

HanuDB: A Bucket-Resident Document Database for a Free-Tier Virtual Machine, and the 108 Experiments That Shaped It

Madhu

Kansas City · Independent

ABSTRACT

We present HanuDB, a single-node document database whose durable state lives in a cloud object bucket rather than on the machine that serves it. The design targets the smallest virtual machine a public cloud offers at no charge: two shared vCPUs sustaining a quarter of a core, one gigabyte of memory and a standard persistent disk delivering a few dozen IOPS. HanuDB combines a log-structured merge tree for document storage with an inverted index for search behind an HTTP API that offers MongoDB-style writes with three per-request durability levels and Elasticsearch-style reads with filters, sort, full-text search and aggregations. A single write-ahead log is group-committed with one data-only `fdasync`, applied to both engines, and shipped to the bucket as hash-chained compressed segments; the document store keeps no journal of its own, and two durability watermarks let recovery replay exactly the records each engine lacks. Physical snapshots make a rebuild proportional to the log tail rather than to the data. The engine was developed through 108 controlled experiments, each a single change benchmarked against a fixed workload with hard gates for recovery, memory and cost and kept or discarded by a rule written down before the first run, and we report the experimental method in full: the workload, the harness, the acceptance rule, the correctness suite and the long-running checks. On a one-core, 768 MB harness the benchmark score rose from 9 047 to 40 116 (34 821 writes/s and 46 218 reads/s); disk bytes written per thousand documents fell from 4.54 MB to 2.06 MB; a ten-minute soak wrote 14 million documents and rebuilt from a 4.5 GB snapshot in 40 s; and on a real free-tier machine the system serves several thousand operations per second in bursts for a measured cost of about one cent per day. We report every negative result, thirteen soak failures and the design change each produced, and the lessons, several of which reversed conventional assumptions about background work, allocators and memory accounting on a single shared core.

Keywords: document database, object storage, write-ahead log, log-structured merge tree, inverted index, group commit, automated performance engineering, free-tier computing.

The engine is named `twindb` in the source tree, the binary and the configuration variables, after its twin-bucket design; HanuDB is the product name. This paper uses HanuDB throughout.

1 Introduction

Small applications need a database that costs nothing while idle, survives the loss of its host, and answers interactively. The motivation is practical. A developer who runs many small applications pays for a managed document database and a managed search cluster per application, at a minimum price that does not fall with idleness and a storage price that never falls at all. Public clouds now offer both halves of an alternative at no charge: a tiny always-free virtual machine and durable, regionally replicated object storage priced in cents per gigabyte-month. What is missing is a database that treats the object bucket, not the machine's disk, as the source of truth while still serving reads and writes at local-disk latency.

Two existing families approach this from opposite sides. Bucket-native storage engines such as SlateDB [1] place the log-structured merge tree itself in object storage; they are durable by construction but pay object-store latency on every cache miss and tens of seconds to open. Replication tools such as Litestream [2] ship a local SQLite database's write-ahead log to a bucket; they keep local latency but inherit SQLite's write model and offer no search. Prior measurements by the author on the same virtual machine class placed a local engine with a shipped log at ten-microsecond reads and one-second takeovers, against hundred-millisecond misses and ten-second opens for the bucket-native approach [3]. That comparison fixed HanuDB's architecture: serve from local disk, make the bucket the durable home, and never let a read touch the bucket.

The second theme of this paper is how the engine was built. Rather than tuning by intuition, we fixed a benchmark and a correctness suite on the first day, wrote down a set of hard gates that define what is inadmissible and a rule for keeping or discarding a change, and then changed one thing at a time, measuring each change in a container shaped like the target machine. Over 108 numbered experiments, 74 were kept, 27 discarded and 3 crashed. The experiments found the changes one would expect, such as filter caches and log-file handling, and several one would not, including that a second thread on a single shared core is a net loss, that the allocator determines how much memory a rebuild keeps, and that resident set size is the wrong quantity for an out-of-memory guard inside a control group.

This paper makes the following contributions:

- A design for a document database with local-latency reads whose durable state is entirely in an object bucket, including a single-log durability model with per-engine watermarks (§3.3), a hash-chained segment format with a create-if-absent fencing rule (§3.4), and physical snapshots that make recovery proportional to the log tail (§3.5).
- An experimental methodology with a fixed workload, hard admissibility gates, a noise-aware acceptance rule and a separate correctness suite (§4), described in enough detail to be reproduced.
- Measurements on a one-core harness (§5.1–5.5), in ten-minute soaks up to 14 million documents (§5.6), and on a real free-tier machine with real buckets in two regions (§5.7), together with every negative result (§5.5) and every soak failure (Table 7).
- A catalogue of design lessons for engines that must share one core with their own background work (§6).

2 Related Work

Object storage as the durable home. Litestream [2] and Turso [4] stream SQLite's write-ahead log to object storage and restore from it; HanuDB adopts the same posture with a document model, full-text search and a hash chain over the shipped segments. SlateDB [1] and similar bucket-native LSM trees make object storage the primary store and are the natural choice at terabyte scale; on a one-gigabyte machine whose working set fits on disk, local serving wins by two orders of magnitude on read latency. Neon [5] separates compute from a distributed storage tier that plays the same role as HanuDB's bucket at far greater scale and cost; HanuDB's point-in-time restore is a small approximation of Neon's branching.

Storage engines. The document store is `lsm_tree`, the log-structured merge core of the `fjall` key-value engine [6], used without `fjall`'s own journal. The index is `tantivy` [7], a Lucene-derived search library whose JSON fields give a schemaless index with per-path fast columns. Both are pure Rust, compile quickly and have small memory footprints, which mattered as much as their throughput.

Write-ahead logging and group commit. The group-commit `fsync` loop follows the standard design [8]; the contribution here is in the file handling: a preallocated, zero-filled log written positionally so that `fdatsync` never flushes metadata, reused without truncation, with a decoder that tolerates stale, torn and zeroed tails.

Performance engineering by controlled experiment. A fixed metric, a frozen benchmark and a keep-or-discard rule are standard practice in empirical tuning; HanuDB applies that discipline to a storage engine, where correctness gates (recovery, byte-identical documents, working search) and resource gates (memory, bucket operations) must accompany the score, and records every discarded change alongside the kept ones.

3 System Design

3.1 Overview

HanuDB is one process (Figure 1). An HTTP server accepts JSON documents into named collections. Each write is appended to a write-ahead log, applied to the document store and handed to an indexer thread. A sealer thread periodically compresses the pending log into a segment object and uploads it to one or two buckets; a snapshot task periodically packs the on-disk engine state into a snapshot object. A fresh machine restores itself from the newest snapshot and the segments after it. Reads are served entirely from local state.

