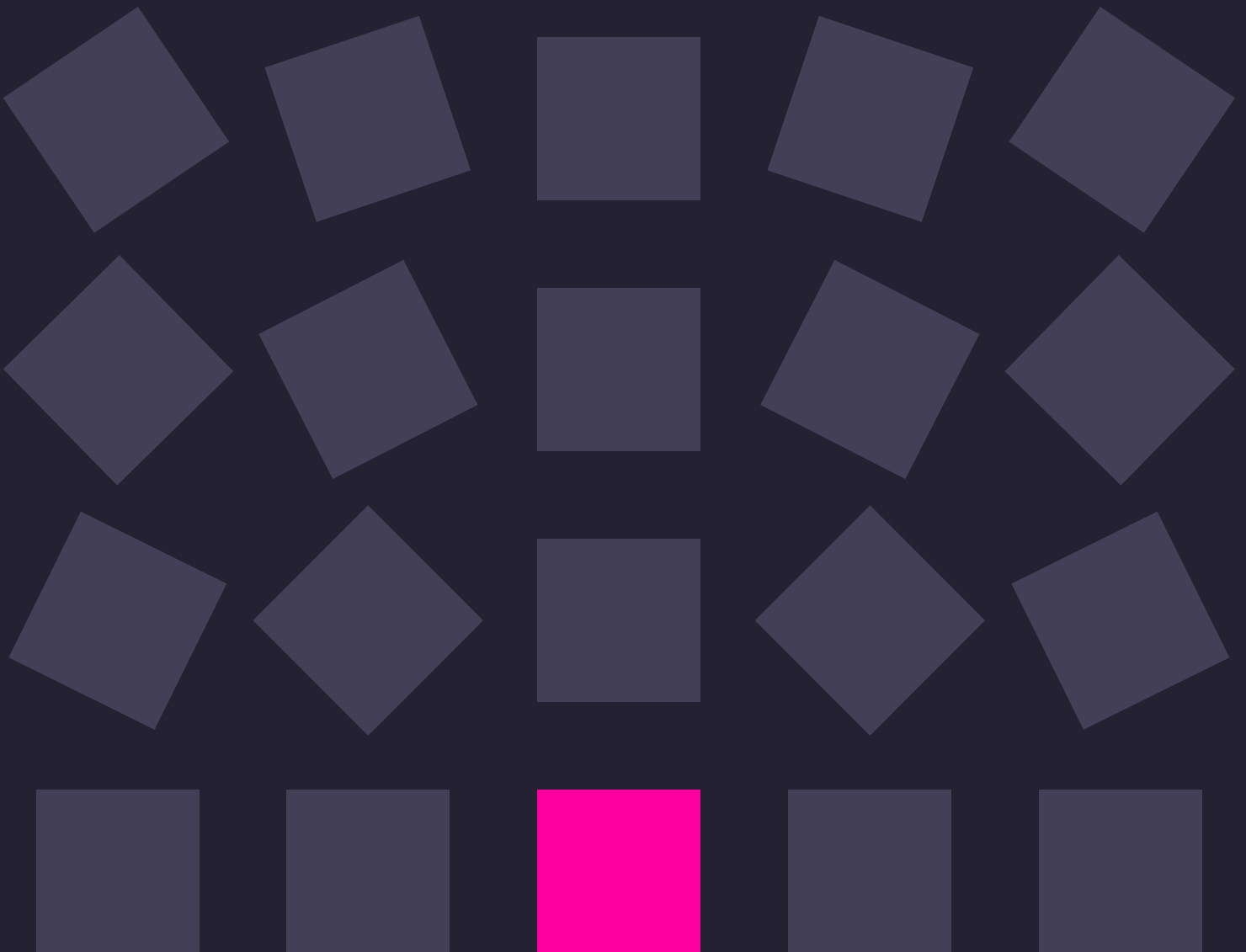


SurrealKV

A versioned LSM engine with
radix-tree memtables and a commit ring



SurrealKV: A Versioned LSM Engine with Radix-Tree Memtables and a Commit Ring

Farhan Khan Tobie Morgan Hitchcock Stu Schwartz

SurrealDB

October 2026

Describes SurrealKV 1.0.0

Abstract

SurrealKV is an embedded key-value storage engine written in Rust, built on a leveled log-structured merge-tree with snapshot-isolated transactions. With versioning enabled, it keeps the history of every key, for a retention period you set or for ever, so data can be read as it stood at a given time. Three structures set it apart from the engines in its family, and this paper explains how each contributes to its throughput. Its memtable is a concurrent adaptive radix tree in a fixed arena, where RocksDB and Pebble use skiplists. Its commits pass through a ring buffer in which many committers validate concurrently, and a single flusher task turns what has accumulated into one group: it numbers the group, appends it to the write-ahead log with at most one fsync, applies it to the memtable and publishes it in order. And its versioning needs no index of its own: a write carries its time in its value, versions stay ordered by sequence number, and compaction keeps history by the rule it already uses for snapshots. A read pins nothing but a sequence number. Tables carry a partitioned index and a partitioned Bloom filter behind one sharded cache, and large values can be kept in a separate log. On one 32-core machine with 500,000 keys, measured against RocksDB, SlateDB, Fjall, ReDB, LMDB and libmdbx, SurrealKV's point reads are 8.3 times faster than RocksDB's and 9.0 times faster than ReDB's. With synchronous durability its creates are 26.8 times faster than RocksDB's and its updates 6.1 times; without it RocksDB's writes are faster than SurrealKV's, by 1.9 to 2.4 times. Its scans are slower than LMDB's. These figures are for unversioned trees: the cost of versioning has not been measured. We set out each design decision, what it spares and what it costs, state the invariants the design relies on, set its commit pipeline, memtable, compaction and recovery against those of LevelDB, RocksDB and Pebble, say how the engine is tested, and say where it falls short.

1 INTRODUCTION

A key-value storage engine sits under everything a database does, so its throughput, its failure behaviour and its cost to operate are inherited by every query. SurrealKV is an embedded, transactional, log-structured merge-tree in Rust, written as SurrealDB's own alternative to RocksDB, a large C++ library that SurrealDB does not control. That is the whole of the case for building it. This paper is about the engine: the three structures that make it unusual, and what they cost.

Thesis. SurrealKV's throughput is the product of a small number of design decisions, each of which removes shared work or shared waiting from a hot path. Many committers validate concurrently and one task then orders them, so a burst of commits costs one log append and at most one fsync. The memtable is a concurrent adaptive radix tree allocated from a fixed arena, where RocksDB and Pebble use a skiplist [4]. A read pins a sequence number and nothing else. Tables are cheap to probe. Versioning, which is opt-in, rides on the same structures: a time is a stamp in the value, versions are ordered by the sequence numbers that order everything else, and compaction decides what history to keep with the rule it uses for snapshots. We describe each decision, say what it spares and what it costs, and state where the evidence for it is a measurement and where it is only the structure of the code.

Contributions. This paper makes six contributions.

