International Journal of Leading Research Publication (IJLRP)
E-ISSN: 2582-8010 ● Website: www.ijlrp.com ● Email: editor@ijlrp.com

# Technical Examination of Key-Value Store Scalability

**Nikhita Kataria**

nikhitakataria@gmail.com

**Abstract**

**Key value stores encapsulate data, metadata, a globally unique identifier and data attributes into a single immutable entity termed an object. In an object store, objects are organized in a flat address space i.e every object exists at the same level in a large scalable pool of storage. Key value stores follow location independent addressing and are accessed via the unique identifier. Key value stores are designed primarily to manipulate data sets that does not have a predefined data model or in other words is unstructured. The key operations for an object store include retrieval (get), store(put) and deletion(delete). In this survey, we aim to evaluate different classes of key values stores in terms of feature set, design choices, architectures, performance and their adaptability in the current market.**

**Keywords: Key Value Stores, Document Stores, Memory, Replication, Scalability, Caching, Read Vs Write Performance**

## I. INTRODUCTION

Key value stores can be categorized into following categories based on the design approach, the object format, replication policy, caching mechanisms, consistency aspects, discovery protocols etc. ***General Key Value stores*** based on the classic key value store design approach of storing data in the form of object value stores. Some examples include Redis, Memcached. ***Column Based stores*** often store column entries as a continuous entry in the disk thus making the access faster. In graph based stores, data is stored in the form of nodes and edges where nodes are instances of the objects and the relationship among objects are represented using edges. HBase and Cassandra are based on this design. ***Document based stores*** embed data in the form key value pairs and the associate metadata in the form of documents. Some of the famous examples are MongoDB, Riak, Terrastore. There are other classes of specialized key value stores designed to suit the needs of certain applications. Some examples are Pregel, VertexDB etc.

There are various implementations of each of the classes mentioned above and the first step for this research was to select the key value stores that should be studied in order to represent the rationale behind these classes. Figure 1 represents the key value stores we explored. Next we selected the representatives on the basis of popularity, adaptability, ease of use, features, scalability, flexibility and reliability factors for each class of the key value stores. In the next few sections, we explore details of the selected representative stores.

TABLE I. **RANKING BASED ON PROPERTIES**

| ***Category*** | ***Popularity*** | ***Performance*** | ***Researched in this paper*** |
|---|---|---|---|
| Regular Key Value Stores | Redis<br>Memcached<br>BerkleyDB<br>TokyoCabinet<br>Voldemort<br>Dynamo | Redis<br>Tokyo Cabinet<br>BerkleyDB<br>Memcached<br>Voldemort<br>Dynamo | BerkleyDB<br>Memcached<br>Redis |
| Document Based Key Value Stores | CouchDB<br>Riak<br>OrientDB<br>MongoDB<br>Azure's Document DB<br>Apache JackRabbit<br>Terrastore | CouchDB<br>Riak<br>OrientDB<br>MongoDB<br>Azure's Document DB<br>Apache JackRabbit<br>Terrastore | Riak<br>MongoDB |
| Column Based Key Value Stores | Cassandra<br>HBase<br>HyperTable<br>BigTable | Cassandra<br>HBase<br>HyperTable<br>BigTable | Cassandra<br>HBase |

## II. DOCUMENT BASED STORES

### *A. Riak*

Riak has adopted its architecture from Amazon's Dynamo with style variations. In general, distributed key value stores follow architecture in lines with master and slave paradigm, Riak takes its way off from this approach by treating all the nodes in the cluster equally responsible for the tasks assigned to them. So from a user's perspective it can contact any node and send a get, put or an update request.

Riak uses consistent hashing to distribute data across cluster nodes, reducing data movement during node failures and enhancing resilience. The $2^{160}$-bit keyspace is split into partitions, grouped into virtual nodes, which are assigned to physical nodes. This setup allows virtual nodes to be redistributed during failures without relying on the physical node layout.

Riak organizes data in buckets, with each bucket holding keys. A binary hash is calculated for each bucket/key pair and placed on an ordered ring, divided into partitions. Each virtual node controls one or more partitions. For data versioning and causal consistency, Riak uses vector clocks to maintain multiple object versions, which is useful in applications like malware analysis.

Cluster topology changes are shared through the gossip protocol, but in edge cases, it may overwhelm network resources due to excessive messages. Riak follows an "always writable" design, ensuring

availability even during partitioned or degraded states. It uses a read-repair mechanism to synchronize missing data and performs active anti-entropy tasks to fix data inconsistencies. However, Riak lacks a built-in load balancer, requiring third-party solutions. The gossip protocol, while efficient in typical conditions, may become a bottleneck in complex or irregular network environments.