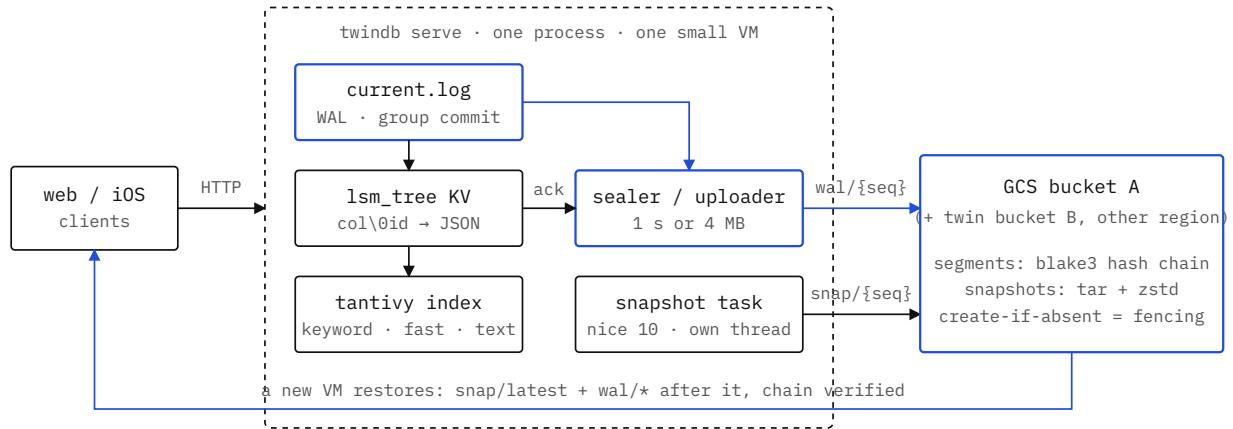


Figure 1. HanuDB architecture. Solid cobalt edges carry the durability path: a write is acknowledged as *local* after the group-commit `fdatsync` of the log and as *bucket* once its segment is in every configured bucket. Reads never leave the machine.

3.2 The write path and durability levels

A request body is parsed once. Under a short lock the engine assigns a monotonically increasing transaction number, encodes the record (transaction, kind, collection, identifier, document bytes) and writes it at the next offset of the open log file, then applies the document to the store's memtable. The record is passed to a dedicated indexer thread through a bounded channel. The handler then waits on a watch channel for the durability level the client requested:

- **none:** acknowledged after the in-memory apply; the log is written but not synced.
- **local** (default): acknowledged after the next group commit. One `fsync` loop wakes on demand and issues a single `fdatsync` for every writer waiting at that moment. This level survives a process crash.
- **bucket:** acknowledged when the segment containing the record is in the bucket, or in both buckets in twin mode. This level survives the loss of the machine.

The log file is preallocated and zero-filled once, and records are written positionally rather than appended, so the file's size and extent map never change and `fdatsync` flushes data blocks only. After each seal the same file is reused from offset zero without truncation. Recovery tolerates both properties: the decoder stops at the first zeroed header, torn record or transaction discontinuity, and discards records whose transaction number lies below the next transaction of the already-sealed segments. On the harness these two changes together yielded 10% more writes per second and 15% fewer bytes written to disk (§5.3). An alternative using `fallocate` was measured and rejected: `ext4` converts each unwritten extent on first write, which returns metadata to every `fdatsync` and cost 19% of write throughput.

3.3 One log, two watermarks

The single most consequential design change was the removal of a second write-ahead log. The `fjall` engine journals every insert before its memtable, so with HanuDB's own log in front of it each docu-

ment reached the disk three times before its first compaction. HanuDB now uses `lsm_tree` directly and the document store keeps no journal: its memtable is lost in a crash and rebuilt from HanuDB's log.

Correctness rests on two durability watermarks. The store records `kv_flushed_txn`, the newest transaction whose documents reside in a flushed table. The index records `index_txn`, the newest committed transaction, stored inside tantivy's own commit payload so that it is atomic with the commit. On boot the engine replays the log past each engine's own watermark, so each side receives exactly the records it lacks: at most one memtable's worth for the store and one commit interval's worth for the index. The floor below which open-log records are treated as stale follows the smaller of the two. Snapshots and clean shutdown flush the memtable first. The change reduced disk writes by 20%, resident memory by 80 MB and produced the best benchmark score of the study (§5.1).

A subtle failure mode was found by the correctness suite shortly after this change and is worth recording. For a few builds the staleness floor followed the index watermark alone, which after a forced kill discarded records the store had not yet flushed. A test that kills the process and verifies every acknowledged document caught it once in three runs; the test now forces the vulnerable window open on every run by waiting for an index commit before the kill. The general rule is that every "how far have we got" quantity in a system with several durability watermarks must follow the minimum.

3.4 *Segments and the hash chain*

Every second, or every 4 MB, the sealer compresses the pending records with `zstd` and writes one segment object named by its sequence number. The segment header carries a magic and version, the sequence, the first transaction, the record count, a timestamp and the blake3 hash of the previous segment; the object ends with the blake3 hash of everything before it. A reader verifies the chain from the snapshot forward and rejects any gap, reorder or corruption.

Segment objects are created with the bucket's create-if-absent precondition. A second writer for the same prefix therefore collides on its first upload and fences itself by exiting with a distinct status. This rule, and not any lease or clock, is the system's safety property against split brain; the lease described in §3.7 provides only liveness.

Cost follows directly from the cadence. One segment is one class-A bucket operation. With a one-second cadence the worst case is one operation per second of write activity and idle costs nothing; reads never touch the bucket. The benchmark measures 0.13 class-A operations per thousand writes.

3.5 *Physical snapshots and recovery*

A logical snapshot, meaning every live document re-serialized, would make recovery time proportional to data size because every document would have to be re-indexed. HanuDB instead takes physical snapshots of the engine directories. Every 200 segments, and only when the log since the last snapshot is at least half the size of the on-disk state, the engine commits the index, flushes the memtable and, under the write lock, hard-links the immutable index and store files into a staging directory; this takes milliseconds. Outside the lock, on its own task and on a thread niced to 10, it tars

and zstd-compresses the staging directory into a file, syncing every 32 MB, and uploads it as a multi-part object. Each tar entry is cut at the size recorded in its header, because a hard-linked table that the flush thread is still writing can grow between listing and copying.

A rebuild downloads the newest snapshot to a file, extracts it entry by entry with the same periodic sync, opens both engines on the result and replays only the segments after the snapshot's sequence. Retention deletes snapshots older than the third-newest and every segment at or below the oldest retained snapshot, so the bucket holds about three snapshots and at most a few hundred segments. Point-in-time restore selects the newest snapshot at or before a requested instant and replays segments up to it into a fresh directory without writing to the bucket.