- An account of the commit pipeline: concurrent validation against a ring, a single flusher that groups, logs, applies and publishes, and the two durability modes built on it, and a comparison with the pipelines of RocksDB and Pebble (Sections 2 and 4).
- An account of the structures that serve reads and writes: the radix-tree memtable in its arena, the log, the read path, the table format, the cache, leveled compaction and the optional value log, and where each departs from LevelDB, RocksDB and Pebble (Sections 5, 6, 7 and 9).
- An account of versioned trees: how a write is stamped with its time, how `get_at` and `history` read, and how compaction keeps history for a retention period (Section 8).
- The invariants the design relies on, a cost model, and a table that sets each throughput-determining decision against what it spares and what it costs (Sections 4.4, 10 and 11).
- An account of recovery and integrity, and of how the engine is tested (Sections 12 and 13).
- A measurement against six other engines, RocksDB, SlateDB, Fjall, ReDB, LMDB and libmdbx, that says what it does and does not show, a discussion of where the engine fits, and its limits (Sections 14, 15 and 16).

Scope. This paper describes SurrealKV 1.0.0. We state what the code does. A tree is unversioned unless it is opened versioned, and the measurements are of unversioned trees. We do not claim novelty for the underlying

techniques, which 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

Log-structured merge-trees. A log-structured merge-tree (LSM-tree) [10] absorbs writes in memory and writes them to disk as sorted, immutable runs that are merged in the background. Bigtable [2] described the arrangement of an in-memory memtable, a write-ahead log and immutable sorted tables, and LevelDB’s implementation notes [13] describe leveled compaction, in which each level is a sorted run several times larger than the one before. RocksDB [5] extends that design for production use. Pebble [4], Fjall [6] and SlateDB [12] are further members of the family. The price of the arrangement is read amplification: a key may sit in the memtable and in several tables, so a read searches them from newest to oldest, and filters keep it from opening tables that cannot hold the key.

The engines in this paper. Table 1 lists the eight engines we set SurrealKV beside and what each is used for. LevelDB, RocksDB and Pebble are design references, and Section 9 compares SurrealKV’s design with theirs. RocksDB, SlateDB, Fjall, ReDB, LMDB and libmdbx are the engines we measure against in Section 14. The last three of them are not log-structured. LMDB [3] serves reads from a memory-mapped copy-on-write B+tree and admits one writer at a time; libmdbx [8] is a descendant of it that keeps that rule; and redb [11] stores its data in copy-on-write B+trees too, and lets readers and a writer proceed without blocking one another. A scan in such an engine walks one tree, where an LSM-tree merges several sorted runs, and Section 14 shows the engines that lead SurrealKV on scans to be these.

Commit pipelines. Each write is given a sequence number, and a read at a sequence number sees the newest version at or below it. A commit pipeline therefore has to write batches to the log, and make them visible, in sequence order. Pebble’s design notes [4] describe the traditional RocksDB pipeline, in which concurrent commits form a batch group whose leader writes the whole group to the log, after which the group’s batches are applied to the memtable concurrently and the visible sequence number is bumped once. They criticise it for copying batch contents into the group and for its thread synchronisation points, and describe Pebble’s alternative: a commit mutex held only while a batch is given its sequence number and written to the log, a concurrent memtable apply, and an ordered publication through a commit queue.

Table 2 sets SurrealKV’s pipeline beside those two. SurrealKV keeps the group, so one append serves a burst of commits, but the leader is a standing task and not an elected thread, and the group is applied to the memtable by that task alone. It still copies each batch, into a scratch buffer, and it replaces the leader-and-follower waits with one hand-off per commit. What it gives up is a concurrent memtable apply. Section 4 gives the details and Table 7 prices the choice.

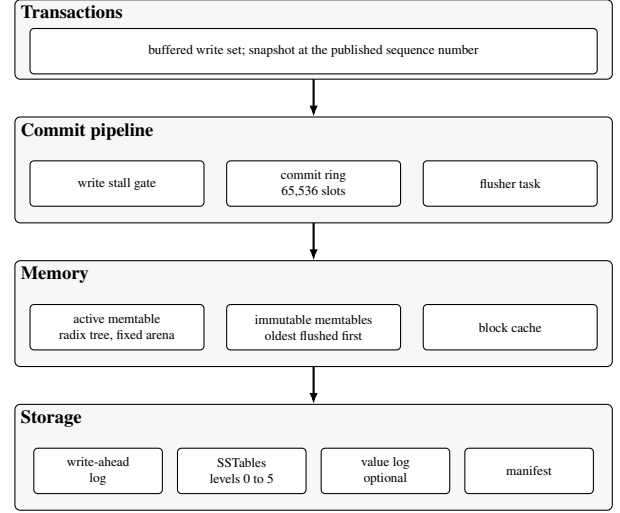


Figure 1: The parts of SurrealKV. One task flushes memtables to level 0 and one task compacts levels.

Value separation and isolation. WiscKey [9] observed that compaction rewrites large values again and again, and keeps values in a separate log so that only keys and small pointers move. SurrealKV offers this as an option. Snapshot isolation [1] lets each transaction read from the snapshot taken when it began and aborts a transaction whose writes collide with a concurrent commit.

Testing. FoundationDB [14] popularised deterministic simulation testing, in which time, scheduling and faults are all under the test’s control. Section 13 says how far SurrealKV’s tests go toward that and where they stop.

3 ARCHITECTURE

Figure 1 shows the parts. A transaction buffers its writes and reads from a snapshot. A commit passes a stall gate, takes a place in the commit ring and is handed to the single flusher task, which appends it to the write-ahead log and applies it to the active memtable. A full memtable joins a queue of immutable memtables, and one background task flushes them oldest first to level 0 of the table store. A second background task compacts levels. A manifest records which tables belong to which level.

The engine’s defaults are leveled compaction with six levels, a 100 MiB memtable, 64 KiB blocks and a 32 MiB block cache. A tree is opened and configured with a builder, and a transaction chooses its own durability.

4 TRANSACTIONS AND THE COMMIT PIPELINE

4.1 Snapshots and conflicts

Every key operation receives a sequence number, a 56-bit counter that the flusher assigns in commit order. A transaction’s snapshot is the published sequence number when it began, and a read sees the newest version of a key at or below it. A transaction that can read registers its snapshot in a tracker of 64 sharded sets, which compaction consults to know which versions must survive. A write-only transaction cannot read and takes no snapshot.

Table 1: The engines this paper compares SurrealKV with. The second column is what each engine’s own documentation or paper says; the third is how this paper uses it.

Engine	Structure and properties it documents	Used here for
LevelDB [13]	LSM-tree: a log, a memtable, sorted tables in levels and a manifest that is itself a log; level 0 is compacted at four files	Design reference
RocksDB [5]	LSM-tree that extends LevelDB’s design for production use; groups concurrent commits into one batch group written to the log by its leader	Design reference and measured
Pebble [4]	LSM-tree that inherits RocksDB’s file formats; a commit mutex held only while a batch is numbered and logged, then a concurrent apply	Design reference
SlateDB [12]	LSM-tree whose files all live in object storage; a single writer enforced through a fenced manifest, and many readers	Measured, on local disk and not object storage
Fjall [6]	LSM-tree in safe Rust, one tree per key-space; optional serializable transactions and optional key-value separation	Measured
ReDB [11]	Pure-Rust embedded store of copy-on-write B+trees, loosely inspired by LMDB; MVCC with concurrent readers and a writer, without blocking	Measured
LMDB [3]	Memory-mapped copy-on-write B+tree; one writer at a time; readers take a slot in a reader table	Measured
libmdbx [8]	A deeply revised descendant of LMDB, forked in 2015; copy-on-write and MVCC; no more than one write transaction at a time	Measured

Table 2: The commit pipelines of RocksDB and Pebble, as Pebble’s design notes describe them, and of SurrealKV.

