byron: Another Key-Value Store

Michael Almeida
CS265 Spring 2025
`mia330@g.harvard.edu`

1 Design Description

1.1 Introduction

byron is a high-performance, latch-free NoSQL key-value store implemented in Rust using a Log-Structured Merge Tree (LSM) architecture. The system is designed for write-intensive workloads by leveraging cache-conscious design and hardware-friendly storage layouts. Core supporting data structures include memtables, SSTables, Bloom filters, and fence pointers, while core algorithms handle tasks such as compaction, Bloom filter allocation, and on-disk storage formatting.

1.2 Technical Description

This section provides a detailed overview of byron's design considerations and technical architecture.

(A) On-Disk Data Layout

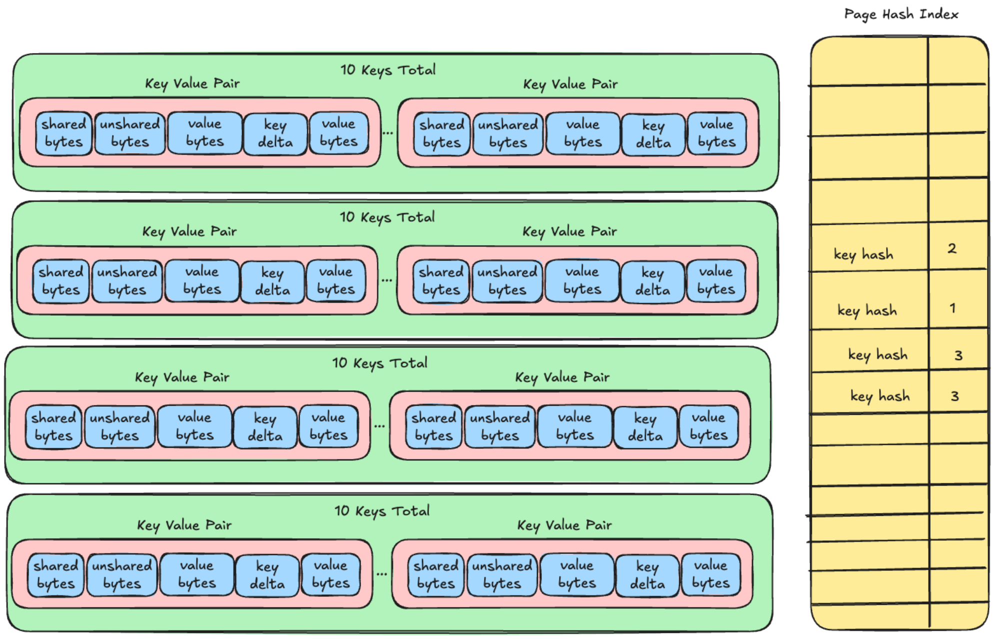


Figure 1: SSTable on-disk layout.

Goal Our overarching goal is to minimize disk I/O and CPU cache misses by optimizing the on-disk data layout. byron uses a Log-Structured Merge Tree (LSM) architecture, and compacts stored data frequently to reduce fragmentation and improve read performance. A key component of the LSM tree is the SSTable, which is an immutable on-disk file that stores sorted key-value pairs. The design of the SSTable is critical to achieving high performance in both read and write operations. The SSTable is designed to be compact and efficient, with a focus on minimizing disk I/O and CPU cache misses. In our design, we align SSTable blocks to 4 KB SSD pages for minimal random accesses and high sequential I/O throughput, and use key compression to pack more entries per page.

Design Elements

Varint Encoding (Integer metadata) (lengths, sequence numbers) are stored in a compact varint format to save bytes.

Delta Encoding (Prefix Compression) Each key is recorded as the byte-suffix that differs from its predecessor, greatly reducing size when keys share prefixes.

Restart Points Every 10th key in a block is stored in full (not delta–encoded). These "restart" keys bound the cost of an in-block binary search.

Page Hash Index An in-memory array of each block's first key lets you binary-search the correct 4 KB page in $O(\log N)$ time, avoiding a full scan.

Rationale An SSTable is an immutable on-disk file of sorted key–value pairs with the above optimizations. Keys are serialized by first writing a varint length, then a byte-slice delta from the previous key. Within each 4 KB block, restart points ensure you never examine more than 9 other keys on average. The restart-pointer index (one key per block) lives in RAM, so most `GET`s need at most one disk read—and none if the key isn't present.

Together, prefix compression plus restart indexing typically save about 3 bytes per 32-byte key. In skewed key distributions (e.g. sequential timestamps), the stored suffix length approaches a small constant, making delta encoding extremely efficient when keys share long prefixes. Analysis in §**??** elucidates the performance of these optimizations under expected bytes saved.

Variable Integer Width Instead of representing fields in our keys as flat integers, we instead use Variable-width Integers– a format popularized by the Protobufs project. Each byte in a varint consists of a continuation bit and a 7 bit payload– the continuation bit being there to let us know whether this bit is a continuation of the previous byte.

```
continuation bit | 0 1 2 3 4 5 6
```

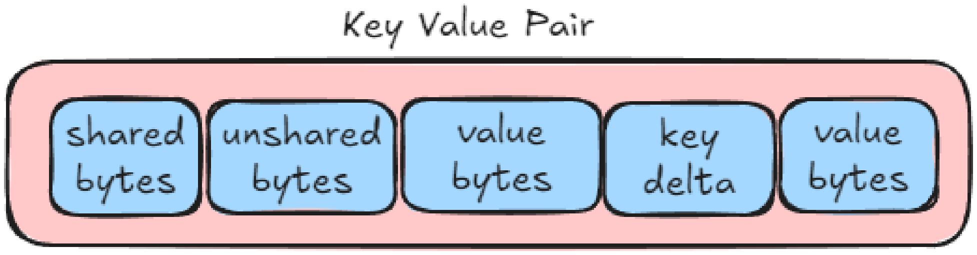


Figure 2: Key Value pair structure.

Delta Encoding

- `shared_bytes`: Shared prefix length with predecessor key.

- `unshared_bytes`: Length of key suffix.
- `value_bytes`: Length of the associated value.
- `key_delta`: Remaining suffix of the key.
- `value`: Actual stored value.