3.6 Index and query execution

The index schema is dynamic in the sense of Elasticsearch's dynamic mapping. Every field whose value is at most 64 characters is indexed as an exact keyword and a fast column, which serve filters, sort and aggregations; longer strings are not indexed as keywords, following Elasticsearch's `ignore_above` rule. All string values in a document are concatenated into one text field per collection, which serves free-text search. Field paths are namespaced by collection so that no per-query collection term is needed. A Mongo-style filter language with nested paths, ranges, set membership, boolean combinators and negation compiles to tantivy queries.

Query execution relies on four caches, each keyed by segment identifier and safe because tantivy segments are immutable. (i) Equality filters become per-segment bitsets built once from postings and applied inside a query wrapper, so that tantivy's block-max WAND pruning continues to operate; this is the filter cache of Elasticsearch. (ii) Sort columns are opened once per segment rather than per query. (iii) Document-store readers are cached per segment with a larger block cache than tantivy's default. (iv) A 150 000-entry LRU holds compact copies of hot documents; copies rather than zero-copy slices, because a slice pins the whole underlying block. A further rule matters for text queries: block-max WAND engages only for a bare term query or a pure union of terms, so single-clause boolean wrappers are peeled before execution. Section 5.2 attributes the read-path gain to each of these.

3.7 Twin buckets, followers and leases

In twin mode every segment and snapshot is written to two buckets in different regions in parallel, and a write is bucket-durable only when both hold it. A rebuild lists both buckets, reads from the first with fallback to the second, and re-mirrors one-sided objects in both directions. Cost is exactly twice the class-A operations.

A follower boots from the bucket like any node and then polls for the next segment every second, verifying the chain and applying it. It serves every read endpoint and answers writes with HTTP 421 so that a client falls back to the leader. With the lease option every node boots as a follower; the one that creates the lease object becomes the writer and renews it by compare-and-swap at a third of the time-to-live. A standby that observes an expired lease takes it by compare-and-swap and promotes itself in place. A leader that fails to renew fences itself and exits. Measured locally with a three-second time-to-live, takeover completed three seconds after a kill signal, twice in succession, with the chain veri-

fied. Under a three-minute mixed workload of 22 700 operations/s on the leader, a follower on its own core was at the leader's transaction and segment within one second of the load ending, with identical document counts (§5.6).

The API is completed by bearer-token authentication, CORS, body limits, idempotency keys stored as replicated documents inside the same transaction, read-your-writes through a transaction number that the client passes back to a search, Prometheus metrics, a chain-verifying audit command, an off-line dump, and a live export of zstd-compressed JSON lines that analytical engines such as DuckDB read directly from the bucket.

4 Experimental Method

The engine was developed through a fixed experimental protocol, written down before the first experiment. It specifies the objective, the parts of the tree that may and may not change, the admissibility gates and the acceptance rule, and it was followed without exception: one change per experiment, every run recorded. This section describes the method precisely enough that the reader can judge every number in §5.

4.1 *Workload and score*

The benchmark client was fixed for the whole study and never edited. Thirty-two concurrent HTTP clients each insert one document of roughly 450 bytes per request for 20 s at the *local* durability level, so that every request waits for a real `fdatsync`. Documents carry a user identifier drawn from 1 000 values, a kind, two tags from 50, a timestamp and a title and body drawn from a 3 000-word vocabulary with a skewed distribution. Thirty-two clients then read for 20 s: 50% get-by-identifier, 25% filtered find sorted by timestamp with a limit of 20, and 25% full-text search with a filter and a limit of 10. Finally the server wipes its local directory and rebuilds from the bucket; every write it had acknowledged as bucket-durable must return byte-identical on a sample, and search must work. The score is the geometric mean of writes per second and reads per second, so that neither side can be traded for the other.

4.2 *Harness*

The server runs in a Linux container pinned to one physical core with a 768 MB memory limit, as a stand-in for the target machine, on an Apple M4 Max host. The bucket is a local directory with simulated latency of 70 ms per PUT and 40 ms per GET, the same-region floor measured from a Google Compute Engine instance; the simulated latency and the bucket-operation metering were held fixed for the whole study. Each run additionally times a fixed hash computation on the server core; runs on an overloaded host (calibration above ~650 ms) are discarded and repeated.

Two harness decisions were themselves the result of experiments, and they define the comparability of the results (Table 1). A CFS CPU quota throttles in 100 ms periods and produced 50 ms tail latencies and $\pm 8\%$ run-to-run noise, so from experiment 5 the container is pinned to a core instead. Running the client on the host added 0.6 ms per request through the container port proxy and

capped writes at 18 000/s, so from experiment 6 the client runs in a second container on the server's loopback. Scores across each boundary are not comparable, and the results log marks both.

Table 1. Harness generations. Scores are comparable only within a generation.

Experiments	CPU limit	Client location	Effect
1–4	CFS quota (--cpus=1)	host, via port proxy	50 ms p99 stalls from quota periods; ±8% noise
5	pinned core (---cpuset-cpus)	host, via port proxy	p99 6.6 ms; writes capped near 18 000/s by the proxy
6–108	pinned core	container on the server loopback	the reference harness for every result in this paper

4.3 Gates and acceptance rule

A change is inadmissible, with a score of zero, if any gate fails: the rebuild must succeed and verify; peak resident memory must not exceed 600 MB; class-A bucket operations must not exceed five per thousand writes; there must be no request errors; find and search must return hits. Tail latency is reported rather than gated, but a change that doubles a 99th-percentile latency for a marginal gain is rejected.

Identical builds scored 27 500 and 32 300 in consecutive runs, because the read phase runs over whatever the write phase produced and in whatever merge state the index is in when reads begin (§5.5 quantifies the band). Every candidate is therefore run twice and its mean compared with the baseline's mean; a keep requires a gain above 3%, or equal speed with less code or less memory. Simplicity is an explicit criterion of the program: a 1% gain for a hundred lines is a discard, and an equal score with fewer lines is a keep. Product features and safety changes are held to a different standard: their requirement is that the score stays within the noise band.

Each accepted change is stored as a numbered patch together with the benchmark log that justified it, and every run, accepted or not, is one line of a tab-separated results log with its score, throughput, tail latencies, peak memory, bucket cost and outcome. The full lineage of the engine is therefore reproducible from the first draft.

4.4 Correctness and long-running checks

Before every keep a native correctness suite must pass. It grew from 41 to 59 checks over the study and covers authentication, create-read-update-delete, filters, text search, sort, aggregations, idempotency, read-your-writes, restart after a kill signal, rebuild from the bucket, loss of one twin bucket, snapshots, compaction, chain verification, resynchronization, dump and TLS. A change that fails it is discarded regardless of score.