	RocksDB	Pebble	SurrealKV
Who orders commits	The leader of a batch group	A commit mutex held while a batch is numbered and logged	One flusher task, in ring order after validation
Log write	The leader writes the whole group	Each batch is written on its own, a copy into the log’s buffer	The whole group in one append
Memtable apply	Concurrent, by the batches of the group	Concurrent, outside the mutex	Sequential, by the flusher
Visibility	The visible sequence is bumped once for the group	Published in order through a commit queue	Published once per group, in ring order
Waits	Leader and followers wait on one another	A queue entry per commit	One hand-off to the flusher and one completion per commit
Sync	Optional	Optional	One fsync per group if any member asked

Isolation is snapshot isolation with first-committer-wins on written keys. A transaction can also name keys it has read with `get_for_update`, and those join the check. This is not a serializable mode: plain reads and scans are not validated, so write skew across different keys is possible, and a range delete is checked only on its start key. A commit is terminal. Whether it succeeds or fails, the transaction is closed and cannot be retried.

At commit, a transaction claims the next slot of a commit ring with one atomic increment and publishes its sorted key list and a 4,096-bit Bloom filter there. It then validates against every entry claimed since its window began, skipping aborted entries and those already visible in its snapshot. Two entries conflict if their key sets intersect, which is decided by the filter first and an exact search of the sorted keys second. An entry whose owner has not yet decided counts as a conflict, so an earlier commit that later aborts can abort a later one for nothing. The ring holds 65,536 slots, and at most 32,768 commits are admitted at once.

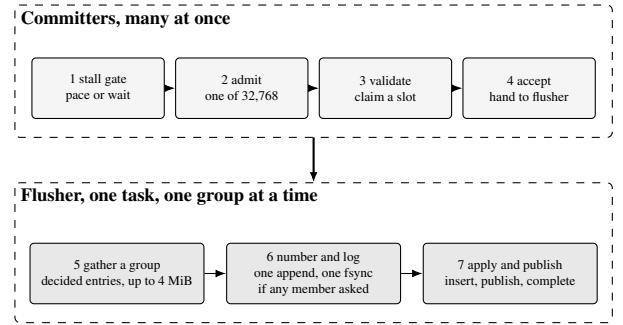


Figure 2: The commit pipeline. Committers take steps 1 to 4 concurrently. One flusher task takes steps 5 to 7.

4.2 The pipeline

Figure 2 shows the pipeline. Committers run it concurrently up to step 4. The flusher runs the rest, one group at a time.

A committer first passes the write stall gate, which may pace it or hold it while the engine is behind on flushes (Section 5). It takes an admission permit, claims a slot and validates, and then hands its batch to the flusher and waits.

From the permit to the verdict nothing awaits, so a cancelled commit can never leave a claimed slot undecided.

The flusher gathers every decided entry after the last one it handled, in ring order, up to 4 MiB of batch bytes. It assigns sequence numbers to the group, writes any values above the value-log threshold to the value log, and appends the whole group to the write-ahead log in one call. If any member asked for a sync, it syncs once for the group. It then applies the group to the active memtable, stores the new published sequence number, marks the entries visible and wakes the committers. There is no linger timer: a group is whatever accumulated while the previous group was being written.

The commit returns when the flusher has finished the group. It does not wait for the memtable to be flushed to a table or for any compaction. Only the flusher writes to the active memtable, so applying a group is not parallel. The pipeline's concurrency is in validation and in reads. If a group fails before anything is applied, the whole group fails and the log is cut back to its length before the group. If it fails after part of it was applied, the database stops and recovery decides the outcome at the next open.

4.3 Durability

A transaction chooses between two modes. The default, *Eventual*, writes the group to the operating system before the commit returns and does not call *fsync*. A committed transaction then survives a crash of the process but not of the machine, and nothing syncs the log on a timer. *Immediate* syncs once for the group before the group is applied and acknowledged, so a commit is durable before it is visible. One *fsync* serves every commit in the group, so an *Eventual* commit that shares a group with an *Immediate* one is synced too.

4.4 Invariants

Five invariants carry the argument that the pipeline and the structures under it give consistent snapshots and safe reclamation. We state them as the code establishes them.

Read order at begin. A transaction that can write registers its pin in the active-transaction tracker, reads its commit window, and only then reads the published sequence number. The order is required: it puts every commit at or below the window inside the transaction's snapshot.

Publish after apply, in order. The flusher advances the published sequence number only after a group is fully applied to the memtable, then marks the group's entries visible and advances the completed prefix of the ring. A reader therefore never sees part of a group.

Values before the log. Values reach the value log, flushed or synced to match the group, before the log record that points to them, and a log segment is not sealed before the value log is synced. A record in the log never points to a value that is not there.

Flush ordering. A flush adds the new level-0 table and removes the memtable from the immutable queue while holding both the manifest's and the queue's write locks. A reader that misses the memtable finds the table.

Compaction reads the horizon first. The compactor reads the published sequence number, then the list of registered snapshots, and never the log sequence number. A restore may lower the published number, which only makes compaction keep more. One window remains, and the code documents it: a transaction that has taken its sequence number but not yet registered its snapshot can be missed.

4.5 A worked example

Consider three transactions that begin when the published sequence number is 100. T_1 writes keys a and b , T_2 writes c and T_3 writes b . They commit in the order T_1 , T_2 , T_3 , each claiming the next ring slot. T_2 's keys do not intersect T_1 's, so it passes validation. T_3 shares b with T_1 , whose entry is not visible in T_3 's snapshot, so T_3 aborts with a write conflict and its slot is marked aborted.

The flusher wakes, finds T_1 and T_2 decided and takes them as one group, skipping the aborted slot. It numbers the three key operations 101, 102 and 103, appends both records to the log in one call, syncs once if either transaction asked for *Immediate*, applies them to the memtable, stores 103 as the published sequence number and wakes both committers. A transaction that begins now has snapshot 103 and sees all three writes, and one that began at 100 sees none of them. If T_3 is retried it begins at 103, reads T_1 's b in its snapshot, and can commit.

5 MEMORY, LOG AND BACK-PRESSURE

Memtable. The active memtable is a concurrent adaptive radix tree [7] with per-key version chains, built from the *artmap* crate. Its nodes, leaves and versions are allocated from a fixed arena whose size is the memtable size. When the next group does not fit, the memtable is sealed, joins the queue of immutable memtables and is replaced. A batch too large for any memtable is written straight to a level-0 table. A background task flushes immutable memtables oldest first, one at a time, writing each as a level-0 table, updating the manifest atomically and deleting the log segments the flush made obsolete. A flush that fails with an error short of fatal is retried with a backoff from 50 ms to 2 s; an I/O error while writing the table is fatal and stops the database.