### B. *MongoDB*

MongoDB is a scalable, document-based key-value store with replication support. It consists of configuration servers that manage metadata and shard servers that store data. Documents are divided into chunks and replicated across shards. MongoDB supports replication through replica sets, where documents are replicated across nodes.

The configuration servers store metadata such as chunk lists and shard ranges. MongoDB offers multiple sharding policies: range-based, hash-based, and location-aware, making it highly adaptable. By default, documents with similar shard key values are co-located on the same shard to enhance locality of reference.

In MongoDB, writes are directed to the primary node, while reads are served by replicas, ensuring high read availability. It also processes writes in batches to improve concurrency and optimize disk I/O. Clients may see write results before they are fully propagated, as a write is considered successful once a quorum of nodes processes it. MongoDB assigns priorities to nodes, with higher-priority nodes more likely to win the election when the primary fails.

## III. REGULAR KEY VALUE STORES

### A. *Redis*

Redis separates read and write services: writes are handled by masters, and reads by clients, in a "split-brain" model where subsystems are unaware of each other. The infrastructure ensures consistency between masters and slaves, managed by the internal cluster manager, redis-trib. Redis uses hash slots for sharding, where each slot is calculated using CRC16 of the key modulo 16384. A hash slot can contain up to 2160 keys. When a new node is added, slots are reassigned, and clients accessing old nodes will receive a "MOVED" error, directing them to the new location.

Redis is single-threaded, so large requests can delay other client interactions. It uses RESP (Redis Serialization Protocol) for communication and a gossip protocol for node interactions. Redis' performance benefits from pipelining and a Pub/Sub system, allowing clients to receive messages once processing is complete.

### B. *Memcached*

Memcached is an in-memory distributed cache designed to speed up web applications by caching frequently accessed data. It uses a doubly linked list and hashmap for efficient key management, sorted by access time with an LRU caching policy for eviction. Memory is organized into 1MB pages, subdivided into chunks based on slab classes. On write, it finds the appropriate slab for the chunk, and on read, it returns data from memory or fetches it from the database.

However, Memcached allocates a 56-byte header for each key-value object, even for size 0 values, introducing overhead.

### C. *Berkley DB*

Berkeley DB differs from traditional KV stores by providing a library that integrates with applications, rather than relying on a separate server application. It offers efficient in-memory key-value storage, using hash tables or B+trees as the primary data structure. B+trees are used for indexing, enabling fast lookups and efficient indexing.

Berkeley DB provides low-level APIs for replication, starting with all nodes and designating one as the replication manager. Replication is system-wide, not entry-specific. It offers three replication policies: None (no sync), All (sync to all), and Simple_Majority (sync to a subset of nodes). Like MongoDB, it follows a write-to-master, read-from-replica model. It also supports partitioning data across cluster nodes, and when reading, it combines data from multiple parts using indexes to optimize searches, sending the result to the client. Sync parameters, including the option to delay syncing, can be adjusted. For example, if a specific master is expected to handle both reads and writes, the sync to replicas can be delayed.

While Berkeley DB simplifies programming, it may not appeal to developers seeking fully featured solutions. It is better suited for those building custom key-value store systems, especially those familiar with the complexities of highly available infrastructure.

Berkeley DB deviates from the paradigm followed by regular KV Stores by providing a library to link with the applications rather than a separate server application. Berkeley DB is a library which provides efficient in memory key value stores, where keys can be stores in hash tables or as B+trees. BerkeleyDB uses B+trees as their primary data structure and builds indexes around the tree nodes for efficient lookup. Also, B+ tress offer ease of indexing.

## IV. COLUMN BASED STORES

### A. *Cassandra*

Cassandra was developed to meet the need for a fast database for Facebook's Inbox Search. Known for its scalability, it partitions data across cluster nodes using consistent hashing. Like DynamoDB, the nodes are mapped to virtual spaces in a ring, with each space responsible for a set of keys. Load balancing is achieved by moving lightly loaded nodes along the ring to relieve heavily loaded ones.

Each node coordinates the keys in its virtual space and replicates data to N-1 other nodes (N being the replication factor). Cassandra offers replication policies like "Rack Aware," "Rack Unaware," and "Datacenter Aware" for fine-tuning performance. It uses Zookeeper to manage the cluster, similar to HBase.

Cassandra's performance is driven by indexing, utilizing Bloom filters to check for keys. It stores keys in-memory up to a threshold, then writes to disk in a single operation, optimizing disk I/O. While

this experiment focuses on key-value pairs, it's important to note that Cassandra also maintains column indexes for faster response times.