Roughly every twenty experiments we also ran checks that a 20 s benchmark cannot express: a scale run that writes about 2.5 million documents and rebuilds; a ten-minute soak that records resident, anonymous and dirty memory every ten seconds and then performs the crash drill; an update-heavy

variant with 40% updates and 10% deletes and a verified rebuild; and a leader-plus-follower run. Thirteen of the long-running checks failed during the study, each for a different reason (Table 7), and each failure became a design change.

4.5 Process discipline

One failure of the process itself is reported because it affects the early results. A shell error in the keep script meant that for experiments 9 through 13 the baseline snapshot never advanced, a discard reset the tree to the first draft, and three experiments measured the wrong program. The affected runs are marked invalid in the log, the kept patches were replayed (experiment 14, whose score matches experiment 8), and the scripts now verify their own result and fail loudly. Every keep since prints the patch file it wrote.

5 Results

5.1 Overall trajectory

Figure 2 plots the score of every kept experiment and Table 2 lists the milestones. The curve has three regimes. Experiments 1 through 6 fixed the harness and the memory model; experiments 8 through 53 removed per-query work from the read path, each step being the elimination of one cost; experiments 57 through 108 added product features and safety mechanisms whose requirement was to keep the score, not to raise it, with two exceptions, the allocator change (87) and the removal of the second journal (91).

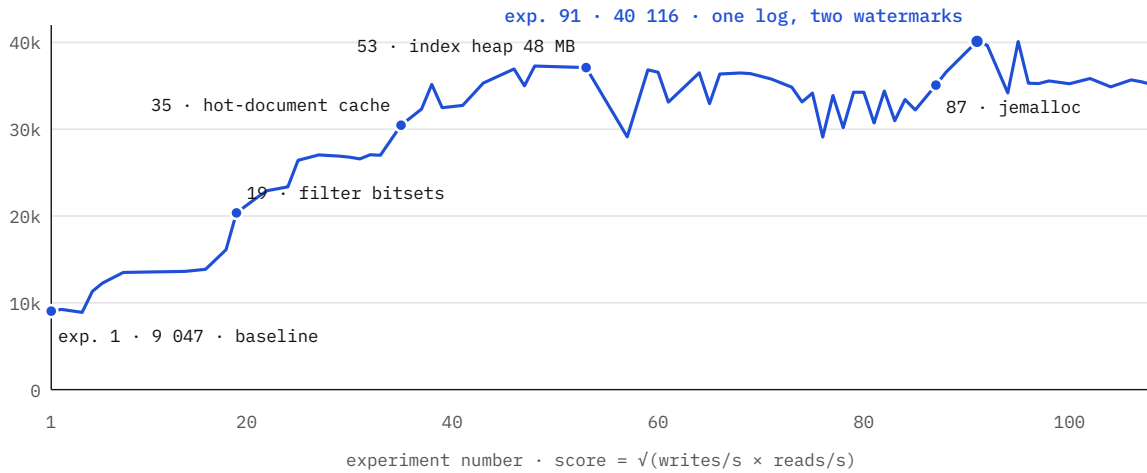


Figure 2. Benchmark score of the 68 kept experiments that were scored, on the one-core, 768 MB harness. Dips inside the plateau after experiment 53 are the read-phase variance caused by index merge state at the start of the read phase (§5.5), not regressions. Experiment 78 is plotted with its valid re-run (78b); its first run coincided with stray containers on the host and is marked invalid in the log.

Table 2. Milestone experiments on the harness. Throughput in operations per second; p99 in milliseconds; RSS is peak resident set size in MB. Dashes are runs whose log did not record that percentile.

Exp.	Score	Writes/s	Write p99	Reads/s	Read p99	RSS	Change
1	9 047	8 747	50.8	9 357	52.0	317	baseline under a CPU quota
5	11 322	18 516	6.6	6 924	11.1	270	pinned core; keyword ignore_above; no delete for fresh ids; index heap 32 MB
6	12 273	25 421	6.3	5 926	13.8	309	loopback client (same code as 5)
16	13 852	28 921	5.7	6 634	12.4	280	one text field per collection
19	20 354	28 932	6.0	14 320	5.4	294	cached per-segment filter bitsets
25	26 425	28 941	5.8	24 127	3.7	276	bare term queries; block-max WAND
38	35 170	27 376	–	45 182	–	310	150 k-entry hot-document cache
46	36 931	29 263	5.9	46 609	–	295	preallocated, zero-filled log
53	37 100	29 688	5.9	46 364	1.0	305	index heap 48 MB
87	35 081	34 283	4.6	35 900	2.2	444	jemalloc
91	40 116	34 821	4.7	46 218	1.1	355	lsm_tree directly; no second journal
108	35 139	34 732	–	35 550	–	378	final build (within the variance band)

Writes and reads improved for different reasons and are analysed separately below. Profiling of the final build attributes 18% of write-phase samples to the kernel, roughly 20% to tantivy indexing and 3% to compression; the read phase is dominated by socket wake-ups (6.8%), per-segment term lookups (5.4%), block-max WAND evaluation (4.2%) and integer decoding (4.1%).

5.2 The read path, one cost at a time

Reads rose from 6 634/s to 46 218/s across eleven kept experiments (Figure 3, Table 3). Each step removed one unit of per-query work, and the size of each step is an attribution of read cost to that work. The three largest were the filter bitset cache (+61%), the hot-document cache (+26% at 50 000 entries, +33% more at 150 000) and the unwrapping of single-clause boolean queries so that block-max WAND pruning engages (+30%).