Write-ahead log. The log is a sequence of 32 KiB blocks with a 7-byte header per record holding a CRC-32 checksum, a length and a type. Each committed transaction is one record, so it is recovered whole or not at all, and a group is appended in one call. Each memtable generation has its own segment. Large values reach the value log before the log record that points to them.

Back-pressure. The gate checks two counts before every commit: immutable memtables and level-0 tables.

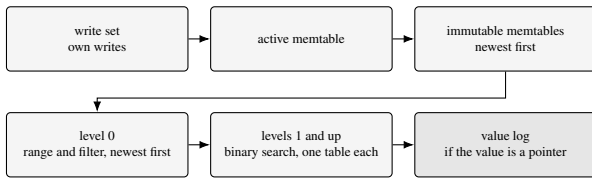


Figure 3: Where a point read looks. The first match at or below the snapshot ends the search.

At 12 level-0 tables or two immutable memtables, commits wait until a flush or compaction clears the condition. Above 8 level-0 tables, each commit is delayed by 100 microseconds for every table over that number. With the default settings the memtable count reaches its hard limit before any pacing applies to it.

6 THE READ PATH

Figure 3 shows where a point read looks. It checks the transaction’s own write set first, so a transaction sees its own writes, and then the snapshot’s sources from newest to oldest, stopping at the first match at or below the snapshot.

Before the point search, the snapshot collects the largest range-delete sequence that covers the key from the active memtable, every immutable memtable and the metadata of every table. The check is cheap when no range deletes exist, since each table holds an atomic flag, but it costs a pass over every table when any table has them.

Within a table, a read probes the Bloom filter partition, binary-searches the top-level index in memory, loads one index partition and one data block through the cache, and seeks in the block by its restart points. The stored value carries a two-byte header, and if it marks a pointer the read fetches the entry from the value log, itself cached. A synchronous read holds the level manifest’s read lock while it reads tables. The asynchronous variant collects its candidate tables under the lock, releases it, and offloads only the data-block read.

Scans merge their sources with a cursor that is positioned in every source and picks the smallest key by a linear scan over all of them at each step. Each overlapping table in a deeper level is its own source, there is no per-level iterator, and a backward step re-scans from the previous restart point of a block and copies the key and value. A block entered by a point read or a scan is copied into its iterator. These costs are consistent with the scan results in Section 14, though we have not measured their share.

7 TABLES, CACHE AND COMPACTION

Tables. A table is a sequence of data blocks of about 64 KiB, with a restart point every 16 entries and prefix compression between them. Figure 4 shows the layout. Its index is two-level: the top-level index is read into memory when the table opens, and index partitions of about 16 KiB are fetched on demand. Its Bloom filter is partitioned in step with the index and built over user keys, at 10 bits per key and 7 probes by default. Every block ends in a compression byte and a masked CRC-32 checksum that is verified whenever the block is read from disk. A fixed 50-

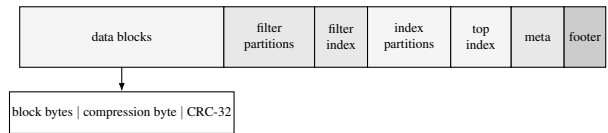


Figure 4: The layout of a table. Every block ends in a compression byte and a masked CRC-32 checksum.

byte footer ends the file. Compression is off by default, and Snappy and Zstd are available per level.

Reads and the cache. Index partitions, filter partitions, data blocks and value-log values share one byte-weighted, sharded cache of 32 MiB by default. Scans and point reads cache the same block under different keys, so one block can be cached twice. Synchronous reads are plain positional reads on the calling thread, and the engine uses no memory mapping and no kernel asynchronous I/O.

Compaction. Compaction is leveled. Level 0 is compacted when it holds four tables or 256 MiB. Each deeper level has a fixed target, ten times the one before, starting at 256 MiB for level 1. The level with the highest score is chosen, and within a level the table with the largest size, weighted up by its share of deletions. Every selected input is rewritten, and output files are about 64 MiB and are cut only between user keys. One compaction runs at a time. A compaction that fails removes its partial outputs, unless the manifest write is uncertain, and is tried again with a backoff from 50 ms to 2 s; writes continue until level 0 reaches its stall limit, and corruption in an input stops the database. Compaction keeps every version that a registered snapshot can read, everything above the published sequence number, and the newest published version of each key. In an unversioned tree it drops hard delete markers only when writing the last level; a versioned tree keeps more (Section 8).

Value log. Value separation is off by default. When it is on, a value longer than 1 KiB is appended to a value log and the tree stores a 25-byte pointer. Files rotate at 256 MiB. A file is deleted only when it lies wholly below the smallest file referenced by any live table or unflushed memtable, and a commit group pins the files its values go to until the group is applied. When no table points into the log, every file but the active one is deleted. Space is therefore reclaimed in file order, and one long-lived value pins every later file. There is no rewriting collector.

8 VERSIONED TREES

A tree can be opened versioned, with a retention in nanoseconds, where zero keeps every version for ever. A versioned tree keeps the history of its keys and answers reads by time. The default is unversioned, and an unversioned tree writes the same records as an engine with no versioning at all. Versioning needs neither the value log nor an index of its own, because each of its parts reuses a structure the engine already has.

Table 3: What a compaction keeps, beyond the versions that readers can see.

Setting	Kept beyond what readers see
Unversioned	Nothing. The newest version, and what each snapshot reads.
Versioned, retention 0	Every version that no barrier hides.
Versioned, retention r	The versions written at or after $T - r$, and, for every reader, the newest version before that time.

A time is a stamp in the value. Every write of a versioned tree is stamped with its time, the time of its commit or one the caller gives with `set_at`; a time of zero means the commit time. The stamp is eight bytes in the record that holds the value, so the log, the memtable, the tables, the value-log pointers and replay all carry it. A soft delete adds a tombstone that is itself a version, and is stored as the stamp alone with a two-byte header. A hard delete, a range delete and a replace are barriers: they hide whatever was written before them. A transaction writes one version of a key for each time, the last of its writes to it, whatever savepoints came between.

Order stays by sequence number. The tree is ordered by sequence number, as an unversioned tree is, and a time is a property of the value. A time can therefore go backwards: a write made later can carry an earlier time. Reading by time is a rule over the versions of a key, and not a seek to a position.

Reading. `get_at` returns the version of a key with the newest time at or before the time given, taking the newest write among versions of one time. It reads the versions of the key that the snapshot can see once, oldest write first, holding only the best so far. `history` lists the versions of the keys in a range, the versions of a key newest time first, forwards or backwards, with tombstones, a range of times and a limit as options, and holds one key’s versions in memory while it returns them. Both honour the transaction’s own pending writes and range deletes. A version written while the tree was not versioned has no time, and reads at the time of the versions written before it.

Compaction keeps history by the rule it uses for snapshots. Table 3 says what each setting keeps. With a retention r , a compaction reads the time T once, when it starts, and keeps the versions written at or after $T - r$ together with the version that a read at $T - r$ sees, which is the newest version before that time. It does so for every reader there is: the readers of the newest versions, the one at the horizon, and each snapshot, which sees only what was written at or before it. A barrier above the last level is kept as a rule, since tables that the compaction does not read may hold what it hides; at the last level a barrier is kept only while a version below it is kept. A version written without a time that is kept at the last level is stamped with the time it reads at.