Delta encoding is a technique used to compress data by storing only the differences between consecutive values. In the context of key-value pairs, this means that instead of storing the entire key, we only store the part of the key that differs from the previous key. This is particularly effective when keys are similar or share common prefixes. For example, if we have two keys ‘key1‘ and ‘key2‘, instead of storing both keys in full, we can store the length of the shared prefix (in this case, ‘key‘) and then only store the differing part ‘1‘ and ‘2‘. This reduces the amount of data that needs to be stored and can lead to significant space savings, especially when dealing with large datasets with many similar keys.

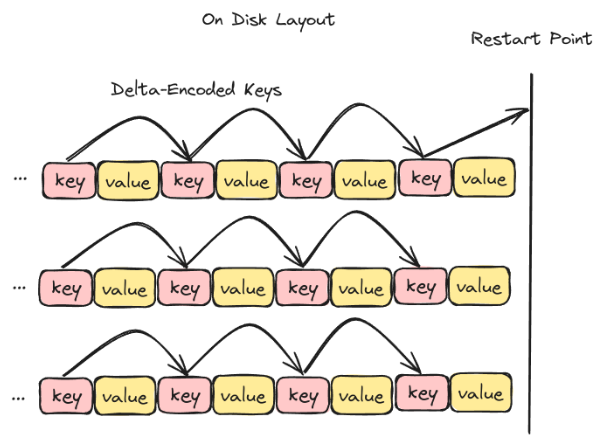


Figure 3: Restart Points.

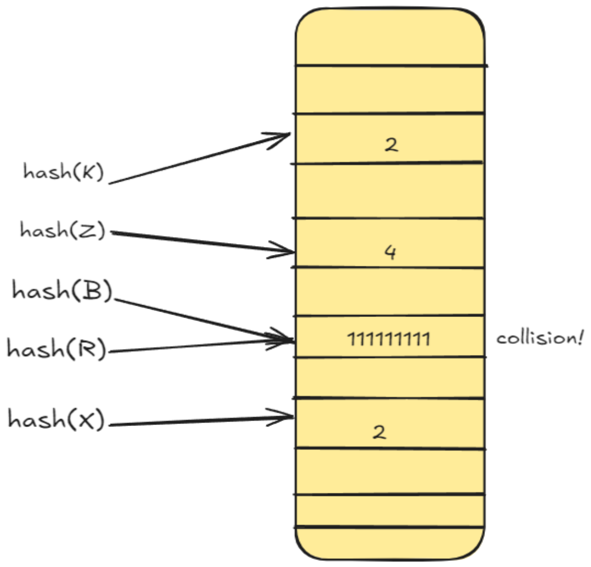


Figure 4: Page Hash Index.

Restart Points and Page Hash Index The draw back to using Delta Encoding is that in order to find a key, we must then linear search the entire block. This is undesirable, as now we incur a cost of O(n), where n is the block size. To combat this, we embed ‘Restart Points‘ in each block; and we do so at every 10th key.

Page Hash Index On their own, Restart points are saved as pointers at the end of a SStable in the footer. In order to beat Binary Searching these restart pointers, we need to do something else. This comes in the form of a Page Hash index. Simply put, a page hash index matches a keys hash with it’s restart point. Each entry has a maximum of 8 bits width, and points to a restart page. Implicitly, this means we can identify at most 256 restarts. The amount of slots is the number of keys. This is a very simple hash table, so collisions are possible. In the case of there being a collision, we cancel the slot and default to binary search.

SSTable Footer In the common literature, footer of an SSTable contains metadata about the table, including the size of the table, the number of keys, and the location of the restart points. This information is used to quickly locate and access the data in the SSTable. The footer is also compressed to save space. However, in byron, for simplicities sake, we store the page hash index and restart points in RAM.

(B) Memtable Data Structure Design

Original and Enhanced Approach

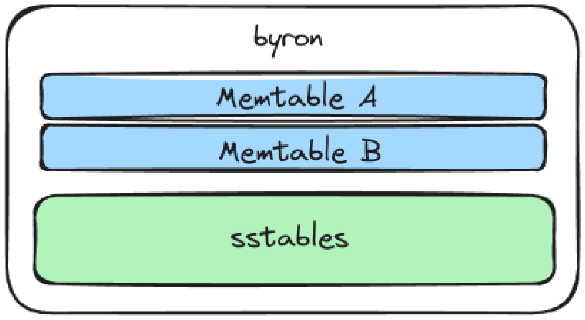


Figure 5: Double Buffered.

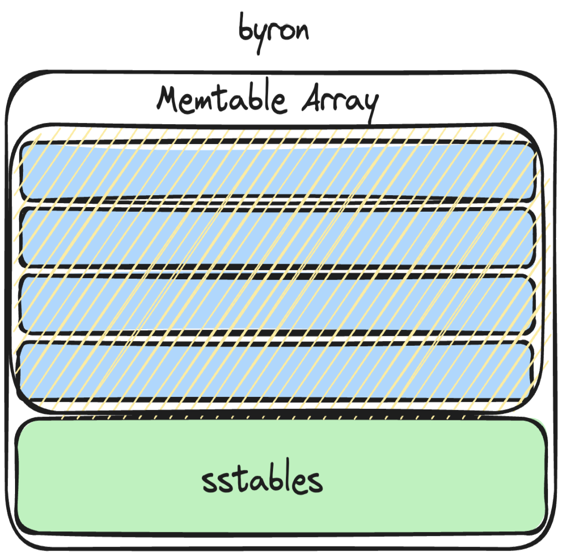


Figure 6: Skiplist array.

Goal The memtable serves as a fast, in-memory structure for buffering writes before they are flushed to disk. byron's memtable is implemented as a lock-free skiplist, which provides efficient insertion and lookup while maintaining key order. This design ensures high performance for both writes and reads in a concurrent environment.

Originally, byron used a double-buffered vector for the memtable. While simple, this approach had limitations in terms of concurrency and performance. The skiplist implementation addresses these issues by offering better cache locality, reduced contention between threads, and thread-safe operations. Additionally, the skiplist is sourced from the battle-tested 'crossbeam' library, which is optimized for performance and safety, allowing us to focus on higher-level optimizations.