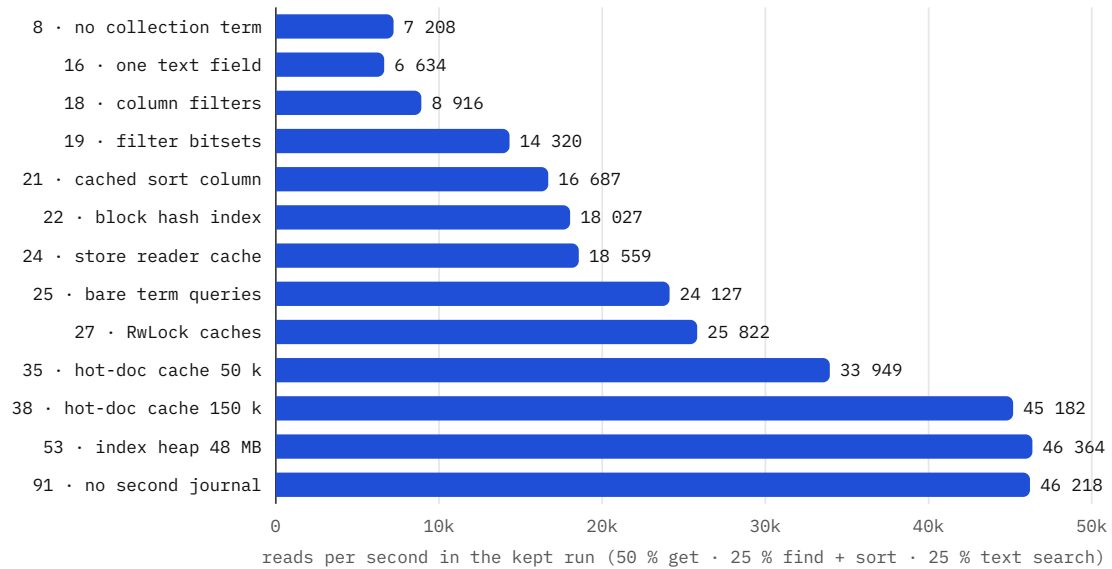


Figure 3. Reads per second on the harness after each read-path experiment. The write phase stayed between 27 000 and 29 000 writes/s throughout, so the read phase in every run searched a state of similar size. Experiment 16 lowered reads only because it made writes 16% faster, so the read phase ran over 16% more documents.

Table 3. Read-path experiments. Reads/s is the value of the kept run; Δ is relative to the previous kept experiment.

Exp.	Per-query work removed	Mechanism	Reads/s	Δ
8	a MUST term on the collection in every query (70% of read CPU in the posting intersection)	index field paths namespaced by collection	7 208	+22%
16	indexing every string twice (per field and joined)	only the joined text field per collection	6 634	-8%*
18	posting-list intersection for string equality and \$in filters under a text clause	filter evaluated on the fast column in the collector	8 916	+34%
19	rebuilding the filter per query	per-segment cached bitsets inside a Query wrapper (pruning kept)	14 320	+61%
21	opening the sort column per segment per query	cached Column<u64> handle per segment	16 687	+17%
22	binary search within a 4 KB store block per point read	hash index inside data blocks	18 027	+8%
24	store-reader block cache thrashing on large segments	4 096-block cache per segment reader	18 559	+3%
25	scoring every match of a text query	single-clause booleans unwrapped so block-max WAND engages	24 127	+30%
27	a mutex acquisition per segment per hit (9% of read CPU)	RwLock cache maps; one store lookup per segment per query	25 822	+7%
35	a store lookup for every get	50 000-entry LRU of hot documents	33 949	+26%
37–38	cache entries pinning a whole 4 KB block (RSS gate failed at 150 000 entries)	compact copies; 150 000 entries	45 182	+33%
53	many small index segments	index heap 32 → 48 MB	46 364	+3%

* Experiment 16 was kept for its write-side gain (+16%) and a 12% lower rebuild peak; its read figure reflects a larger state, not a slower read.

5.3 The write path and bytes written to disk

Writes are bound by the group-commit `fdatsync`, measured at 0.55 ms on this disk, and by HTTP framing on the single core; they rose in three steps and were otherwise flat. The steps were the harness correction (5–6), the log-file handling (46) and the allocator (87). Every attempt to free CPU elsewhere on the write path changed nothing (Table 5), which is the experimental confirmation that the path is bound by the `fsync` loop and request framing rather than by document processing.

Bytes written to the local disk are a separate objective, because the target machine's disk sustains a few dozen IOPS, and were measured from experiment 66 onward through `/proc/self/io` before the crash drill of each run. Figure 4 and Table 4 give the sequence. The log-file reuse, a leveled compaction with a larger level-0 threshold, larger memtables and the removal of the second journal together reduced disk writes from 4.54 MB to 2.06 MB per thousand documents with no loss of

throughput; an optional LZ4 compression of every store level reaches 1.76 MB at a 9% read cost and is enabled on the deployed machine.

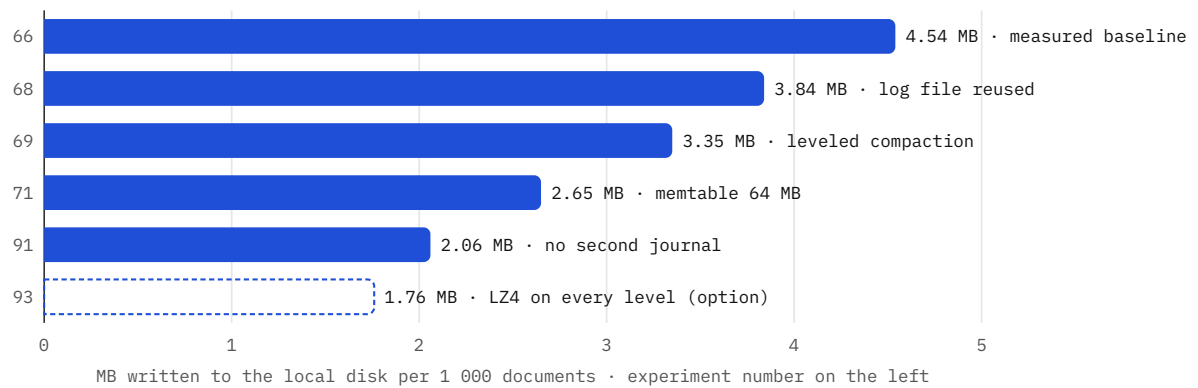


Figure 4. Disk bytes written per thousand documents on the harness after each disk-oriented experiment. The dashed bar is an option (`--kv-compress all`) that is not the default because it costs 9% of reads on the harness; on the free-tier disk the bytes matter more.

Table 4. Write-path and disk-bytes experiments. Writes/s are the kept run (two-run means where the log records them).

Exp.	Change	Writes/s	MB per 1 000 docs	Outcome
46	preallocated, zero-filled log; positional writes; data-only fdatsync	27 000 → 29 300	–	kept (+10% writes, p50 0.74 → 0.69 ms)
66	disk-write metering added	28 100	4.54	baseline for this series
67	fallocate instead of zero-fill	22 800	3.36	discarded (–19% writes: ext4 extent conversion returns metadata to fdatsync)
68	log file reused from offset 0 after each seal	28 400	3.84	kept (–15% bytes)
69	leveled compaction, L0 threshold 8, 128 MB tables	28 500	3.35	kept (–13%)
70	LZ4 on the fjournal	28 800	3.34	discarded (no change)
71	store memtable 16 → 64 MB	28 900	2.65	kept (–21%; +50–100 MB RSS)
87	jemalloc as the global allocator	34 283	–	kept (+22% writes)
91	lsm_tree directly; no second journal	34 821	2.06	kept (–20% bytes, –80 MB RSS, best score)
93	LZ4 on every store level	33 962	1.76	not default (reads –9%); exposed as <code>--kv-compress all</code> (94)
99	precise group-commit window (150 / 300 μ s spin)	35 100	–	discarded (unchanged; p99 4.7 → 6.0 ms)

Two measurements from the discarded write-side experiments fix the picture of where write time goes. During the write phase the server core is 92% busy with 2% I/O wait; the fdatsync itself takes 0.55–0.6 ms; and the kernel plus HTTP framing account for about 27% of samples. The write path is therefore CPU-bound on the request path with the indexer fitting into the gaps, not waiting on the disk, which is why a longer group-commit window (26, 99), batched log writes (42), skipping JSON re-serialization (62) and passing the parsed document to the indexer (89) all measured as no change.