What it costs. The stamp adds eight bytes to every version, and a soft delete, which has no value, is ten. A read

by time reads every version of its key, so its cost grows with the history kept, and `history` holds one key’s versions in memory. A compaction that applies a retention decides over the versions of each key, and allocates to do it. We have not measured any of these.

9 DESIGN LINEAGE: LEVELDB, ROCKSDB AND PEBBLE

SurrealKV belongs to the family that LevelDB’s implementation notes [13] describe: a log, a memtable, immutable tables in levels and a manifest. Table 4 sets its choices beside LevelDB’s, as those notes state them, and Table 5 beside RocksDB’s and Pebble’s, as Pebble’s account of its differences from RocksDB states them [4]. Both tables compare designs as documented, not measurements, and a later release of any of the engines may differ.

What it shares. SurrealKV shares LevelDB’s leveled shape, its overlapping level 0 and its trigger of four tables, and the arrangement in which one log segment belongs to one memtable. Like Pebble’s, its arena is fixed in size, so no batch can grow a memtable without bound.

Where it differs. It raises every size: tables of about 64 MiB against LevelDB’s 2 MB, and a first level of 256 MiB against 10 MB. LevelDB’s notes name the trade, which is fewer files at the expense of more bursty compactions. Its memtable index is a radix tree and not a skiplist, and an oversized batch goes straight to level 0, where Pebble keeps it as a sorted layer of the immutable list. Its manifest is a file that is replaced and not a log that is appended to. It slows writers as level 0 fills, as RocksDB does and as LevelDB’s notes propose, where Pebble adds no delay. It does not pace background I/O: Pebble’s notes describe un-paced flushes and compactions as occurring as fast as possible, with latency spikes for concurrent reads and writes, and our measurement does not examine latency.

Where it is more conservative. Two rules are more conservative than LevelDB’s. A delete marker survives until the last level, which costs space where LevelDB drops it as soon as no deeper level overlaps its key; and opening a store loads every table, which front-loads work that the notes prefer to defer.

10 A COST MODEL

Table 6 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 11.

11 WHERE THE PERFORMANCE COMES FROM

Table 7 sets each 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 14 does not apportion the speed-up among them.

Table 4: SurrealKV beside LevelDB, as LevelDB’s implementation notes describe it.

	LevelDB	SurrealKV
Log and memtable	One log; at about 4 MB it becomes a sorted table and a new log starts; the memtable is the log’s in-memory copy	One log segment per memtable generation; the memtable is an arena of 100 MiB, and one background task flushes a sealed memtable to level 0
Level 0	Compacted into level 1, together with the overlapping level-1 files, when four files exist	Compacted when it holds four tables or 256 MiB
Level sizes and files	Level L holds up to 10^L MB, so 10 MB at level 1; files of 2 MB	256 MiB at level 1 and ten times more at each level below, six levels; output files of about 64 MiB
Choosing a compaction	One file of level L , taken in turn through the key space from where the last compaction at that level ended	The level with the highest score and, within it, the largest table, weighted up by its share of deletions
Delete markers	Dropped when no higher-numbered level holds a file whose range overlaps the key	Hard delete markers dropped only when writing the last level
Manifest	A log of changes, appended to; a new manifest on every reopen, named by a <code>CURRENT</code> file	One file listing the tables of every level, replaced atomically by a temporary file, a sync and a rename; ends in a checksum trailer
Tables at open	The notes prefer to open tables lazily	Opens every table the manifest lists
Slowing writers	The notes propose slowing writers as level-0 files accumulate	100 microseconds per level-0 table above eight, and a stall at twelve

Table 5: SurrealKV beside RocksDB and Pebble, as Pebble’s notes describe the first two.

	RocksDB	Pebble	SurrealKV
Memtable index	Skiplist	Skiplist, linked by offsets from the start of its arena	Adaptive radix tree
Arena	Fixed-size chunks, with no upper limit on their number	One arena of fixed size	One arena of fixed size, 100 MiB by default
Batch larger than a memtable	The memtable grows to take it; the notes warn that this can exhaust memory and repeat on restart	The memtable is made immutable and the sorted batch joins the immutable list as one more layer	Written straight to a level-0 table
Write throttling	Artificial delays as thresholds such as the level-0 slowdown are approached	None; writes are served as fast as possible	100 microseconds per level-0 table above eight, then a stall at twelve tables or two immutable memtables
Background pacing	An optional rate limit on flush and compaction I/O, fixed or auto-tuned	Flushes paced to the rate of user writes and compactions to an estimate of compaction debt	None

12 RECOVERY AND INTEGRITY

Open. Opening a tree refuses a directory that holds data but no usable manifest, so an open never silently creates an empty database. The manifest is a single file listing table identifiers per level, replaced atomically by writing a temporary file, syncing it, renaming it and syncing the directory. It ends in a 16-byte trailer that holds its length and a checksum, which opening verifies; a manifest without one, written by an earlier release, is accepted and gets one the next time it is written. Opening loads every table the manifest lists. The log is then replayed from the manifest’s log number, one memtable per segment, and the engine continues in a fresh segment so that a damaged segment is never appended to. Table files the manifest does not list, value-log files below the smallest referenced one, and the stale temporary files of earlier manifest writes are deleted.

Failures. Table 8 says what the engine does for each failure it handles. The default recovery mode trims only a

proven torn tail: a damaged record at the end of the newest log segment, with no intact record after it. It copies the segment to a quarantine file, keeping the last three copies, and truncates the segment in place. Damage with intact records after it, or in any older segment, is refused, and the error names the segment, the offset and how many intact records would be lost. Two other modes exist. One refuses every damaged log without touching a file. The other, chosen explicitly, keeps the valid prefix of the segment and discards what follows it. A report says what recovery replayed, dropped, trimmed and quarantined.

Integrity. Table blocks carry a masked CRC-32 checksum, the IEEE polynomial and not CRC-32C, checked on every read from disk and not re-checked in the cache. Log records carry a CRC-32 checked at recovery. An off-line command reads every block of every table, of all six kinds, and reports failures. The manifest carries a checksum. There is no background scrubber. A database created with a key manager is encrypted at rest: the manifest,

Table 6: What each operation’s cost grows with. m is the number of immutable memtables, z the level-0 tables that pass the key range and filter, L the number of levels, w the commits in a window and s the sources of a scan.

Operation	Cost grows with
Point read	m memtable lookups; z level-0 probes; one table per level for L levels, each a filter probe and at most one index partition and one data block, cached or read; one more read for a separated value
Scan	s sources, each positioned, and a comparison with every source at each step
Commit: validate	w range tests and filter probes, and a binary search of the commit’s keys on each hit
Commit: flush of a group	The group’s bytes, up to 4 MiB: one append, at most one fsync, and a sequential insert of every key
Memtable flush	The memtable’s size: one level-0 table
Compaction	Every selected input is rewritten; with a factor of ten between levels, a key is rewritten at each level it reaches
Read by time	Every version of the key that the snapshot can see, in every memtable and table, read once; <code>history</code> holds one key’s versions in memory
Compaction, versioned	As above, and a decision over the versions of each key, with buffers that grow with them

tables, value log and log are sealed with AES-256-GCM or XChaCha20-Poly1305 under a key of their own, derived from the root key, and every block is bound to its file and position, so a block that is moved or edited is refused. Value-log checksums are checked only if the tree is configured to.