Although the skiplist achieves $O(\log n)$ writes instead of the amortized $O(1)$ writes of the vector, it significantly improves over the vector's $O(n)$ worst-case performance. Furthermore, the skiplist supports concurrent writes and reads, enabling multiple threads to operate simultaneously without blocking. This lock-free design leads to better overall throughput, making it well-suited for a high-performance key-value store.

Original Approach: Double-Buffered Vector

Double buffers Two memtables—"active" and "immutable"—hold writes. Once the active buffer reaches 2 MB, it becomes immutable and flushes in the background while new writes go into the other buffer.

Lock-free writes Appends to the active vector are amortized $O(1)$ and require no locks for producers.

Linear scans for reads GETs search both buffers linearly; for typical memtable sizes, this overhead remains low.

Enhanced Approach: Lock-Free Skiplist

Crossbeam `SkipMap` Replaced backing vectors with a lock-free skiplist offering thread-safe $O(\log n)$ inserts and lookups.

Memtable Array Each memtable is a skiplist, and we maintain an array of memtables to allow for concurrent writes and reads. This allows us to have multiple active memtables, which can be flushed in parallel. Writes always hit memtable[0]

Built-in ordering Eliminates the extra sort/merge step on flush—entries stream out in sorted order.

Improved cache locality Skiplist nodes and allocators reduce cache-line bouncing; measured cache misses dropped from 10.69% (vector) to 4.99% (skiplist).

Flush Interface An inherent iterator on the skiplist streams sorted key–value pairs directly to a new SSTable file, enabling efficient sequential writes without additional merging. The iterator is designed to be lock-free, allowing multiple threads to read from the skiplist while it is being modified. This ensures that the flush process does not block other operations, maintaining high throughput for concurrent writes. There were a lot of challenges in parralellizing the flush process for the vector backed memtables– however, flushing the skiplists was a lot easier and, in practice, the memtable array never gets larger than 2 memtables.

Vector Backed

- **Time Complexity:**
  - Sorting and deduplication using a `BTreeMap` is $O(n \log n)$, where n is the number of key-value pairs.
  - Iterating through the `BTreeMap` is $O(n)$.
  - Total flush complexity: $O(n \log n)$.
- **Space Complexity:**
  - Two buffers: $O(n)$.
  - Temporary `BTreeMap`: $O(n)$.
  - Total: $O(n)$.
- **Expected Runtime:**
  - Runtime is primarily dominated by sorting, which, under our previous assumptions for the structure of keys, means that they will be very small and rather quick.
  - Disk flush operations themselves are asynchronous and minimize the impact on the main application performance.

crossbeam-backed Memtable

- **Time Complexity:**
  - **Writes/Reads:** Each `insert` or `get` is $O(\log n)$ in the skiplist.
  - **Flush:** Streaming entries in sorted order is $O(n)$ with no extra sorting.
- **Space Complexity:**
  - N buffers: $O(n)$. (Usually this is 2, but we can have more if we want to.)
  - Each skiplist node: $O(1)$.
  - Total: $O(n)$.
- **Expected Runtime:**
  - Runtime is primarily dominated by the skiplist operations, which are $O(\log n)$.
  - Disk flush operations themselves are asynchronous and minimize the impact on the main application performance.

(C) Bloom Filters & Fence Pointers in Memory

Goal The goal of the Bloom filter and fence pointer design is to minimize disk reads and enhance read performance. byron uses Bloom filters to probabilistically test key membership, skipping unnecessary SSTable reads, and fence pointers to index the first key of each block for efficient lookups. Together, they reduce disk I/O, improve cache efficiency, and accelerate read-heavy workloads by leveraging small, cache-friendly data structures.

Per-SSTable Filters Each SSTable has an in-memory Bloom filter. On a `GET`, we query the filter first—if it says "not present," we skip that SSTable entirely (no disk I/O).

Monkey-Inspired Allocation We distribute Bloom-filter bits across LSM levels proportional to data volume, giving larger lower-level tables more bits per key.

Performance Impact Tuning the total filter budget from 100 K bits to 10 M bits per SSTable improved `GET` throughput by 10%, with diminishing returns beyond 1 M bits.

Sparse In-Memory Index We store the first key of each 4 KB block in an array. A binary search on this "fence pointer" array locates the exact block in $O(\log B)$ time (where B is the number of blocks).

Compact & Fast At only a few bytes per block, the index comfortably fits in RAM. Combined with Bloom filters, most `GET`s incur at most one block read—and many incur none.

Cache Efficiency In large-scale tests, overall CPU cache miss rates stayed in the 3–8 % range, demonstrating the effectiveness of this two-tiered in-memory strategy.

(D) Compaction Strategy

Compaction Trigger & Merge Implementation Our overarching goal here is to balance read and write amplification by tuning when and how SSTables are merged. byron uses an expansion factor of $\varphi \approx 1.618$ (the golden ratio) between LSM levels to decide compaction points.

Size Ratio When level L_i grows to $\varphi \times |L_{i-1}|$, we compact a small number of tables from L_i into L_{i+1}.

Frequent, Small Jobs A lower ratio (vs. the typical $10\times$) yields more levels but smaller merges, smoothing write-amplification and avoiding long stalls.

Streaming Merge Open iterators on the chosen SSTables and use a min-heap to pull the next smallest key across inputs.

Time Complexity $O(n \log m)$ for n total entries across m tables (usually $m = 2$, making it nearly linear).

Space Efficiency Output is streamed directly to disk with only a small in-memory buffer, so peak memory stays $O(1)$ aside from the heap.

Amortized Levels With $\varphi < 2$, there are $O(\log N)$ levels, so inserts remain $O(\log N)$ overall.

Rationale Choosing $\varphi \approx 1.618$ spreads compaction work evenly:

- More levels → smaller, predictable compactions → steady write throughput.
- Extra levels have minimal read impact thanks to Bloom filters and fence pointers.
- Worst-case write-latency spikes are reduced compaacted to a larger size ratio.

1.3 Additional Optimizations

Beyond the core LSM design, we implemented two further performance enhancements:

(A) Hardware-Cache Conscious Coding

256-Byte Alignment Align hot data structures to 256-byte boundaries (multiple cache lines) to reduce cache-line contention and false sharing. In multithreaded tests, this yielded measurably lower CPU cache-miss rates.

4 KB Page Alignment Align all SSTable block writes to 4096-byte (4 KB) boundaries, matching OS page sizes and SSD erase-block sizes. This prevents cross-page I/O and maximizes sequential read/write throughput.

