runs without persistence. All three therefore measure an engine that pays nothing for durability.

These paths take no read tracking and, for reads, no snapshot, so a transaction that reads and validates would cost more.

10.3 Throughput

The read and scan advantage is what Section 7 predicts: no lock, no shared write, no per-read snapshot. The write advantage over LMDB is consistent with LMDB admitting one writer at a time, which the harness’s 128 clients queue behind. Against RocksDB it is smaller, because

Table 11: Scans per second over the same data. Limit returns the first 100 rows, skip and limit walks and discards 5,000 first, keys returns the keys alone, and count visits every key, 1,000 times.

Scan	SurrealMX	LMDB	RocksDB
Limit	18,031,692	1,188,365	836,666
Keys, limit	15,524,603	1,192,034	837,377
Skip and limit	232,121	552,885	47,671
Keys, skip and limit	253,039	554,451	47,816
Count	17.3	24.2	5.0

Table 12: Resting and peak memory over the whole run, 15 million keys.

Engine	Resting	Peak
SurrealMX	15.9 GiB	32.0 GiB
LMDB (in memory)	19.8 GiB	20.1 GiB
RocksDB (in memory)	26.3 GiB	96.1 GiB

RocksDB’s in-memory writes, with its log disabled, cost less than its reads.

On the full count SurrealMX is slower than LMDB, at 17.3 scans per second against 24.2, and so is RocksDB, at 5.0. We have not examined why. On scans that skip 5,000 rows SurrealMX is also slower than LMDB. We have not examined why either.

10.4 Memory

SurrealMX has the lowest resting footprint of the three: 15.9 GiB for 15 million documents, about 1.1 KB each. Its peak is 32.0 GiB, which is higher than LMDB’s 20.1 GiB. The measurement does not say why. The design offers candidates: the deferred freeing of replaced chains, the copy made on every write, merge-ring entries that hold whole write sets until they are retired, and the queue of the asynchronous log. We have not measured which dominates.

10.5 Threats to validity

Internal. Each figure is the best of three runs, so it is higher than a typical run, and the spread between runs is not shown.

Construct. The figures measure the direct operations. A transaction that reads, validates and writes costs more, and the default mode adds read and scan tracking. Persistence was off, so the numbers say nothing about the cost of the log or of snapshots. LMDB ran with flushing off, and its single writer is a design property that this write-heavy workload exposes.

External. The figures come from one machine, one key count and generated values, and are best read as ratios, since results on other hardware will differ. They say nothing about mixed workloads with long-running transactions, where the pinned snapshot bounds collection.

10.6 What the measurement does not cover

Five measurements would test claims this paper makes from the structure of the code and have not been run: throughput as cores are added, which the design predicts is near-linear for reads; the cost of the default validating mode against plain snapshot isolation and the direct operations; the effect of one long-lived reader on memory and commit latency; the cost of each persistence mode; and ablations that remove one decision of Table 6 at a time. A latency distribution under load is also missing.

11 DISCUSSION

Where it fits. SurrealMX suits working sets that fit in memory, read-heavy or scan-heavy workloads, short transactions and embedded or in-process use, including the browser. Its design spends memory and write-side work to keep readers free: every write copies a chain, and a snapshot’s lifetime bounds collection.

Where it does not. It is not a fit for data larger than memory, for workloads that need durability before visibility without paying for a log on every commit, for read-only transactions that must be serializable with updates, for long-running transactions or scans that would pin the collector, or for a memory budget that must be enforced to the byte, since the limit counts keys and values and not the engine’s own structures, and is off by default.

Trade-offs. Two decisions deserve a second look. The central counters keep commit ordering simple and cheap per commit but are the shared writes that Silo avoided, and strict claim-order publication means a stalled commit delays the others. Copy-on-write chains keep readers lock-free but raise peak memory. Both are priced in Table 6.

Open questions. a proof of the default mode’s serializability for update transactions; loom models for the merge ring and registry; an option to publish only after the log; and the ablations above.

12 LIMITATIONS

- **Memory is unbounded by default.** The memory limit, the transaction age and the queue limits are all off unless set. The limit counts the bytes of keys and values and not the engine’s structures, so the process can exceed it. A long-lived transaction, a long scan or a thread that holds an epoch pin delays the freeing of old versions whichever options are set, and a transaction that is not used again is not timed out until it is.
- **Visibility precedes durability.** A commit is visible before it is in the log, and a failed append leaves it visible and stops the database.
- **Commits are ordered.** Versions are published in the order they were claimed, so a stalled commit delays those behind it. The rings hold 65,536 commits.
- **Read-only transactions are not validated.** The default mode validates transactions that write, in commit order. A transaction that only reads runs under

Table 13: Defaults and constants.