5.4 Memory

Memory was gated, not scored, and most memory work was driven by the rebuild rather than the live path. Table 5 lists the changes that moved it.

Table 5. Memory experiments. "Rebuild" figures are the in-process crash drill unless stated.

Exp.	Change	Before → after
5	no delete_term for ids that do not exist yet (716 k pending deletes held ~1.1 GB)	rebuild peak 1.8 GB → 0.7 GB
5	keyword ignore_above 64 characters	index 226 → 106 MB; writes 11 700 → 18 000/s
5	index heap 64 → 32 MB (column writers live outside the budget)	rebuild peak 0.7 → 0.39 GB at ~8% indexing speed
29	store memtable 64 → 16 MB	peak RSS 368 → 221 MB (reverted to 64 MB at 71 for disk bytes)
37	hot-document cache stores compact copies, not block-pinning slices	peak RSS 309 → 261 MB at 50 k entries; 150 k entries became admissible
81	memory guard reads anonymous + dirty memory, not resident size	394 MB of real pressure where RSS reported 464 MB
86	glibc M_ARENA_MAX=1	discarded: a 3.8 M-document rebuild grew to 960 MB and never finished
87	jemalloc, 1 s decay, two arenas	in-process rebuild of 4.9 M documents settles at 245 MB anonymous (glibc: 585 MB; fresh process: 229 MB)
88	two store worker threads and a replay throttle	replay write buffer bounded at 73 MB (was unbounded: 542 MB after 1.2 M records)
91	no second journal	peak RSS 435 → 355 MB

5.5 *Negative results and the noise band*

Twenty-seven experiments were discarded. Table 6 lists those that tested a hypothesis rather than a typo. Three of them (7, 55, 90) also bound the noise: the block cache at 32, 64, 128 and 256 MB gave reads inside the same band at every size, and three runs of the 128 MB build alone spanned 34 200 to 45 900 reads/s. That band, 27 000 to 47 000 reads/s across the study, is the merge state of the index at the moment the read phase begins; the merge-policy experiments (48, 52, 58, 102) confirm it from the other side, since removing merges halves reads while forcing them early costs writes without lifting the ceiling. The two-run acceptance rule of §4.3 exists because of this band.

Table 6. Discarded experiments that tested a hypothesis. Percentages are relative to the baseline of the day.

Exp.	Hypothesis	Measured	Why discarded
3	sealing the segment on a spare thread frees the write lock	writes -13% (two runs)	on one core the thread only competes with request handling
7, 55, 90	a larger store block cache speeds reads	reads unchanged at 32-256 MB; +85 to +180 MB RSS	reads are not missing the cache
10, 50, 51	a different index commit interval (2 s, 250 ms)	2 s: writes +3% but reads -25%; 250 ms: -10% score	500 ms stays
11	the id as a fast column instead of the doc store	reads -19%	a 500 k-term dictionary lookup per hit is slower than a block read
15	mimalloc	writes +13%; rebuild peak +25%, container OOM	rejected on a mismeasured baseline; see 87
17	a single tokio worker	+1.2% (noise)	the target has two vCPUs
20, 23	uncompressed or differently blocked doc store	+1% (noise); -6%	LZ4 decode is not the cost; larger blocks are
26	a 200 μ s group-commit window batches more writes per fsync	writes -44%	the runtime timer resolves to ~1 ms and adds it to every write
28	store the JSON in the search index so hits skip the doc store	reads +15%, writes -20%, RSS 368 $\rightarrow$ 507 MB	memory gate; memory is the currency on 1 GB
34, 40	TCP_NODELAY; a hand-rolled HTTP/1 loop	no change	the framework is not the cost
42	batched log writes, one syscall per group commit	no change	writes are fsync-bound
52, 58, 102	eager merging; an idle-time merge kick	no gain; read p99 1.0 $\rightarrow$ 3.8 ms; writes -10%	the merge steals the core; kept only as a knob
54	index heap 64 MB	same as 48 MB	48 MB uses less rebuild memory
62, 89	skip JSON re-serialization; hand the parsed document to the indexer	no change	not on the critical path; more plumbing
67	fallocate the log instead of zero-filling it	bytes -26%, writes -19%	ext4 extent conversion; superseded by file reuse (68)
70	LZ4 on the store's journal	bytes unchanged	superseded by removing the journal (91)
86	one malloc arena reduces residue	residue 5 MB, but a rebuild grew to 960 MB	one heap fragments worse than per-thread arenas
93	LZ4 on every store level	bytes -15%, reads -9%	not the default; a knob for IOPS-bound disks
99	a precise spin-wait before each fdatsync	writes unchanged, p99 4.7 $\rightarrow$ 6.0 ms	the request path is CPU-bound, not waiting on the disk

5.6 *Scale, soaks and what they broke*

Table 8 summarizes the long-running checks that passed. The ten-minute soak wrote 14 million documents with zero errors while seven snapshots of up to 4.5 GB were packed in the background; anonymous memory stayed within a 100 MB band for the whole run. In the subsequent crash drill the 4.5 GB snapshot was extracted in 10.6 s and a 474 000-record tail replayed, 39.7 s in all, and every document returned byte-identical. Reads over 14 million documents, far beyond the page cache, fell to 1 478/s, which is the disk's random-read rate rather than the engine's. The update-heavy soak, with 40% updates and 10% deletes, exercised tantivy deletions, store tombstones and compaction; the live count matched exactly and memory stayed flat. A follower under a three-minute mixed load was at the leader's transaction within one second of the load ending.

Before these passes, thirteen long-running checks failed. Table 7 lists each one, the cause found, and the change that followed. They are the most informative measurements in the study: every failure was invisible to the 20 s benchmark, and every one produced a mechanism that the design now depends on.

Table 7. Long-running checks that failed, in order, with the root cause and the resulting change. All in the same one-core, 768 MB container.