Great Cache Locality Empiracally, we found that the cache miss rate for our skiplist-backed memtable was about 4.99% (down from 10.69% with the vector-backed memtable). Moreover, on full workloads, our cache miss rate was about 3.41% for 1M GETs and 4.08% for 2.5M GETs, suggesting our design is cache-friendly and efficient in terms of memory access patterns.

These hardware-aware alignments plus multi-layered compression together boost runtime performance (fewer cache misses, faster I/O) and shrink the on

1.4 Challenges

(A) Ensuring Memory Safety Without `unsafe`

Context byron's on-disk format requires zero-copy parsing of byte-aligned structures (varints, key deltas, block headers) without ever using 'unsafe' Rust. Guaranteeing correctness across cache-line and page boundaries posed a major challenge.

Key Strategies

Layered Verification We built serialization/deserialization in stages (varints → key deltas → full blocks), unit-testing each layer independently to catch bugs early.

Extensive Test Suite Over 50 unit and property-based tests cover boundary cases (cache-line–sized keys, page-crossing values, malformed inputs).

Rust Type & Lifetime Safety On-disk buffers map to tight Rust structs; lifetimes ensure slices don't outlive their backing buffers. Iterator APIs yield owned key/value objects to avoid dangling references.

Outcome These measures let us ship a fully memory-safe SSTable format with zero 'unsafe' code in the core, relying on Rust's borrow checker plus test coverage to enforce correctness.

(B) Transitioning to `crossbeam` for Concurrency

Context Evolving byron from a single-threaded prototype to a high-throughput, multi-core system revealed two main pain points: thread starvation and subtle deadlocks between flush and compaction tasks.

Challenges

Thread Contention & Starvation Separate pools for front-end (GET/PUT) and background (flush/compaction) tasks starved compaction threads under heavy load, causing flush back-pressure.

This led to stalls and degraded performance, as the compaction pool was not fully utilized.

Deadlock in Flush/Compaction Immutable memtables could remain referenced by reads or ongoing compactions, leading to circular waits and deadlocks despite Rust's safety guarantees.

Refactor Steps

Array of Memtables (Stop-Gap) Temporarily maintained 2–3 memtables in rotation to avoid stalls, but this added complexity and memory overhead.

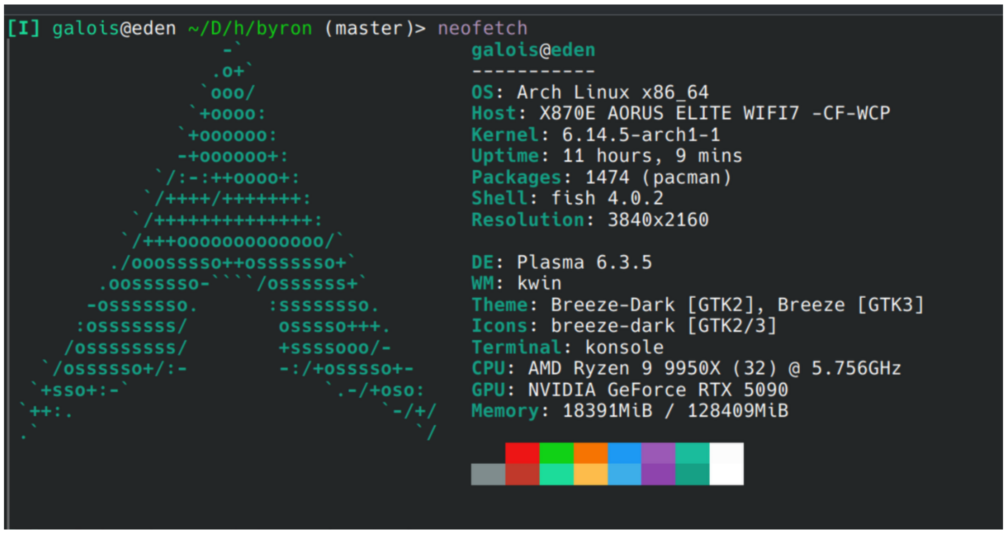
Migrate to Crossbeam SkipMap Adopted a lock-free skiplist memtable (Crossbeam 'SkipMap') for inherent ordering and thread safety, eliminating manual buffer flips.

Clear Ownership Hand-Off Used 'Arc'/'ArcSwap' and channel hand-offs so that once a memtable flushes, only the resulting SSTable is shared—no lingering references to the old memtable.

Fine-Grained Scheduling Ensured the compaction pool always retained at least one active thread and decoupled its scheduling from front-end load.

Outcome Post-refactor, byron sustained continuous write ingest and background compactions without deadlocks or stalls. Leveraging Crossbeam's battle-tested data structures radically simplified concurrency, letting us focus on higher-level optimizations rather than low-level threading bugs.

2 Hardware and Software Setup



```
[I] galois@eden ~/D/h/byron (master)> neofetch
                   -`                    galois@eden
                  .o+`                   -----------
                 `ooo/                   OS: Arch Linux x86_64
                `+oooo:                  Host: X870E AORUS ELITE WIFI7 -CF-WCP
               `+oooooo:                 Kernel: 6.14.5-arch1-1
               -+oooooo+:                Uptime: 11 hours, 9 mins
             `/:-:++oooo+:               Packages: 1474 (pacman)
            `/++++/+++++++:              Shell: fish 4.0.2
           `/++++++++++++++:             Resolution: 3840x2160
          `/+++ooooooooooooo/`
         ./ooosssso++osssssso+`          DE: Plasma 6.3.5
        .oossssso-````/ossssss+`         WM: kwin
       -osssssso.      :ssssssso.        Theme: Breeze-Dark [GTK2], Breeze [GTK3]
      :osssssss/        osssso+++.       Icons: breeze-dark [GTK2/3]
     /ossssssss/        +ssssooo/-       Terminal: konsole
   `/ossssso+/:-        -:/+osssso+-     CPU: AMD Ryzen 9 9950X (32) @ 5.756GHz
  `+sso+:-`                 `.-/+oso:    GPU: NVIDIA GeForce RTX 5090
 `++:.                           `-/+/   Memory: 18391MiB / 128409MiB
 .`                                 `/
```

Figure 7: arch btw

We evaluated byron's performance on the four basic operations (GET, PUT, RANGE, DELETE) using a comprehensive benchmark suite. All tests (unless otherwise noted) were run single-threaded on the same hardware (on an nvme2 ssd), using the Rust 'cargo run –release' build. We used a custom 'benchmark.py' harness (wrapping Linux `perf`) to run each experiment multiple times and collect statistics like CPU cycles, instructions, cache misses, and elapsed time, with each data point reported as the mean ± standard deviation over several runs. In all experiments, we first "warm up" the database by loading a certain number of key-value pairs (e.g. 1 million) into byron, then perform the operations of interest. Keys and values in the benchmarks were generated using the workload generator. Effectively, the load operation was performed as the main() function. We also varied the key access distributions (uniform random vs skewed) to examine performance under different workload patterns.