Checkpoints. A checkpoint flushes the memtables, hard-links every table into the destination, copies the manifest and the value log, writes an empty log directory, syncs what it has written, and writes its metadata file last, so that an interrupted checkpoint does not look complete. It refuses a destination that is the database directory, lies inside it or contains it, and writes to one destination at a time. Restoring validates the whole checkpoint, including the manifest and every table it lists, before it removes anything. It writes a marker before the first removal and removes the marker after everything restored is synced, and opening a directory that holds the marker fails. It waits for running compactions, and fences commits that were in flight so that none reaches the restored state. A restore is not atomic, but an interrupted one is detectable.

13 VERIFICATION

A seeded differential test drives the engine and an in-memory reference model with the same actions: begin, set, delete, delete range, savepoints, rollback, reads, commit, range scans, a log flush and a restart. Any divergence in a read, a scan, a commit verdict or the full state after a restart fails the run. Continuous integration runs 500 seeds of 5,000 steps over a pool of 30 keys with up to 12 transactions open. It is not a simulator in the sense of deterministic simulation testing [14]: time, files and scheduling are real, no fault is injected, and its restart is a graceful close and reopen. Its workload runs the default configuration, so the value log, compression and versioning are

not exercised. It does not generate soft deletes, replaces, locked reads or backward iteration, and its commits are never concurrent.

A second randomised model covers versioned trees. It checks `get`, `get_at`, `history`, scans and iterator walks against a model written from the rules of Section 8, over transactions, savepoints, flushes, compactions to deep levels, crashes, writes made without versioning, a transaction on an old snapshot, retention, and trees of two levels. The longer test runs 300 sequences.

Other suites cover the cases that test does not. They emulate power loss at the file-system contract by letting unsynced files come back empty, tear the log at many byte offsets across restarts in every recovery mode, cut a crash image at any byte of a grouped commit, inject failures into fsync, manifest replacement and value-log writes, and check that checkpoints refuse overlapping destinations. Further suites take a copy of the database directory at each step of a compaction, a level change, a checkpoint and a restore and reopen it, exercise the value-log collector, race restores against commits, flushes and checkpoints, and alter one bit at every byte of a manifest and require each alteration to be refused. The compaction retention rule is checked against a model of readers over more than 500,000 small cases of three kinds of entry: sets, deletes and range deletes. Property-based tests cover only the encryption layer, and one randomised test exercises the log reader; there is no coverage-guided fuzzing.

14 EVALUATION

14.1 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 and a 4 TB PCIe 4.0 NVMe drive. They run 500,000 keys across 48 worker threads and 128 clients, and every figure is a single run. In the harness SurrealKV runs with a 256 MiB memtable, 64 KiB blocks, a cache sized from the machine’s memory, value separation on and versioning off. The comparators are RocksDB, SlateDB (on disk, not object storage), Fjall, ReDB, LMDB and libmdbx, described in Table 1. LevelDB and Pebble are not measured. RocksDB runs through optimistic transactions with a 256 MiB write buffer, level compaction and separate blob files for values of 4 KiB and over.

Each engine is measured in up to two configurations, which one flag of the harness selects. *With synchronous durability*, a write is on stable storage before it is acknowledged. *Without it*, the write is handed to the operating system or held in memory, and what a crash leaves behind depends on the engine. Table 9 gives each engine’s setting in the two. SurrealKV, RocksDB and SlateDB were measured in both. ReDB, Fjall, LMDB and libmdbx were measured only without synchronous durability, because with it their runs were too slow to complete in the time available. Reads do not depend on durability, but each configuration is its own run, so the read figures differ a little between them.

The machine runs Ubuntu 24.04 with kernel 6.8.0-146, the `powersave` frequency governor, transparent

Table 7: The design decisions that determine throughput, and their prices.

Decision	What it spares	What it costs
One flusher turns concurrent commits into one group	One log append and at most one fsync for a whole burst of commits, and sequence numbers assigned in order with no coordination between committers.	Applying to the memtable is single-threaded, and one slow fsync delays every commit in its group.
Validation against a ring, not a lock table	No per-key lock or state: a commit claims its place with one atomic increment and reads only the slots in its window.	Validation grows with the commits that ran during the transaction, and a long-open write transaction lets the ring overflow into memory that is not bounded.
An arena memtable	Nodes, leaves and versions are allocated from a bump arena, and readers of the tree do not block, as the crate documents.	The arena size is fixed, and when the next group does not fit the memtable is rotated.
Reads pin only a sequence number	No read lock and no per-key bookkeeping for a reader.	A synchronous read holds the level manifest’s read lock across its table reads, so a manifest swap waits.
Partitioned index and filter behind one cache	A table costs a small top-level index in memory, not its whole index and filter.	A cold partition costs an extra read, and a scan and a point read cache the same block separately.
Value separation, when on	Compaction moves keys and pointers, not large values.	A read of a large value costs one more read, and space is reclaimed only as a prefix of files.
A time stamped in the value, versions ordered by sequence number	A time index, the value log, and any change to the records of an unversioned tree.	Eight bytes a version; a read by time reads every version of its key; a later write can carry an earlier time.

Table 8: What the engine does on each failure.

Failure	Behaviour
Data but no manifest, or a manifest that fails to decode or fails its checksum	The open is refused
Torn record at the end of the log	Quarantined and trimmed in place; earlier acknowledged commits survive
Damage with intact records after it, or in an older segment	Default: refused, with the segment, offset and records at stake. Salvage mode: the valid prefix is kept and the rest discarded after quarantine. Strict mode: refused, nothing touched
Failed fsync of the log	The segment is poisoned and the next group replaces it; a commit that returned an error can still take effect at the next open
Failed memtable flush	Retried with a backoff of 50 ms to 2 s if the error is short of fatal; stalled writers are released with an error. An I/O error writing the table is fatal
Failed compaction	Partial outputs removed; retried with a back-off of 50 ms to 2 s; writes continue until level 0 reaches its stall limit. Corruption in an input stops the database

huge pages in `madvise` mode and an 8 GB swap file. The harness uses the `mimalloc` allocator. The figures are for SurrealKV 1.0.0. The comparators are the `surrealdb-rocksdb` crate 0.24.0-surreal.5, which wraps RocksDB 11.0.0, `slatedb` 0.15.0, `fjall` 3.1.10, `redb` 4.3.0, `heed` 0.22.1 over `lmdb-master-sys` 0.2.6 for LMDB, and `libmdbx` 0.6.6. The harness is `crud-bench`.

14.2 What each operation measures

Table 10 says what each harness operation does in the SurrealKV adapter. Every operation is a transaction, including the reads.

Table 9: What the harness’s synchronous flag sets in each engine’s adapter.

Engine	With	Without
SurrealKV	Immediate: one fsync for the group	Eventual: written to the operating system, no fsync
RocksDB	Sync option on every write	Log written without sync
SlateDB	Commit awaits durability; flush every 1 ms	Commit does not wait; flush every 100 ms
Fjall	Journal synced on every write	Journal buffered
ReDB	Immediate	None: not persisted until a later durable commit
LMDB	Data and metadata synced at commit	No sync of data or metadata, writes through the memory map flushed by the operating system
libmdbx	Durable	UtterlyNoSync

14.3 Throughput

Table 11 gives operations per second with synchronous durability and Table 12 without it. Table 13 gives SurrealKV’s throughput divided by each comparator’s in the same configuration.