Run	Failure	Root cause	Change
65	OOM at 768 MB during the rebuild	a 741 MB snapshot was downloaded into memory	snapshot streamed to a file and extracted from it
71d	the benchmark client died after ~3 min	the client kept every written document for the verification	client keeps every txn and a 1-in-64 document sample (72)
72b	OOM after ~4 min of writes	memtables piled up while the core packed a 2.4 GB snapshot every 15 s	memory guard; snapshots only when the log grew $\geq 50\%$ of the state (73)
73c	OOM at ~7 min while packing	the pack's own 3 GB of writes were dirty pages inside the cgroup	fdatasync every 32 MB while packing (74)
74b	OOM during the rebuild's extraction of a 2.7 GB snapshot	the same, on the extraction path	streaming extraction with periodic fdatasync (75)
75b	OOM at ~5.5 min with the guard active	merges of multi-GB index segments	segments above 2 M documents are never merged (76)
76b	OOM during the rebuild's replay	recovery bypassed the memory guard	the replay pauses while memory is over the limit (77)
77b	213 786 client timeouts; OOM after a pack	the pack ran inline in the seal loop (no seals, uploads or guard for 32 s); the guard read RSS, which counts clean mapped pages; the local-bucket upload never synced	guard reads anonymous + dirty; synced local uploads (81); pack at nice 10 (82); pack on its own task (83)
82b	OOM after 184 s, right after a 955 MB pack	the pack and upload were still inline, so the guard ran 13 times in 184 s	the snapshot runs on its own tokio task, one at a time (83)
83b	the rebuild hung	the rebuilt engine's own footprint was above the limit with nothing left to drain, and the pause had no bound	the pause is bounded at 10 s per segment (84)
85b	OOM during the rebuild	memory grew ~5 MB per replayed segment, i.e. with volume replayed, not with the state	instrumented; allocator changed (87); cause found at 87c
87b-c	OOM during the rebuild after 5.5 M replayed documents	fjall rotated a full memtable through a try-send to a one-thread worker pool that was busy compacting; the message was dropped and the memtable grew ~600 B per replayed document; the write-buffer limit was stored, never enforced	two worker threads; a replay throttle on the write buffer (88). 88b passed end to end.
91b	the rebuild found 0 documents	the crash hit while a 4.4 GB snapshot was half-copied into the local test bucket and the rebuild	uploads staged under a hidden prefix; floor follows the minimum watermark (92).

Run	Failure	Root cause	Change
104	intermittent rebuild failure under the update-heavy load	picked the partial file; separately, the staleness floor followed the index watermark alone (§3.3)	92b passed: 14.0 M documents.
		tar copied a hard-linked table that the flush thread was still growing, past its recorded size	every tar entry is cut at its header size (104). 104b passed.

Table 8. Long-running checks that passed, on the same one-core, 768 MB container. Anonymous memory is the resident set minus clean mapped index pages.

Run	Documents	Writes/s	Anon. memory	Snapshot	Rebuild	Outcome
scale, 120 s (65)	2.45 M	20 412	681 MB peak RSS	741 MB	10.5 s	verified
soak, 600 s (88b)	13.12 M	21 870	260–330 MB	1.9 GB	258 s	0 errors, 0 mismatches (7.3 M-record tail)
soak, 600 s (92b)	14.00 M	23 337	240–340 MB	4.5 GB	39.7 s	0 errors, 0 mismatches
update-heavy soak, 300 s (104b)	6.27 M ops	20 855	220–317 MB	1 GB	21.9 s	live count exact; 128/128 sampled documents at their last version
leader + follower, 180 s (105)	2.05 M live	22 659	~200 MB (follower)	—	—	follower at the leader's txn within 1 s; counts identical
soak + point-in-time restore (107b)	13.48 M	22 462	190–330 MB	2.9 GB	146 s	0 mismatches; restore to 390 s earlier in 45 s (9.9 M documents)

5.7 A real free-tier machine

The same binary was deployed to an e2-micro in us-central1 with a real bucket and, in twin mode, a second bucket in us-east1. The benchmark client shared the machine's two shared vCPUs, so these figures include client cost. Each row is a separate run on a different day of the study; the build is noted.

Table 9. Measurements on a GCE e2-micro with real GCS buckets. Latencies are p50 / p99 in milliseconds.

Mode (build)	Writes/s	Write lat.	Reads/s	Read lat.	Class-A / 1k	Rebuild from GCS
single bucket, 10 s (33)	4 188	6.9 / 21.9	9 093	3.5 / 7.2	0.21	41 922 docs in 2.6 s
twin buckets, 10 s (39)	3 360	8.1 / –	11 554	–	0.42	33 637 docs in 8.2 s
twin, zero-filled log (46)	3 699	5.4 / 85.6	10 677	–	–	–
twin, snapshot staging off the lock (47)	5 543	4.6 / 21.6	7 472	–	–	–
4 clients, twin (71)	2 073	1.7 / 4.8	4 868	0.3 / 1.2	–	–
sustained, 180 s (65)	~1 025	– / 255	~89	– / 1 660	0.40	95 s
after a 2-minute cooldown (65)	4 386	–	1 575	– / 232	–	–

Two rows show engine changes landing on the real machine: the zero-filled log cut the twin-mode write median from 8.1 to 5.4 ms, and moving snapshot staging off the write lock cut the write p99 from 85.6 to 21.6 ms. The four-client row is what an application sees: sub-millisecond gets, finds and searches with a 1.2 ms read p99.

The sustained row is the honest limit of the free tier and, in our view, the most useful measurement in this paper. An e2-micro bursts to two vCPUs but sustains a quarter of one; once the burst credit is spent, the engine's background work (merges, compaction, snapshot packing, uploads) competes with request handling for that quarter core. The second limit is the disk. The engine wrote 3.66 MB to disk per thousand documents on the VM at the time (2.06 MB on the final build, 1.76 MB with store compression), so a few thousand writes per second exceed what a 30 GB standard persistent disk sustains, and once the working set outgrows the page cache, reads queue on the same disk: after the cooldown, writes recovered to 4 386/s but reads stayed at 1 575/s with 65% of CPU time in I/O wait and no steal. Bursty traffic is served well; sustained ingestion or a hot set above roughly 400 MB requires the next machine size and a balanced disk, together about fifteen dollars per month.

5.8 Cost

After the first day of operation, during which every measurement above was run against the deployed machine, each bucket held 211 MB and about 150 class-A operations had been billed: roughly one cent, with the machine and its disk free. Under a nominal load of ten thousand writes per day arriving in bursts, segment cadence bounds the bill below ten cents per month. An external probe from a Cloud Run service every five minutes, which touches about two segments per bucket per probe, adds about fifteen cents per month with a twin.

6 Discussion: What the Experiments Found

The negative results and the soak failures shaped the engine as much as the accepted changes. We list the findings that reversed an assumption or changed the design, with the experiment that established each.

