An Empirical Comparison of Key-Value Store Performance: A Study of B+ Tree, LSM Tree, and Hash Table Architectures

Matthew Song
CS497 - Advanced Database Systems

***Abstract*—Different key-value (KV) stores exhibit vastly different performance characteristics due to their underlying data structures. This report presents an empirical analysis of four popular KV stores, each representing a distinct architectural approach: Redis and Memcached (in-memory hash table), MongoDB (B+ Tree), and RocksDB (Log-Structured Merge Tree). Using the Yahoo! Cloud Serving Benchmark (YCSB), I subjected these engines to a series of stress tests, including balanced, read-heavy, write-heavy, and range-scan workloads, while scaling concurrency from 1 to 16 threads. My findings reveal clear performance trade-offs. In-memory hash tables like Redis excel in balanced workloads, achieving over 103,520 ops/sec. The LSM-Tree-based RocksDB dominates read-heavy workloads at 119,190 ops/sec and is vastly superior in range scans, outperforming Redis by nearly 30x. The B+ Tree implementation in MongoDB offers consistent, albeit lower, throughput but demonstrates the most efficient multi-core scaling. These results provide a quantitative framework for guiding engineers in selecting the optimal KV store by aligning the engine's architectural strengths with specific application workload patterns.**

I. Introduction

Key-value (KV) stores have become a foundational component of the modern technology stack, prized for their simplicity, scalability, and performance. However, the choice of a KV store is non-trivial, as their internal designs—ranging from B+ trees to Log-Structured Merge (LSM) trees and in-memory hash maps—lead to significant variations in performance. An engine optimized for high-volume, write-heavy data ingestion may falter in a read-intensive or scan-heavy scenario.

This study aims to provide a clear, side-by-side comparison of how these architectural differences manifest under realistic stress tests. I evaluate four popular KV engines, selected as representatives of their respective data structure families:

- **Hash Table (In-Memory):** Redis and Memcached
- **B$^+$-tree:** MongoDB (using the WiredTiger storage engine [4])
- **LSM Tree:** RocksDB [2]

By subjecting these databases to a standardized suite of benchmarks using YCSB [1], I seek to answer the following question: *How do hash-table, B$^+$-tree, and LSM-tree designs shape key-value-store latency, throughput, and multi-core scaling under contrasting workloads—and how can these findings guide engineers in choosing and tuning the right engine for a given application?*

II. Initial Assumptions

Before conducting the benchmarks, I held several common assumptions regarding the expected performance characteristics of the different key-value store architectures, based on their theoretical designs and widely accepted knowledge in the database community:

A. Hash Table Architectures (Redis, Memcached)

- **Point Operations:** Expected to offer the lowest latency and highest throughput for single-key reads (GETs) and writes (SETs) due to their typical O(1) average-case time complexity and in-memory operation.
- **Range Scans:** Anticipated to perform poorly, or not support ordered scans natively, as hash tables do not maintain keys in a sorted order.
- **Scalability:** Multi-core scalability was presumed to be a potential bottleneck, particularly for designs like Redis [3] employing a single main event loop for command execution. Memcached, with its multi-threaded architecture, might scale better initially but could face contention on internal locks.
- **Consistency/Durability Overheads:** Memcached, being non-persistent by default, was expected to have less overhead than Redis when Redis persistence features (like AOF or RDB snapshots) are active.

B. B$^+$-tree Architectures (MongoDB)

- **Balanced Performance:** Expected to provide a good balance between read and write performance, suitable for general-purpose workloads.
- **Point Operations:** Reads might involve multiple disk I/Os if data is not cached (logarithmic complexity), potentially leading to higher latency than in-memory hash tables. Writes involve in-place updates, which can be complex and may trigger page splits.
- **Range Scans:** Anticipated to be efficient due to the ordered nature of data storage in leaf nodes, allowing sequential access.
- **Scalability:** Expected to scale well with multiple cores due to mature concurrency control mechanisms like page-level or document-level locking.
- **Write Amplification:** Lower write amplification compared to LSM-trees, as updates are often in-place.