Reads. Reads are the figure that Section 11 predicts most directly: a read pins a sequence number, searches an arena memtable and then tables that are cheap to probe. SurrealKV reads 8.3 times faster than RocksDB and 4.8 times faster than LMDB, the fastest comparator. Its two runs agree to within one percent.

With synchronous durability. Writes benefit from the grouped pipeline, which lets concurrent commits share one append and one fsync. SurrealKV’s creates are 26.8 times RocksDB’s and 27.8 times SlateDB’s, its updates 6.1

Table 10: What each measured operation does in the harness’s SurrealKV adapter.

Operation	What the adapter does
Read	Begin a read-only transaction, get the key, decode the value
Create, update	Begin a read-write transaction, set its durability, set one key, await the commit
Delete	As a write, with a delete
Scan	Begin a read-only transaction, open a cursor over the whole key range, skip to the start and read up to the limit

Table 11: Operations per second with synchronous durability, 500,000 keys, 48 worker threads, unversioned trees.

Engine	Read	Create	Update	Delete	Scan
SurrealKV	6,804,864	229,376	221,341	264,089	905,132
RocksDB	855,851	8,568	36,263	32,043	852,661
SlateDB	92,303	8,257	8,122	8,242	17,985

and 27.3 times, and its deletes 8.2 and 32.0 times. We have not measured how much of the difference the grouping accounts for. RocksDB’s synchronous creates are a fraction of its synchronous updates, 8,568 against 36,263, and we have not examined why.

Without synchronous durability. RocksDB writes faster than SurrealKV: its creates are 1.9 times SurrealKV’s, its updates 2.2 times and its deletes 2.4 times. SurrealKV is ahead of every other engine on all three. With no fsync to share, the grouped pipeline has nothing left to amortise, and one flusher still applies every group to the memtable, so insertion runs on one core (Section 16). We have not measured whether that is the limit. Moving SurrealKV from Immediate to Eventual raises its creates by 1.4 times, its updates by 1.2 times and its deletes by 1.2 times, although the two figures come from separate runs.

The other engines. LMDB and libmdbx admit one writer at a time, so the harness’s 128 clients queue behind it. libmdbx’s file grew past 160 GB in UtterlyNoSync mode, and its memory in Table 15 reflects that growth; we have not established why the file grew, and its low update figure may be related. Both describe that mode and not the engine at its best. Fjall deletes at 2,676 per second against 62,608 creates, and we have not examined why.

14.4 Scans

Table 14 gives five scan shapes over the same keys, without synchronous durability. *Limit* returns the first 100 key-value pairs of the range. *Keys* returns only the keys. *Skip* walks and discards 5,000 entries before returning 100, and *Keys, skip* does the same for keys only. *Count* visits every key in the range.

LMDB is faster than SurrealKV on whole-entry limit scans (by 1.3 times), skip scans (5.7 times) and counts (2.2 times), and slower on key-only limit scans, where SurrealKV leads by 1.9 times. libmdbx is faster on skip scans (1.9 times) and counts (2.1 times). The gaps on

Table 12: Operations per second without synchronous durability, 500,000 keys, 48 worker threads, unversioned trees.

Engine	Read	Create	Update	Delete	Scan
SurrealKV	6,992,767	321,715	274,172	307,034	896,366
RocksDB	846,293	603,452	608,494	739,357	851,848
SlateDB	96,397	6,314	6,352	6,380	26,187
Fjall	992,874	62,608	64,059	2,676	132,698
ReDB	779,262	35,171	35,805	65,686	561,837
LMDB	1,461,261	127,045	163,099	157,965	1,192,772
libmdbx	613,904	24,378	5,026	11,521	384,128

Table 13: SurrealKV’s throughput divided by each comparator’s, from Tables 11 and 12. A ratio below one means the comparator is faster.

Against	Read	Create	Update	Delete	Scan
<i>With synchronous durability</i>					
RocksDB	8.0×	26.8×	6.1×	8.2×	1.1×
SlateDB	73.7×	27.8×	27.3×	32.0×	50.3×
<i>Without synchronous durability</i>					
RocksDB	8.3×	0.53×	0.45×	0.42×	1.1×
SlateDB	72.5×	51.0×	43.2×	48.1×	34.2×
Fjall	7.0×	5.1×	4.3×	115×	6.8×
ReDB	9.0×	9.1×	7.7×	4.7×	1.6×
LMDB	4.8×	2.5×	1.7×	1.9×	0.75×
libmdbx	11.4×	13.2×	54.6×	26.7×	2.3×

skip scans and counts are consistent with the scan costs of Section 6 and with the structure of those engines, which walk one tree and not several sorted runs. SurrealKV’s limit scans are level with RocksDB’s, 1.1 times, a difference that one run cannot separate from variation, and it is ahead of RocksDB, Fjall, ReDB and SlateDB on every other shape.

14.5 Memory

SurrealKV’s resting footprint, 221 MiB, is the lowest of the seven, just below ReDB’s 257 MiB and SlateDB’s 265 MiB. Its peak, 1.79 GiB, is above RocksDB’s, LMDB’s and Fjall’s and below ReDB’s and SlateDB’s. Peak memory expands under concurrent commits and contracts afterwards.

14.6 Threats to validity

Internal. The figures are for SurrealKV 1.0.0, and the comparators are the versions given in the method. Every figure is a single run, so run-to-run variation is not measured and small differences should not be read as an ordering.

Construct. The harness runs SurrealKV with value separation on and a larger memtable than the engine’s defaults, so the figures describe that configuration. Every operation is a transaction, so the read figures include beginning and ending a read-only transaction, and the write figures include 128 clients committing concurrently, which is the condition under which grouping pays. The configurations without synchronous durability are not equivalent across engines. SurrealKV’s Eventual commit survives a crash of the process but not of the machine, whereas

Table 14: Scans per second over 500,000 keys, without synchronous durability.

Engine	Limit	Keys	Skip	Keys, skip	Count
SurrealKV	896,366	2,313,942	96,365	98,612	402
RocksDB	851,848	851,580	35,539	35,902	159
SlateDB	26,187	26,266	7,460	7,477	87
Fjall	132,698	151,861	4,335	4,383	42.4
ReDB	561,837	571,629	44,438	44,462	249
LMDB	1,192,772	1,195,166	549,220	552,090	902
libmdbx	384,128	260,769	183,361	185,073	848

Table 15: Resting and peak process memory sampled over each engine’s whole run without synchronous durability.

Engine	Resting	Peak
SurrealKV	221 MiB	1.79 GiB
RocksDB	590 MiB	1.02 GiB
SlateDB	265 MiB	2.73 GiB
Fjall	627 MiB	1.71 GiB
ReDB	257 MiB	2.17 GiB
LMDB	776 MiB	826 MiB
libmdbx	40.98 GiB	117.71 GiB

ReDB’s `None` does not persist a commit until a later durable one, and LMDB and libmdbx document that with their no-sync flags a crash of the machine can corrupt the database or roll it back by an arbitrary amount. We have not tested what survives in any of them. The comparison without synchronous durability therefore measures speed and not safety, and only Table 11 compares engines that each flush before acknowledging.

