KV-Store Benchmark

Matthew Song

Introduction

- **Motivation** – Different key-value stores have wildly different internal designs (B+ trees vs. LSM trees vs. in-memory hash maps)

- **Aim** – put a bunch of popular KV engines through the same stress tests so we can see, side-by-side, how they behave under different workloads.

- **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?

Introduction

- **Database Benchmarked (expected)**
  - B+ Tree Engines
  - LSM Tree Engines
  - HashMap Engine

Introduction

- **<u>Database Benchmarked (actually)</u>**
  - B+ Tree Engines

    MongoDB → WIREDTIGER
  - LSM Tree Engines

    RocksDB
  - HashMap Engine

    redis MEMCACHED

Introduction

- **High level Workflow**
  - Install all databases + YCSB
  - Run each YCSB workload (some manually implemented) against each engine
  - Collect JSON output
  - Feed results into a simple Flask server and create visualized dashboard

Introduction

- **Testing Workloads**
  - Read Heavy Workload (90% reads, 10% updates)
  - Write Heavy Workload (10% reads, 90% updates)
  - Balanced Workload (50% reads, 50% updates)
  - Range Query Workload (95% scans, 5% inserts)

Introduction

- **Throughput (operations per second, ops/sec)**
  - *Higher is better*: shows how much work the engine can finish in a fixed time window.
- **Average Latency (mean response time)**
  - Represents the “typical” user experience for a single request.
- **95th-Percentile Latency**
  - Time within which 95 % of operations finish—captures near-worst-case delays for most users.
- **99th-Percentile Latency**
  - Tightens the lens to the slowest 1 % of requests; crucial for tail-latency–sensitive applications such as real-time analytics or interactive dashboards.
- **Multi-Core Scalability**
  - Efficiency of speed-up as threads (or CPU cores) increase indicating near-linear scaling or revealing bottlenecks (locks, I/O contention, compaction, etc.).

Introduction

MongoDB®

- **Data structure**: on-disk B^+-tree with prefix/value compression; one buffer-cache layer in RAM.

- **Read path**: root → internal → leaf traversal (≈ 3–4 hops); if a node is cached, lookup stays in memory.

- **Write path**: update the leaf page, split if full, then mark dirty for later flush.

- **Concurrency model**: document-level intent locks let many readers and writers operate in parallel

Introduction

RocksDB

- **Data structure**: WAL + mutable memtable in RAM; immutable memtables flush to sorted SST files on disk.
- **Read path**: consult Bloom/Ribbon filters for each level; at most one SST per level is touched, often from block cache.
- **Write path**: append to WAL, insert into memtable; background flush and compaction rewrite SSTs to keep read-amp low.
- **Concurrency model**: foreground inserts are lock-free; multiple flushers and compaction threads run in the background.

Introduction

- **Data structure**: keys map to in-RAM objects (strings, hashes, lists, sets, etc.) with space-saving encodings chosen on the fly.

- **Execution model**: single main event loop processes all commands

- **Read/Write cost**: both are O (1) hash look-ups

- **Data structure**: global hash table protected by per-bucket mutexes; values stored in fixed-size slab pages to curb fragmentation.

- **Process model**: one listener thread hands connections to multiple worker threads

- **Read/Write cost**: O (1) hash probe plus brief mutex hold; no persistence layer, so no fsync penalties.

* **For SCAN!** - Redis treats itself like a tiny in-memory database, so it offers a non-blocking SCAN command that steps through the keyspace (all stored keys) in small batches using a “cursor” without pausing traffic; Memcached is a pure cache where items can vanish anytime (TTL or eviction), and a full scan would need heavy locks on its hash-table/slab memory, stall performance, and reveal sensitive key names, so the feature is intentionally left out.