C. LSM-tree Architectures (RocksDB)

Based on the Log-Structured Merge-tree design [5]:

- **Write Performance:** Expected to exhibit excellent write throughput, as writes are typically append-only to an in-memory memtable and then sequentially flushed to disk.
- **Read Performance:** Point read performance was anticipated to be variable. Reads could be very fast if data is in the memtable or a shallow SST (Sorted String Table) level, but potentially slower if data resides in deeper levels, requiring multiple lookups (read amplification). Bloom filters were expected to mitigate some of this.
- **Range Scans:** Expected to be efficient, as SST files are sorted, though merging data from multiple levels during a scan could add overhead.
- **Compaction Overhead:** Background compaction processes, necessary to merge SST files and reclaim space, were assumed to potentially impact foreground operation latencies, especially tail latencies.
- **Space Amplification:** Could be higher than B^+-trees due to obsolete data versions retained until compaction.

III. Benchmark Methodology

To ensure a fair and reproducible comparison, I established a standardized benchmarking environment and methodology based on YCSB [1].

A. Test Configuration

All benchmarks were executed on a macOS Darwin 24.5.0 platform. Each of the 60 test scenarios involved a single database to avoid resource contention.

- **Record Count:** 100,000
- **Operation Count:** 100,000
- **Thread Counts:** 1, 4, 8, and 16 client threads
- **Request Distribution:** Zipfian, to simulate realistic access patterns where some records are more popular than others.

B. Workloads

I utilized four standard YCSB workloads to simulate common application behaviors:

1) **Balanced Workload (50% Reads, 50% Writes):** Simulates general-purpose applications with a mix of read and write operations.
2) **Read-Heavy Workload (90% Reads, 10% Writes):** Represents applications like caching layers or content delivery systems where data is frequently read but seldom updated.
3) **Write-Heavy Workload (10% Reads, 90% Writes):** Models data ingestion pipelines, logging systems, or applications with frequent data creation and updates.
4) **Range Query Workload (95% Scans, 5% Inserts):** Evaluates performance for analytics or sequential data processing tasks that require scanning ranges of records.

C. Data Collection

I collected performance metrics for both the data loading and execution phases of each test. The primary metrics for this analysis are **throughput** (measured in operations per second) and **latency** (measured in microseconds (μs) at the average, 95th (P95), and 99th (P99) percentiles).

IV. Results and Analysis

My analysis reveals that each data structure presents a unique performance profile, excelling in some areas while lagging in others.

A. Overall Performance

The highest throughput achieved by each data structure's representative database highlights their core strengths:

- **LSM Tree (RocksDB):** Peaked at **119,190 ops/sec** in a read-heavy workload with 8 threads.
- **Hash Table (Redis):** Peaked at **103,520 ops/sec** in a balanced workload with 16 threads.
- **Hash Table (Memcached):** Pretty balanced across all workloads.
- **B^+-tree (MongoDB):** Peaked at **51,414 ops/sec** in a balanced workload with 16 threads.

B. Workload-Specific Deep Dive

1) Read-Heavy Workload: In this scenario, RocksDB is the clear winner, followed by Redis.

1) **RocksDB (LSM Tree):** The combination of a large block cache and Bloom filters allows RocksDB to rule out most disk reads. A lookup is either served directly from cache or requires touching a single data block, resulting in extremely high throughput.
2) **Redis (Hash Table):** As an in-memory store, a GET operation is a simple hash and pointer dereference. Throughput climbs to just under **100,000 ops/sec** but is ultimately bottlenecked by the single event-loop thread that all commands must pass through.
3) **Memcached (Hash Table):** Although the data live entirely in DRAM, each lookup has to grab a hash-bucket or slab mutex, and that lock accounting shows up in the numbers: around **25,000** ops/s with one worker and only around **56,000** ops/s even under heavier load. The mutex traffic, plus an extra memory copy from slab to write buffer, trims enough microseconds from every request to keep Memcached well below both Redis and RocksDB.
4) **MongoDB (B^+-tree):** Every key access requires traversing three to four B-tree nodes. Even when cached, these extra pointer hops and object metadata checks result in lower throughput.

2) Write-Heavy Workload: Here, the append-only nature of LSM trees gives RocksDB a distinct advantage.