- **On one core every background thread competes with request handling** (3, 52, 58, 102). Moving segment sealing off the write lock onto a spare thread reduced writes by 13%. Blocking writers for microseconds is free; additional CPU is not. Eager index merging and an idle-time merge trigger both lost for the same reason, even though merges themselves are worth their cost: without them reads halve (48).
- **Two write-ahead logs are one too many** (91). Removing the store's journal in favour of per-engine watermarks (§3.3) was the largest single improvement in throughput, disk writes and memory.
- **Bytes written matter more than CPU on a slow disk** (46, 67–71, 91, 93). The zero-filled positional log, the reuse of the log file, a leveled compaction with a larger level-0 threshold, larger memtables and the store compression option together reduced disk writes per thousand documents from 4.54 MB to 1.76 MB. Skipping JSON re-serialization and the indexer's re-parse freed CPU but changed nothing (62, 89), because the request path, not document processing, is the bottleneck.
- **Resident set size is the wrong quantity for an out-of-memory guard** (77b, 81). It counts clean memory-mapped index pages that the kernel reclaims under pressure. What kills a process inside a control group is anonymous memory plus dirty page cache, and dirty pages count against the limit, so every path that writes gigabytes must sync periodically. Three separate such paths were found, each by a soak being killed: snapshot packing (73c), snapshot extraction (74b) and the upload to a local bucket (77b).
- **Never run a minute-long job inline in a control loop** (77b, 82b, 83). Snapshot packing originally ran in the sealer loop, so for up to 130 s per snapshot nothing was uploaded, the memory guard did not execute and clients timed out. Moving it to its own task turned 213 786 client timeouts into zero, and on one core the pack must also run at reduced priority (82).
- **Read the storage engine's worker model before trusting its limits** (87c, 88). The ffall engine sized its worker pool at one thread on the target machine and ran compactions on it, so a long compaction blocked every memtable rotation and the active memtable grew without bound at replay speed. This was the cause of every rebuild out-of-memory failure in the soaks. Two workers and a replay throttle bounded the write buffer at 73 MB.
- **The allocator decides how much of a rebuild you keep paying for** (15, 85c, 86, 87). With glibc malloc, an in-process rebuild of 4.9 million documents settled at 585 MB of anonymous memory where a fresh process needs 229 MB, because freed pages remained in per-thread arenas. jemalloc with a one-second decay settled at 245 MB and improved write throughput by 22%. An earlier trial of mimalloc had been rejected only because its rebuild peak was compared with glibc's understated live figure; the measurement, not the allocator, was wrong.

- **Block-max WAND has preconditions** (25). tantivy prunes top-k evaluation only for a bare term query or a pure union of terms. Wrapping the text clause in a boolean query with one MUST clause silently scores every match; for common terms that was the whole search cost. Peeling single-clause booleans gave 30% more reads.
- **Filter caches are worth building twice** (18, 19). Evaluating an equality filter on a fast column instead of intersecting postings was worth 34%; a cached per-segment bitset applied inside the query, so that pruning stays active, was worth a further 61%.
- **Copy what you cache** (36, 37). A zero-copy slice over a store block, once placed in the document cache, kept the entire 4 KB block resident; 150 000 such entries exceeded the memory gate. Compact copies halved the cache's footprint.
- **Fewer, larger index segments are worth their merge CPU, but the timing is uncontrollable** (48, 90). The run-to-run read variance of the benchmark, from 27 000 to 47 000 reads/s, is entirely the merge state of the index when the read phase begins. The acceptance rule of §4.3 exists because of this.
- **With several durability watermarks, every derived quantity must follow the minimum** (91b, 92). The one data-loss bug of the study was a staleness floor that followed the wrong watermark, and it was found by a correctness check, not by the benchmark.
- **Fix the harness before the engine** (5, 6). A CPU quota instead of a pinned core, and a client behind a port proxy, added more distortion than most engine changes were worth.

7 Limitations

HanuDB is a single-writer system; its write ceiling is one core and one disk's fsync rate. It offers atomic multi-document batches but no interactive transactions across requests, no secondary indexes beyond the dynamic keyword and fast columns, no query planner and no sharding. A hot set larger than memory degrades to the disk's random-read rate. Field-scoped full-text search is not indexed, only the joined text. Automatic failover requires a second machine, which is not free. The harness core is faster than the target machine's, so harness figures are an upper bound; the free-tier measurements in §5.7 are the numbers an application should plan against. The scored experiments measured a single insert-heavy workload; the update-heavy soak exists precisely because the scored benchmark does not exercise deletions. Finally, the harness bucket is a local directory: it reproduces the latency and the operation count of object storage but not its failure modes, and one of the soak failures (91b) was an artifact of that difference.

8 Conclusion

A document database can keep local-disk latency while making an object bucket its only durable home, on the smallest machine a cloud offers for free, if three things hold: one write-ahead log serves every engine, with per-engine watermarks in place of per-engine journals; the log is shipped as hash-chained segments whose creation is itself the fencing mechanism; and recovery is proportional to the log tail through physical snapshots. A disciplined series of single-change experiments with a fixed workload, hard gates and a two-run acceptance rule was sufficient to take such an engine from a first

draft to 34 800 writes/s and 46 200 reads/s on one core, and its failures under long-running load taught more than its successes: on a shared core, the engine's own background work is its principal adversary, and memory must be accounted the way the kernel accounts it. The implementation, the benchmark, the correctness suite, the experiment log with every kept and discarded change, and the deployment scripts accompany this paper.

References

1. SlateDB contributors. SlateDB: an embedded key-value store built on object storage. <https://slatedb.io>, 2024–2026.
2. B. Johnson. Litestream: streaming replication for SQLite. <https://litestream.io>, 2021–2026.
3. Madhu. twinlog: a two-bucket write-ahead log protocol; measurements of GCS latency from a GCE instance and a comparison with bucket-native engines. Technical notes, September 2026.
4. Turso contributors. Turso database and libSQL. <https://turso.tech>, 2023–2026.
5. Neon. Neon: serverless Postgres with separated storage. <https://neon.tech>, 2022–2026.
6. M. Fenner et al. fjall and lsm_tree: an LSM-based embedded key-value storage engine in Rust. <https://github.com/fjall-rs/fjall>, 2023–2026.
7. P. Masurel et al. tantivy: a full-text search engine library in Rust. <https://github.com/quickwit-oss/tantivy>, 2016–2026.
8. D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker and D. A. Wood. Implementation techniques for main memory database systems. In *Proc. ACM SIGMOD*, pp. 1–8, 1984.
9. S. Ding and T. Suel. Faster top-k document retrieval using block-max indexes. In *Proc. ACM SIGIR*, pp. 993–1002, 2011.
10. P. O'Neil, E. Cheng, D. Gawlick and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33(4):351–385, 1996.

Preprint, revision 2. Every number in this paper is taken from the benchmark logs of kept experiments (results.tsv, experiments/), from the soak logs, or from the deployed machine's status endpoint; the experiment log lists every run, including the discarded ones. Source, benchmark, correctness suite and deployment scripts: <https://github.com/madhudream/hanudb>.