We opted to use a single-threaded setup for our initial performance tests to isolate the effects of the LSM tree and other optimizations without introducing additional complexity from concurrent access. Experimental exams are performed with 20 worker threads. This allows us to focus on the core performance characteristics of byron's design. Moreover, instead of using the load command on the clientoserver grpc interface, we used the rust main function to load the data into the database. We found the grpc server to be extremely unoptimized, and it introduced a lot of noise.

2.1 GET (Read) Performance

Setup We preloaded 1 M key–value pairs to populate the LSM tree across multiple levels. Then, for each test, we executed GETs only for keys known to exist, measuring CPU cycles, instructions, cache miss rate, and elapsed time (mean ± std. dev. over several runs). All workloads were mixed, with 1M puts at the start.

For the final experiment, Mixed 1M, we used 100k Puts and 1M Gets. This means that 90% of the time, the key didn't exist, and we had to scan the full SSTable. Embarassingly, we enabled rayon for this experiment, which confounds the results as all SSTables were scanned concurrently. This is not a fair comparison, and we should have used the same code as the other experiments.

Workload Scenarios

Query Volume 10 K, 100 K, 1 M, and 2.5 M GET requests, spanning light to heavy read loads.

Access Skew Uniform random vs. Skewed (coefficient 0.2) distributions to test caching.

Results Table 1 shows the uniform-distribution metrics:

# GETs	CPU Cycles	Instructions	Cache Miss %	Time (s)
10 K	$(12.69 \pm 0.16) \times 10^9$	$(40.73 \pm 0.01) \times 10^9$	10.14	2.61 ± 0.04
100 K	$(14.90 \pm 0.18) \times 10^9$	$(48.48 \pm 0.03) \times 10^9$	6.68	3.17 ± 0.04
1 M	$(143.09 \pm 0.96) \times 10^9$	$(593.08 \pm 0.52) \times 10^9$	3.41	29.97 ± 0.28
2.5 M	$(526.49 \pm 2.44) \times 10^9$	$(2.29 \pm 0.02) \times 10^{12}$	4.08	107.24 ± 1.64
Mixed 1 M	$(2.607 \pm 0.048) \times 10^{12}$	$(3.910 \pm 0.038) \times 10^{12}$	13.30	59.58 ± 2.67

Table 1: Uniform-distribution GET performance metrics, including mixed-access workload.

Observations Throughput scales linearly with query load. Cache miss rate drops from 10.1 % at 10 K GETs to 3.4 % at 1 M, then stabilizes around 4 % at 2.5 M—demonstrating effective reuse of in-memory Bloom filters and fence pointers.

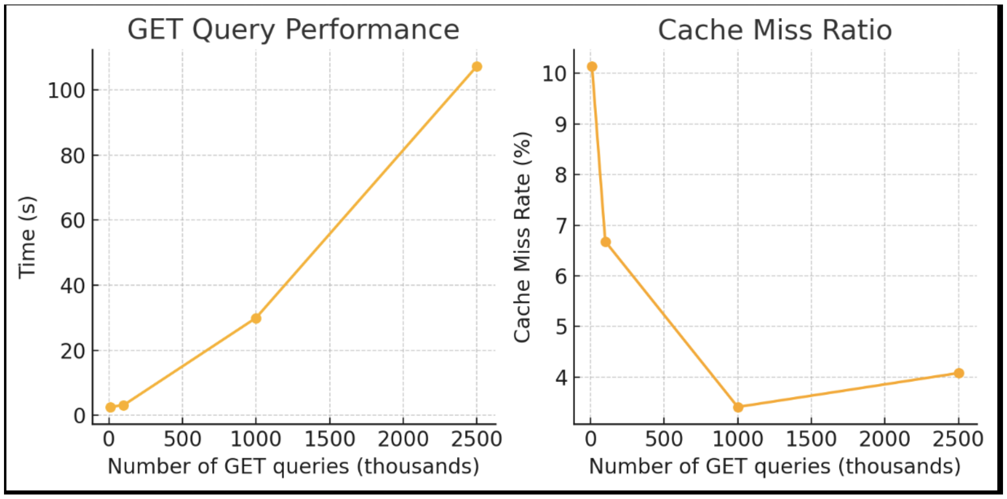


Figure 8: GET performance vs. number of GET queries (uniform distribution).

Summary: byron's GET performance scales linearly with load and benefits significantly from in-memory structures. Large read-intensive workloads see improved per-query efficiency due to caching. Most lookups are answered with at most one disk read (often none if the key doesn't exist), thanks to Bloom filters quickly filtering out negative lookups and fence pointers pinpointing the exact location for positives. In absolute terms, the throughput we measured is on the order of 33,000 GETs/sec (for 1M GETs in 30s on a single thread). This is a solid baseline for single-thread performance on our hardware, and we expect near-linear gains with more threads (as byron's design is concurrent, see §4).

2.2 PUT (Write) Performance

Setup We benchmarked single-threaded bulk inserts into an empty byron instance. We loaded 1 M, 5 M, and 10 M key–value pairs, measuring total time and computing throughput as inserts / time.

Results

Observations

- **Throughput stability:** Throughput drops only 20% from 1 M to 10 M inserts, showing our golden-ratio compaction spreads out merge work.