1) **RocksDB (LSM Tree):** An insert is a sequential WAL append and a memtable write. With large in-memory

write batches and background compaction, the foreground path remains exceptionally short, leading to the highest throughput at every thread count.

2) **Redis (Hash Table):** Writes are in-RAM pointer assignments; without an immediate fsync, the critical path is only hash computation and object creation. Although a single event loop serialises commands, that loop is efficient enough that Redis outpaces Memcached in three of the four thread settings.
3) **Memcached (Hash Table):** Pure memory inserts keep base latency low, but each mutation briefly grabs both a hash-bucket lock and a slab-class lock, adding overhead absent in Redis. Those mutex costs hold Memcached just below Redis except at the four-thread mark, where extra workers momentarily tip it ahead.
4) **MongoDB (B^+-tree):** Every put traverses multiple B^+-treelevels, may trigger leaf-page splits, and must be durably journaled. That heavier per-write work leaves MongoDB well behind the in-memory engines and far below RocksDB's append-only path in all configurations.

3) Balanced Workload: As cores scale up, Redis takes the lead.

1) **Redis (Hash Table):** When reads and writes are both O(1) RAM operations, adding the other half of the workload hardly hurts—until memory pressure forces evictions. With the data volume used here Redis stays comfortably in memory and tops the chart.
2) **RocksDB (LSM Tree):** Each read is still cache-plus-filter fast and each write is still append-only, but the MemTable flush frequency increases relative to the read-heavy run. Those extra background flushes explain why it trails Redis in the mixed load.
3) **Memcached (Hash Table):** Lock contention now comes from both gets and puts, so the effective hit on concurrency is bigger than in either extreme workload, pushing the bar below RocksDB.
4) **MongoDB (B^+-tree):** The engine must interleave random writes, journal syncs, and buffer-cache reads. Cache churn and page pins cut into both sides of the 50/50 mix, leaving it last.

4) Range Query Workload: This workload exposes the most dramatic architectural differences.

1) **RocksDB (LSM Tree):** SST files are immutable, fully sorted runs. Once a scan starts, the iterator streams large contiguous blocks, which modern SSDs pre-fetch efficiently. Levelled compaction keeps key ranges from spanning many files, so very few overlaps have to be merged on the fly.
2) **MongoDB (B^+-tree):** B^+-treeleaves are physically linked. After locating the first key, the cursor walks those links sequentially. It is fundamentally a good fit for scans, but each hop still means a pointer chase and an occasional uncached page, hence 2.5× slower than RocksDB's flat-file streaming.
3) **Redis (Hash Table):** Fails catastrophically. The native key space is hash-distributed, so a true lexicographic range requires a full iteration (SCAN) over all keys. Only a small test data set keeps this from dropping to single-digit kops/sec.

C. Threading Scalability Analysis

Scaling efficiency—how well a database utilizes additional CPU cores—is critical for modern applications.

- **MongoDB (B^+-tree):** Demonstrates the best threading scalability across all workload types due to its document-level locking mechanism and WiredTiger storage engine optimizations. The database achieves excellent parallel execution by allowing multiple threads to operate on different documents simultaneously without blocking each other, while its group-commit journal optimization amortizes fsync costs across multiple operations. For range queries, MongoDB's B+ tree structure enables each scanning thread to walk independent leaf-page chains with minimal lock collisions, making it the clear winner for concurrent operations at scale.
- **Memcached (Hash Table):** Memcached exhibits moderate threading scalability that benefits from worker threads until hash-bucket and slab-allocator mutex contention becomes the limiting factor. While multiple reader threads can operate concurrently, each lookup must acquire hash-bucket locks, and the slab allocator introduces additional mutex overhead that becomes more pronounced at higher thread counts. The architecture's reliance on memory-only operations provides good baseline performance, but the locking granularity prevents it from achieving the same scaling efficiency as MongoDB, particularly under write-heavy workloads where slab allocation becomes a bottleneck.
- **Redis (Hash Table):** Redis suffers from fundamental threading scalability limitations due to its single-threaded event loop architecture, where all command execution occurs on a single core regardless of the number of client threads. While additional threads can handle network I/O and queue commands, the core execution remains serialized, causing throughput to plateau rather than scale linearly with thread count. This design choice, while simplifying consistency and eliminating race conditions, creates a clear ceiling for concurrent performance that becomes especially apparent under write-heavy and range-query workloads where command processing becomes the bottleneck.
- **RocksDB (LSM Tree):** RocksDB demonstrates the poorest threading scalability among all tested databases due to the inherent complexities of its LSM tree architecture and background operations. The database suffers from reader contention on block cache and Bloom-filter metadata, frequent memtable flushes that stall writers, and background compaction processes that compete for I/O resources with foreground operations. Mixed workloads particularly expose these limitations as they trigger more