External. The workload is single-key operations on generated values and 500,000 keys, small beside the machine’s 128 GB, so it says little about reads that miss the cache and reach the disk. The figures come from one machine and are best read as ratios.

14.7 What the measurement does not cover

Eight measurements would test claims this paper makes from the structure of the code and have not been run: throughput as cores are added; the effect of the group size; value separation on and off; behaviour under sustained writes, where the stall gate and compaction interact; datasets larger than memory; mixed read and write workloads and long-running transactions; versioned trees against unversioned ones, for writes, reads by time and compaction; and ablations that remove one decision of Table 7 at a time. A latency distribution under load is also missing, and so is a measurement of ReDB, Fjall, LMDB and libmdbx with synchronous durability.

15 DISCUSSION

Where it fits. SurrealKV suits a persistent embedded engine for a database whose workload is dominated by point reads and many small, concurrent, durable writes, which is the case its pipeline is built for. Grouping makes synchronous durability cheaper per commit as concurrency grows. It also suits a database that must read its keys as of a time, since a versioned tree keeps that history itself.

Where it does not. It is not a fit where a serializable mode is needed, where writes without synchronous durability are the whole workload, since RocksDB writes faster in the measurement above, or where scans dominate: LMDB is ahead on whole-entry scans and libmdbx on skip scans and counts, and the read path of Section 6 suggests why.

Trade-offs. Two decisions deserve a second look. Applying a group on one thread keeps ordering simple and avoids the synchronisation points that Pebble’s notes criticise, but it caps memtable insertion at one core. And the value log’s reclamation is the simplest that is correct, deleting a prefix of files, at the price that one long-lived value holds every later file.

Open questions. A concurrent memtable apply; a re-writing value-log collector; parallel compaction; a serializable mode; a simulation that controls time and injects faults; and the ablations above.

16 LIMITATIONS

- **Snapshot isolation only.** There is no serializable mode, and scans are not validated. Write skew is possible.
- **Versioned trees.** A later write can carry an earlier time, a read by time reads every version of its key, and `history` holds one key’s versions in memory. A version written while the tree was not versioned reads at the time of the versions before it, and not at the time it was written. No benchmark of a versioned tree exists.
- **Single flusher.** Applying to the memtable is single-threaded, and the log append for a group runs on the flusher.
- **Single compaction.** One compaction runs at a time.
- **Open write transactions.** A long-open read-write or write-only transaction holds back the retirement of commit-ring entries, and the overflow it causes is not bounded.
- **Value log.** Space is reclaimed only as a prefix of files, and value checksums are off by default.
- **Recovery.** Damage in the middle of a segment, or in an older one, stops the open. The only way past it is the salvage mode, which keeps the valid prefix and discards what follows it, after copying the segment aside.
- **Checkpoints.** A checkpoint flushes first and copies no log. A restore is not atomic and waits for running compactions; an interrupted one is refused at the next open.
- **Encryption.** Encryption at rest is chosen when a database is created and cannot be added to one that exists. It does not hide file names, file sizes or access timing, and its on-disk formats are not frozen.
- **Memory.** The memtable size budgets the arena, not the bytes of keys and values, and no memory limit is enforced.

- **Platforms.** The engine builds for WebAssembly, and the primitives for browser storage exist, but a tree does not persist there yet.

17 CONCLUSION

SurrealKV attains its throughput by grouping commits, validating without a lock table, keeping the memtable in a radix tree in an arena and making reads and table probes cheap, and a versioned tree keeps its history with the same sequence numbers and the same compaction rules. Against the pipelines of RocksDB and Pebble it keeps the group and gives up the concurrent apply, and against LevelDB’s design it raises every size, replaces the manifest log with a replaced file and delays writers as level 0 fills. The measurement shows that on one machine for reads and for writes with synchronous durability. It also shows where it does not lead: writes without synchronous durability, where RocksDB is faster; scans, where LMDB is ahead; and peak memory. The limits above are the ones to weigh before relying on it.

A REPRODUCING THE FIGURES

The harness is `crud-bench`, which is open source. Its SurrealKV adapter sets a memtable of 256 MiB, a block size of 64 KiB, value separation on, versioning off, and a block cache sized from the machine’s memory. Its synchronous flag selects `Immediate` durability, and without the flag the adapter uses `Eventual`. The engine’s other settings are its defaults, listed in Appendix B. Each figure comes from one invocation of the form

```
CRUD_BENCH_CONFIG=config/memory.toml \
./run.sh -d surrealkv --no-build \
--optimised -s 500000 -c 128 -t 48
```

with `-d` naming the engine and `-sync` added for the configuration with synchronous durability. The configuration file sets the document shape and the scans.

B CONFIGURATION REFERENCE

Table 16 lists the engine’s defaults, as the code states them.

REFERENCES

- [1] 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>.
- [2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, pages 205–218, Seattle, WA, November 2006. USENIX Association. URL <https://www.usenix.org/conference/osdi-06/bigtable-distributed-storage-system-structured-data>.
- [3] 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>.
- [4] 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>.
- [5] 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>.
- [6] Fjall. Fjall: LSM-based embeddable key-value storage engine. URL <https://fjall-rs.github.io>. Latest release at access time: 3.1.12, 3 October 2026.
- [7] 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>.
- [8] libmdbx. libmdbx. URL <https://sourcecraft.de/v/dqdkfa/libmdbx>. Latest release tag at access time: v0.14.4, 18 September 2026.
- [9] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, February 2016. USENIX Association. URL <https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu>.
- [10] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (LSM-tree).

Table 16: Defaults.

Setting	Default
Block size, restart interval	64 KiB, 16
Index partition size	16 KiB
Bloom filter	10 bits per key, 7 probes
Compression	none; Snappy and Zstd (level 3) available
Block cache	32 MiB
Levels	6
Memtable size	100 MiB
Level-0 trigger	4 tables or 256 MiB
Level base and multiplier	256 MiB, ten times per level
Output file size	64 MiB
Stall: memtables, level 0	2, 12
Value separation	off; threshold 1 KiB; files of 256 MiB
Value-log checksums	off
Versioning, retention	off; when on, 0 keeps every version
Log recovery mode	trim a torn tail, refuse other damage
Flush on close	on
Commit ring, permits	65,536 slots, 32,768 permits
Group limit	4 MiB
Durability	Eventual

Acta Informatica, 33(4):351–385, June 1996. doi: 10.1007/s002360050048. URL <https://doi.org/10.1007/s002360050048>.

- [11] redb. redb: Rust embedded database. URL <https://www.redb.org>. Latest release at access time: 4.3.0, 15 September 2026.
- [12] SlateDB. SlateDB: an embedded database built on object storage. URL <https://slatedb.io>. Latest release at access time: v0.17.0, 29 September 2026.
- [13] The LevelDB Authors. LevelDB implementation notes. LevelDB repository, doc/impl.md. URL <https://github.com/google/leveldb/blob/main/doc/impl.md>.
- [14] 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”.