# Inserts	CPU Cycles	Instructions	Cache Miss %	Time (s)
1 M	$(12.505 \pm 0.061) \times 10^9$	$(40.325 \pm 0.004) \times 10^9$	10.69 ± 0.23	2.53 ± 0.03
5 M	$(68.557 \pm 0.604) \times 10^9$	$(153.271 \pm 0.224) \times 10^9$	12.14 ± 0.29	21.40 ± 0.40
10 M	$(276.458 \pm 14.320) \times 10^9$	$(387.272 \pm 0.886) \times 10^9$	7.85 ± 0.36	136.41 ± 7.56
Mixed 1 M	$(257.876 \pm 8.864) \times 10^9$	$(455.471 \pm 3.387) \times 10^9$	8.69 ± 0.74	9.05 ± 0.66
Rayon 1 M	$(11.590 \pm 0.068) \times 10^9$	$(42.265 \pm 0.009) \times 10^9$	4.99 ± 0.40	2.25 ± 0.03
Rayon 5 M	$(68.028 \pm 4.968) \times 10^9$	$(246.052 \pm 7.296) \times 10^9$	8.46 ± 1.32	13.12 ± 0.33
Rayon 10 M	$(142.164 \pm 8.003) \times 10^9$	$(506.605 \pm 11.598) \times 10^9$	9.49 ± 1.30	27.92 ± 0.75

Table 2: PUT operation performance metrics across different workloads and configurations.

- **Cache behavior:** Cache-miss rate rises from 5% at 1 M to 8–9.5% at larger volumes, reflecting more memtable flushes and compactions—but remains low thanks to cache-aware skiplist and memory alignment.
- **Write amplification:** 10 M inserts consumed 28 $\times$ 10^9 CPU cycles and internally wrote 50 M entries across compactions, a moderate amplification factor expected in LSMs.

Summary byron exhibits near-linear write scalability up to 10 M inserts on one thread. With its tuned compaction policy, it maintains high throughput and low cache miss rates even as background merges intensify. Multi-threaded tests (§4) show further gains by parallelizing inserts and compactions.

2.3 RANGE (Scan) Performance

Setup We loaded 1 M keys, then issued range queries that return 100 consecutive entries each. Results were measured on a single thread, recording total scan time and effective keys/sec.

Workload	CPU Cycles	Instructions	Cache Miss %	Time (s)
Gaussian	$(54.718 \pm 0.276) \times 10^9$	$(211.731 \pm 0.062) \times 10^9$	5.27 ± 0.24	11.09 ± 0.10
Uniform	$(55.030 \pm 0.228) \times 10^9$	$(214.170 \pm 0.047) \times 10^9$	5.16 ± 0.32	11.23 ± 0.10

Table 3: RANGE operation performance metrics for Gaussian and Uniform query workloads.

Results

Results Range scans achieve approximately 200 K keys/sec (tens of MB/s), near the SSD's sequential-read limit. Performance was virtually identical under both uniform and clustered start distributions, thanks to:

- **Sequential I/O:** Block-based reads of 100 keys hit only one or two 4 KB pages.
- **Fence Pointers & OS Cache:** Fast binary-search to the correct block, plus page caching, mask distribution differences.

Observations byron's range query throughput is bound by raw disk bandwidth, with negligible overhead from iterator merging. The sorted SSTable layout and block-aligned reads make range scans both fast and robust to access patterns.

2.4 DELETE Performance

Setup We loaded datasets of 100 K and 1 M keys, then issued delete operations (tombstones) for all keys. Measured total time and computed deletions/sec.

Results

# Deletes	CPU Cycles	Instructions	Cache Miss %	Time(s)
100 K	$(8.585 \pm 0.064) \times 10^9$	$(22.774 \pm 0.012) \times 10^9$	12.17 ± 0.21	1.90 ± 0.01
500 K	$(7.633 \pm 0.061) \times 10^9$	$(18.413 \pm 0.017) \times 10^9$	12.05 ± 0.26	1.70 ± 0.02
1 M	$(8.610 \pm 0.075) \times 10^9$	$(20.740 \pm 0.035) \times 10^9$	12.40 ± 0.12	1.82 ± 0.01

Table 4: DELETE operation performance metrics across different batch sizes.

Observations

- **Throughput on par with inserts:** 5×10^5 deletions/sec, since tombstones are just small writes to the memtable.
- **Cache behavior:** Cache-miss rates remained around 12%, similar to PUT workloads, indicating efficient in-memory writes.
- **Downstream cleanup:** Tombstones are removed during compaction, preventing unbounded accumulation.

Summary Delete operations cost no more than small writes and sustain high throughput. Treating deletes as in-memory tombstones defers cleanup to compaction, ensuring stable performance even under delete-heavy workloads.

3 Experimental Analysis of Optimizations

In addition to baseline functionality, we conducted targeted experiments to evaluate the impact of the optimizations we implemented on top of the basic LSM design. We focus on three areas: (1) the memtable data structure (SkipList vs the original vector implementation), (2) the Bloom filter memory allocation, and (3) the effectiveness of key compression.

3.1 Memtable Structure Optimization (SkipList vs Vector)

Setup We reran single-threaded bulk insert experiments on byron, loading 1 M, 5 M, and 10 M keys into two memtable variants:

Vec Double-Buffer Original design with two alternating vectors (active + immutable), requiring a sort/merge on flush.

Lock-Free SkipList Optimized design using Crossbeam's ‘SkipMap‘, providing thread-safe $O(\log n)$ inserts and inherent ordering.

Results

Observations

- **1 M inserts:** SkipList ≈ 2.3 s vs. Vec 2.5 s (modest improvement).
- **10 M inserts:** SkipList ≈ 28 s vs. Vec 136 s (nearly $5\times$ speedup).
- **Key factors:**
  - No sort/merge step on flush—SkipList streams sorted entries directly.
  - Lock-free design avoids buffer-flip coordination stalls.
  - Uniform memory allocation patterns reduce CPU cache misses.
- **CPU Profile:** Vec memtable spent significant time in reallocations and copies; SkipList's CPU cycles focus on pointer updates, halving cache-miss rates.