frequent compactions and cache conflicts, while the need for SST level reconciliation during range queries further limits the benefits of additional threads, making RocksDB better suited for single-threaded or lightly concurrent scenarios.

D. Latency Analysis

Low average latency is important, but tail latency (P95, P99) often has a greater impact on user experience.

- **RocksDB:** RocksDB consistently achieves the lowest latency across most workload types, with sub-10 microsecond averages and 99th percentiles typically under 30 microseconds, thanks to its efficient Bloom filter-guided lookups and sequential memtable operations. The LSM tree architecture excels at both point lookups through cache-friendly access patterns and range queries via sorted SSTable streaming, though occasional compaction activities can cause brief cache line pre-emption that slightly elevates tail latency.
- **Redis:** Redis delivers strong latency performance for point operations with O(1) hash probes averaging around 70 microseconds, but suffers from periodic stalls caused by AOF fsync operations and hash table rehashes that can double P95 and P99 latencies. The single-threaded architecture eliminates lock contention but creates severe performance degradation for range queries, where hash-distributed keyspace forces full SCAN operations resulting in multi-millisecond pauses at higher percentiles.
- **Memcached:** Memcached provides competitive mean latency performance close to Redis for simple operations, but worker thread lock contention significantly inflates tail latency, pushing 95th and 99th percentiles to 2.5× the mean under load. Hash bucket and slab allocator mutex waits become particularly problematic during write-heavy workloads, where lock pressure can elevate 99th percentile latency above 300 microseconds.
- **MongoDB:** MongoDB consistently exhibits the highest point-operation latency across all workloads due to B+ tree traversal overhead and synchronous journal writes, with 99th percentile latencies often exceeding 500 microseconds. Random 4KB SSD reads on cache misses and page splits during writes create significant I/O queue collisions, while document locking and metadata checks add computational overhead that pushes tail latency well above other systems.

V. Discussion and Use-Case Recommendations

The performance characteristics observed directly map to the architectural trade-offs made by each database engine, making them suitable for different use cases.

- **RocksDB (LSM Tree):**
  - *Strengths:* Unmatched write throughput and exceptional performance on both point reads and range scans.
  - *Weakness:* Scaling efficiency degrades under heavy concurrent load due to contention between foreground and background operations.
  - *Best Fit:* **Data-ingest pipelines,** time-series databases, and logging systems. These applications are dominated by append-only writes, where RocksDB's single-thread performance shines and multi-core scaling is less critical.
- **Redis (Hash Table):**
  - *Strengths:* Microsecond latency for in-memory point operations; leads in balanced workloads.
  - *Weaknesses:* Poor multi-core scaling and catastrophic failure on range queries.
  - *Best Fit:* **Front-end caching layers,** user session stores, and real-time leaderboards. These workloads demand the lowest possible latency for single-key operations, can tolerate relaxed durability, and fit within system RAM.
- **Memcached (Hash Table):**
  - *Strengths:* Extremely simple, stateless, and free from persistence overhead, offering latency close to Redis.
  - *Weaknesses:* No durability, no ordered scans, and limited scaling due to lock contention.
  - *Best Fit:* **Transient edge caches** for API throttling keys or CDN metadata. Here, simplicity is key, occasional data loss is acceptable, and the application layer handles sharding.
- **MongoDB (B^+-tree):**
  - *Strengths:* Best multi-core scaling efficiency and strong range-scan performance.
  - *Weaknesses:* Highest per-operation latency and lowest single-thread throughput.
  - *Best Fit:* **Document-centric OLTP systems** and microservice backends. These applications run on multi-core servers, serve many concurrent users, and benefit from flexible schemas and secondary indexes, making MongoDB's superior scaling a decisive advantage.