Setting	Value
Transaction pool size	512
Reset threshold	100
Collection interval	10 s, enabled
Cleanup interval	1 s, enabled
Commit and merge ring capacity	65,536 slots each
Registry	1,024 chunks of 1,024 slots
Commit Bloom filter	4,096 bits, 3 hashes, built when a commit has more than two keys
Asynchronous log batch	100 frames
Snapshot compression	LZ4 at level 7, off by default
Log mode, snapshot mode, fsync mode	never, never, never
Memory limit, maximum transaction age	none, none
Asynchronous queue depth and bytes	unlimited

snapshot isolation, and so does any transaction that asks for the optional mode, which tracks no reads.

- **No history.** There are no historical reads. A snapshot is the latest committed state.
- **Platforms.** On WebAssembly there is no persistence, no background workers and no LZ4.

13 CONCLUSION

SurrealMX takes shared writes off its hot paths: reads, snapshots, commit ordering and collection. Against Hekaton and Silo it chooses a point between them, keeping Hekaton’s multi-version reads and a global counter while validating against concurrent commits as Silo validates against records. The measurement shows large read and scan advantages over in-memory LMDB and RocksDB on one machine, with a resting footprint below both and a peak above LMDB’s. The limits above are the ones to weigh before relying on it: memory that is unbounded unless limits are set, visibility ahead of durability, and read-only transactions that run under snapshot isolation and are never validated.

A REPRODUCING THE FIGURES

The harness is `crud-bench`, which is open source. Each engine was run in turn through the harness’s `run.sh`, which waits for the system’s load to fall below 0.5 before it starts:

```
CRUD_BENCH_CONFIG=config/memory.toml \
./run.sh -d surrealmx --no-build \
--optimised -s 15000000 -c 128 -t 48
```

The comparators were run with `-d lmdb -e memory` and `-d rocksdb -e memory`. The engine settings that affect the figures are SurrealMX’s defaults, listed in Appendix B.

B CONFIGURATION REFERENCE

Table 13 lists the engine’s defaults and constants, as the code states them.

Three presets adjust these: a high-performance preset doubles the pool and the reset threshold and sets collection to 30 s and cleanup to 100 ms; a low-resource preset halves

the pool and the threshold and sets 60 s and 500 ms; and a reduced-memory preset halves them and sets 5 s and 100 ms.

REFERENCES

- [1] Atul Adya, Barbara Liskov, and Patrick O’Neil. Generalized isolation level definitions. In *Proceedings of the 16th International Conference on Data Engineering (ICDE 2000)*. IEEE, 2000.
- [2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O’Neil, and Patrick O’Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD ’95)*, pages 1–10. ACM, June 1995. doi: 10.1145/223784.223785. URL <https://doi.org/10.1145/223784.223785>.
- [3] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD ’08)*, pages 729–738. ACM, June 2008. doi: 10.1145/1376616.1376690. URL <https://doi.org/10.1145/1376616.1376690>.
- [4] Howard Chu. MDB: A memory-mapped database and backend for OpenLDAP. Paper presented at the 3rd International Conference on LDAP, October 2011, October 2011. URL <https://www.openldap.org/pub/hyc/mdb-paper.pdf>.
- [5] cockroachdb/pebble contributors. Pebble vs RocksDB: Implementation differences. Pebble repository, docs/rocksdb.md, section “Commit Pipeline”, 2019. URL <https://github.com/cockroachdb/pebble/blob/master/docs/rocksdb.md>.
- [6] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server’s memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD ’13)*, pages 1243–1254. ACM, June 2013. doi: 10.1145/2463676.2463710. URL <https://doi.org/10.1145/2463676.2463710>.
- [7] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49. USENIX Association, February 2021. URL <https://www.usenix.org/conference/fast21/presentation/dong>.
- [8] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12 (3):463–492, July 1990. doi: 10.1145/78969.78972. URL <https://doi.org/10.1145/78969.78972>.
- [9] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, April 2013. doi: 10.1109/ICDE.2013.6544812. URL <https://doi.org/10.1109/ICDE.2013.6544812>.

- [10] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN 2016)*, pages 1–8. ACM, June 2016. doi: 10.1145/2933349.2933352. URL <https://doi.org/10.1145/2933349.2933352>.
- [11] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 18–32. ACM, November 2013. doi: 10.1145/2517349.2522713. URL <https://doi.org/10.1145/2517349.2522713>.
- [12] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, March 2017. doi: 10.14778/3067421.3067427. URL <https://doi.org/10.14778/3067421.3067427>.
- [13] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. FoundationDB: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*, pages 2653–2666. ACM, June 2021. doi: 10.1145/3448016.3457559. URL <https://doi.org/10.1145/3448016.3457559>. Deterministic simulation testing: Section 4, “Simulation Testing”.