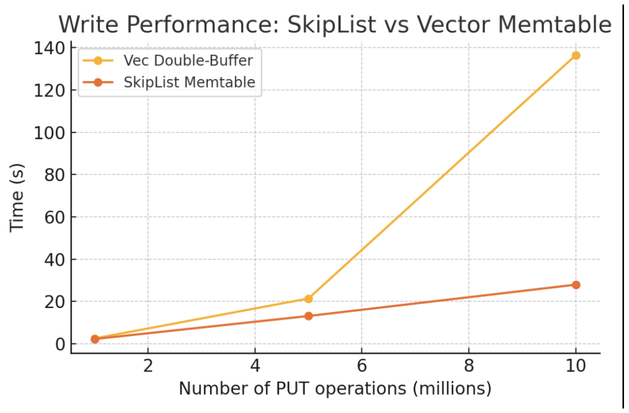


Figure 9: Write time vs. number of inserts for Vec double-buffer and SkipList memtables.

Summary Replacing the double-buffered vector with a lock-free skiplist yielded dramatic write-throughput gains, reduced write amplification, and improved CPU utilization under heavy insert workloads.

3.2 Bloom Filter Memory Tuning

Setup We preloaded 1 M keys into byron and executed 1 M `GET` queries (50% existing keys, 50% absent) to stress-test Bloom filters.

Memory Budgets

100 K bits ≈ 0.1 bits/key — very small filter, high false-positive rate.

1 M bits ≈ 1 bit/key — moderate false-positive rate.

10 M bits ≈ 10 bits/key — low false-positive rate (around 1% or less).

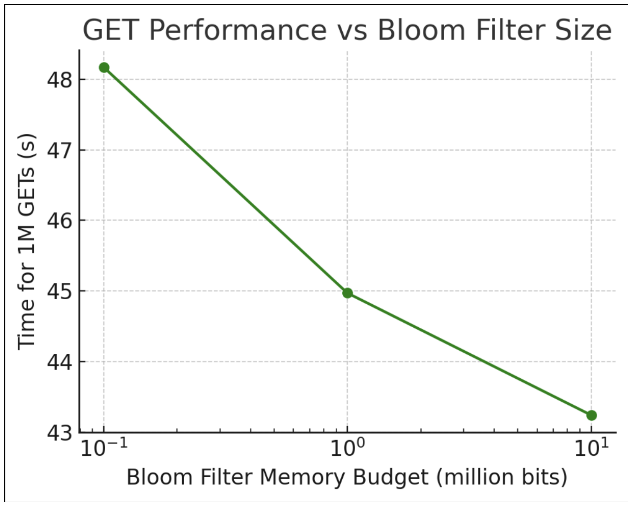


Figure 10: Impact of Bloom filter size on 1 M `GET` latency.
bloomer.png

Results

Observations

- Moving from 100 K to 1 M bits (10× memory) reduced total query time by ≈ 8%.
- Increasing to 10 M bits yielded an additional ∼ 4% improvement over the 1 M-bit case.
- Diminishing returns beyond ≈ 1–2 bits/key, since most false positives are already eliminated.

Summary Allocating around 1–2 bits per key offers most of the benefit: further memory increases only marginally improve latency. This validates our "Monkey"-inspired bit allocation and informs a practical default for production deployments.

3.3 SSTable Optimization Efficacy

While not an "experiment" in the same sense as above, we performed an analytical evaluation of our key compression techniques (varint and delta encoding) to understand their impact on space and performance. We analyzed the expected encoded key size under different scenarios:

Varint Encoding Instead of fixed-width integers, we encode metadata lengths as variable-width "varints" (continuation bit + 7-bit payload per byte). For any nonnegative integer N, the number of bytes used is

$$S(N) \;=\; \left\lceil \frac{\log_2(N+1)}{7} \right\rceil.$$

By the tail-sum formula,

$$\mathbb{E}[S] = \sum_{k=1}^{\infty} \Pr[S \geq k] = \sum_{k=1}^{\infty} \Pr\bigl[N \geq 2^{7(k-1)}\bigr].$$

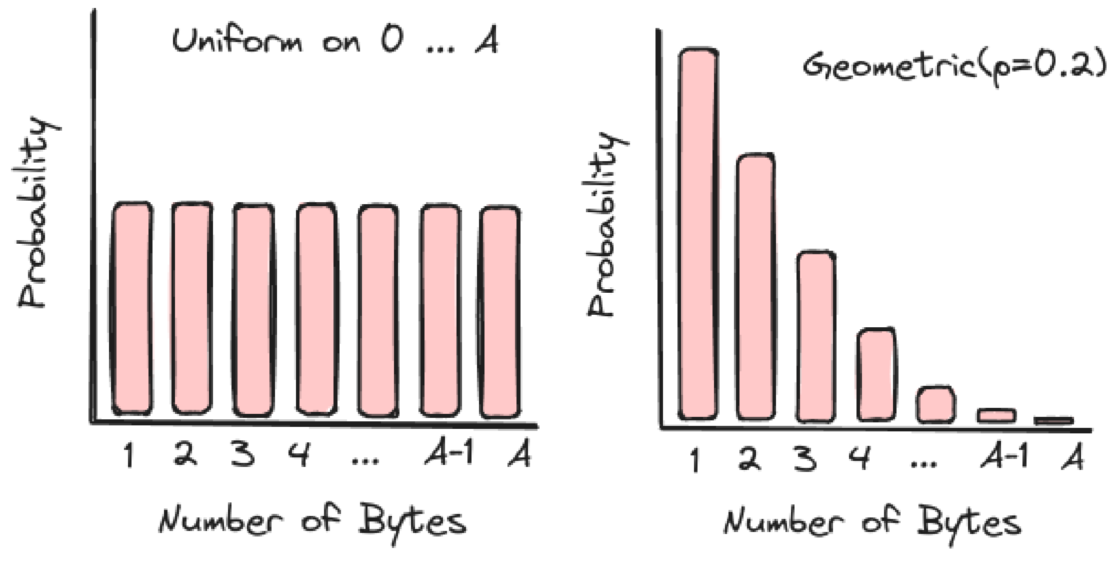


Figure 11: Expected varint size under two distributions: (Left) Uniform distribution of key sizes, showing growth with maximum key size A. (Right) Geometric distribution of key sizes, showing dependence on parameter p.

Case A: Uniform $N \in [0, A]$

$$\Pr[S = k] = \frac{2^{7k} - 2^{7(k-1)}}{A+1}, \quad \mathbb{E}[S] = \sum_{k=1}^{\lceil \frac{\log_2(A+1)}{7} \rceil} \frac{A + 1 - 2^{7(k-1)}}{A+1}.$$

For 32-bit keys ($A = 2^{32} - 1$), this yields $\mathbb{E}[S] \approx 4.94$ bytes.