A. Comparison with Initial Assumptions

The benchmark results (Section IV) largely align with my initial assumptions (Section II), but also revealed several nuances and surprising outcomes:

1) Hash Table Architectures (Redis, Memcached):

- **Point Operations Superiority:** The assumption of low latency for point operations was generally confirmed. Redis demonstrated excellent average latency. However, RocksDB's cached reads sometimes showed even lower average latency, indicating that highly optimized caching in other architectures can challenge in-memory stores for average-case reads, though Redis's in-memory nature provides strong predictability.
- **Range Scan Inefficiency:** This was strongly validated. Redis's range query throughput was extremely low, and Memcached offered no effective support.

- **Scalability Limitations:** The limited multi-core scaling for Redis was evident, as its throughput did not increase linearly with higher thread counts beyond a certain point, consistent with its single event-loop design. Memcached also showed diminishing returns, likely due to internal lock contention.
- **Memcached Simplicity vs. Redis Performance:** While Memcached is simpler, Redis generally achieved higher throughput across workloads, suggesting its event loop and data structure implementations are highly optimized even with optional persistence.

2) B^+-tree Architectures (MongoDB):

- **Balanced Performance Profile:** MongoDB provided consistent, though not peak, performance across different workloads, aligning with the assumption of a balanced profile. However, it was generally outperformed by specialized engines in their optimal scenarios (e.g., RocksDB for writes/scans, Redis for cached reads).
- **Point Operation Latency:** As expected, MongoDB's point operation latencies were higher than in-memory stores and well-cached RocksDB reads, reflecting its B-tree traversal costs.
- **Range Scan Efficiency:** This was confirmed, with MongoDB providing the second-best range scan performance after RocksDB.
- **Multi-Core Scalability:** MongoDB demonstrated the best scaling efficiency among the tested databases, a key finding that validated the assumption of mature concurrency mechanisms.

3) LSM-tree Architectures (RocksDB):

- **Dominant Write Performance:** RocksDB's excellent write throughput was strongly confirmed, leading in write-heavy workloads as anticipated due to its append-optimized design.
- **Strong Read Performance (with Caching):** Contrary to concerns about potentially slow reads, RocksDB excelled in read-heavy workload, indicating very effective caching, Bloom filters, and memtable utilization that largely mitigated read amplification for hot data. Its average read latency was surprisingly low when data was cached.
- **Efficient Range Scans:** This was strongly validated, with RocksDB significantly outperforming all other systems in range query throughput.
- **Compaction Impact on Scalability:** The assumption about compaction overhead affecting performance was supported by RocksDB's reduced scaling efficiency in mixed workloads, where increased write activity would trigger more compactions, contending for resources with foreground operations.

The primary deviation from my initial assumptions was the exceptionally strong read performance of RocksDB, challenging the notion that LSM-trees are solely write-optimized at the expense of read speed. Modern LSM-tree implementations with sophisticated caching and filtering mechanisms can achieve read performance competitive with, or even exceeding, other architectures for certain access patterns.

VI. Conclusion

My benchmark analysis confirms that there is no universally "best" key-value store. The choice of engine is a critical engineering decision that requires a deep understanding of the application's specific workload.

- For **write-intensive logging and analytics**, the append-only, streaming-scan design of an **LSM tree (RocksDB)** is superior.
- For **low-latency caching of individual items**, an **in-memory hash table (Redis, Memcached)** is the optimal choice.
- For **general-purpose workloads with high concurrency**, the superior multi-core scaling of a **B^+-tree (MongoDB)** provides the most consistent and predictable performance.

By quantifying these trade-offs, this report provides a clear guide for engineers to select and tune the right engine, ensuring their systems are not only functional but also highly performant.

References

[1] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10)*, 2010.

[2] The RocksDB Team, *RocksDB official documentation*, 2024. [Online]. Available: https://rocksdb.org/

[3] Redis, *Redis official documentation*, 2024. [Online]. Available: https://redis.io/documentation

[4] MongoDB, Inc., "The WiredTiger Storage Engine," *MongoDB Manual*, 2024. [Online]. Available: https://www.mongodb.com/docs/manual/core/wiredtiger/

[5] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.