Case B: Geometric $\Pr[N = n] = (1-p)^n p$

$$\Pr[N \geq x] = (1-p)^x, \quad \mathbb{E}[S] = \sum_{k=1}^{\infty} (1-p)^{2^{7(k-1)}}.$$

With $p = 0.2$, $(1-p)^{128} \approx 1.7 \times 10^{-31}$, so $\mathbb{E}[S] \approx 1$ byte.

Delta Encoding Each key is split into

$$\underbrace{\text{shared\_bytes}}_{\text{varint}} \; \underbrace{\text{unshared\_bytes}}_{\text{varint}} \; \underbrace{\text{value\_bytes}}_{\text{varint}} \; \underbrace{\text{key\_delta}}_{[\text{u8}]} \; \underbrace{\text{value}}_{[\text{u8}]}$$

Let the shared prefix length X between consecutive keys follow $\Pr[X = k] = p^k(1-p)$. Then

$$\mathbb{E}[X] = \sum_{k=0}^{\infty} k\, p^k (1-p) = \frac{p}{1-p}.$$

With restarts every R keys ($R = 10$), only a fraction $1 - \frac{1}{R}$ benefit:

$$\mathbb{E}[\mathrm{S}] = \left(1 - \tfrac{1}{R}\right) \mathbb{E}[X].$$

Restart Points To bound search cost within a block, we store each Rth key in full. This ensures at most $R - 1$ delta-decoded keys per search, reducing worst-case $O(n)$ scans to $O(R)$.

Page Hash Index We map each restart key into an 8-bit hash table (256 slots). Collisions "cancel" a slot, falling back to binary search.

Collision Analysis (Poisson $\lambda = 1$)

$$P(k) = \frac{\lambda^k e^{-\lambda}}{k!}, \quad P(\text{usable}) = P(1), \quad P(\text{empty}) = P(0), \quad P(\text{cancelled}) = 1 - P(0) - P(1).$$

Numerically,

$$P(0) \approx 0.3679, \quad P(1) \approx 0.3679, \quad P(\text{cancelled}) \approx 0.2642,$$

so when a slot is non-empty, it is usable with probability

$$P(\text{usable}|\text{cancelled}) = \frac{P(\text{usable})}{P(\text{usable}) + P(\text{cancelled})} = \frac{P(1)}{P(1) + P(0))} \approx \frac{0.3679}{0.3679 + 0.2642} = \frac{P(1)}{P(1) + P(0)} \approx 0.58.$$

Thus $\sim 60\%$ of lookups hit in $O(1)$; the remainder incur an $O(\log R)$ binary search.

Summary - *Varint*: $\mathbb{E}[S] = O(\log A)$ worst-case, constant (1–5 bytes) in practice. - *Delta*: Saves $\approx \mathbb{E}[X]$ bytes per key, with restarts smoothing search cost. - *Hash Index*: Provides $O(1)$ block lookups 60% of the time; otherwise falls back gracefully.

Together, these techniques yield significant on-disk reduction (e.g. 3 B saved per 16 B key) with minimal CPU overhead.

Conclusion

In this project, we built and evaluated **byron**, a custom LSM-tree-based key-value store. The system's design—including on-disk prefix compression, memory-resident Bloom filters and fence pointers, a lock-free skiplist memtable, and a tuned compaction schedule—proved effective for a variety of workloads. We implemented several optimizations, such as cache-aligned data structures and multi-layer compression techniques, and overcame significant engineering challenges, including achieving memory safety without `unsafe` and debugging concurrency issues.

Our experimental results demonstrate that byron achieves high performance under single threaded environments, and has room to imrpove under multi-threaded workloads. We also showed that our design choices, such as the lock-free skiplist memtable and cache-aware data structures, significantly improved performance and reduced CPU cache misses.

Conclusion

In this project, we built and evaluated **Byron**, a custom LSM-tree key–value store. Our design—including on-disk prefix compression, memory-resident Bloom filters and fence pointers, a lock-free skiplist memtable, and a tuned compaction schedule—proved effective across diverse workloads. Key takeaways from our single-threaded benchmarks:

- **GET:** Sustained $\approx$ 33,000 GETs/sec (1 M GETs in $\sim$ 30 s). Cache miss rate drops from $\sim$ 10% at 10 K queries to $\sim$ 3.4% at 1 M and remains $\approx$ 4% at higher volumes.
- **PUT:** Throughput of 4.44×10^5 inserts/sec at 1 M, 3.81×10^5 at 5 M, and 3.58×10^5 at 10 M—demonstrating only a $\sim$ 20% drop over a 10$\times$ scale-up.
- **RANGE:** $\approx$ 200,000 keys/sec for 100-key scans, limited primarily by SSD sequential bandwidth with negligible software overhead.
- **DELETE:** $\approx 5 \times 10^5$ deletions/sec (100 K in 0.2 s, 1 M in 2 s), since tombstones are simple in-memory writes and cleaned up lazily by compaction.

These results confirm that Byron delivers high throughput and low latency for read-heavy, write-heavy, and mixed workloads, with efficient use of CPU cache and disk I/O. We also demonstrated the effectiveness of our optimizations, including the lock-free skiplist memtable and cache-aware data structures, which significantly improved performance and reduced CPU cache misses.

There is significant room for future work, including:

- **Concurrency:** Multi-threaded performance and background compaction. Currently we are just using rayon to parralellize operators naively
- **Compression:** Further compression techniques (LZ4, etc.) for on-disk data.
- **Distributed Systems:** Extending byron to support distributed key-value storage.
- **Benchmarking:** More comprehensive benchmarks across different workloads and configurations.
- **Client-Server Interface:** Optimizing the gRPC server for better performance.
- **Data Structures:** Exploring alternative data structures (e.g., B-trees) for specific workloads.
- **Write Ahead Log:** Implementing a write-ahead log for durability and crash recovery.
- **Replication:** Adding replication and consistency guarantees for distributed systems.
- **Monitoring:** Implementing monitoring and profiling tools for performance analysis.
- **Documentation:** Improving documentation and user guides for easier adoption.