Cassandra uses the gossip protocol to propagate control states across nodes. During bootstrapping, nodes sync data from pre-configured "seeds," which can be set in Zookeeper's config. Read/write requests can be directed to any node. For writes, the responsible node ensures a quorum of replicas confirms success. Cassandra writes data sequentially to disk to boost throughput. Reads are typically routed to the closest replica, though this can be customized. TCP is used for replication and request routing, while UDP handles other messages. Data is stored in blocks of up to 128 keys, enabling fast searches. If the key is in memory, it is returned without disk I/O. Cassandra meets the needs of a highly available key-value store and supports key mutation (updating keys with different values).

### B. *HBase DB*

HBase is a distributed, scalable big data store built on top of Hadoop, offering strong consistency. Its deployment includes the HMaster (manages metadata and table sharding), Region Servers (store data in memory and sync to HDFS), and Zookeeper (manages region servers and clusters).

HMaster maintains a META table, helping clients locate the correct region server for a specific key. Clients query the table for the server name, region identifier, and start key, allowing them to determine the key range and contact the appropriate server. When a region server syncs data to HDFS, it follows Hadoop's default replication factor of 3.

When a region server's table size reaches a threshold, keys are sharded to servers with lighter loads, with Zookeeper handling load balancing. Each region server also acts as a Zookeeper peer, forming a quorum. Each region server keeps a write-ahead log (WAL) file, which is replicated to slave clusters.

HBase processes every request as a MapReduce job, handled by the region servers for the relevant data chunk. It offers two caching techniques: LruBlockCache (default, using heaps) and Bucket Cache (since HBase 0.98.6). Bucket Cache reduces fragmentation, while LruBlockCache provides faster response times. While HBase requires deployment expertise, it offers a robust distributed solution as a scalable key-value store.

## V. FEATURE COMPARISON

### A. *Replication Features*

The table below sheds light on the important factors that determine an efficient replication policy.

TABLE II. **PROPERTY SURVEY**

| KV Store | Granularity | Node Discovery Protocol | Consistency |
|---|---|---|---|
| **Riak** | Per bucket (set of keys) | Gossip Protocol | Eventual |
| **Redis** | Master-Slave Replication (Node to Node replication) | Gossip Protocol | Eventual, Strong |

| | | | |
|---|---|---|---|
| **MongoDB** | Master-Slave (Node to Node replication) | Proprietary Replication Protocol | Strong |
| **HBase** | Per table replication | Uses Zookeeper | Eventual, Strong |
| **Cassandra** | Per table replication | Gossip Protocol | Eventual |
| **Berkeley DB** | Node to Node replication | Two-phase voting protocol over TCP/IP | Strong |
| **Memcached** | Per keyspace (set of tables) | Servers do not communicate, client does. | Client Specific |

TABLE III. **PROPERTY SURVEY EXTENDED**

| **KV Store** | **Replication Policies** | **Read Concern** | **Write Concern** | **Cluster Manager** |
|---|---|---|---|---|
| **Riak** | Master-less multi-site replication | Yes | Yes | In-Built |
| **Redis** | Master-Slave | No. | No | In-Built |
| **MongoDB** | Range based, Hash based and Location-aware sharding | Yes | Yes | In-Built |
| **HBase** | Cyclic (master to master), master to slave. | Yes | Yes | Zookeeper |
| **Cassandra** | Rack Aware, Rack Unaware and Datacenter Aware | Yes | Yes | In built |
| **Berkeley DB** | Master-Replica | Yes | Yes | None |
| **Memcached** | LRU Caching policy | No | No | None |

While Redis, MongoDB and Berkeley DB support node to node replication of complete data set, Riak, HBase, Cassandra and Memcached give one further abstraction to find tune the replicas per table (keyspace). Most of the key value stores Gossip protocol as their control state protocol and offer eventual consistency. MongodBN and Cassandra support different replication policies giving clients a flexibility to fine tune their deployment. Redis, MongoDB, HBase, Cassandra and Berkeley DB offer the capability to replicate/read data to/from a quorum of N nodes (out of the replication factor R) and consider the request a success. This certainly improves the overall response time from the client's perspective.

### *B. Memory Features*

Caching is crucial in key-value stores as it improves hit ratios and reduces I/O overhead. While most key-value stores use the Least Recently Used (LRU) policy, some adopt write-through or copy-on-write strategies. Batching requests further enhances memory performance by fetching multiple records in a single I/O operation. Built-in pipelining also boosts overall performance, increasing the number of

requests handled per second. Though sharding is a key feature of parallel file systems, some key-value stores also implement this approach.

TABLE IV. **MEMORY SURVEY**

| **KV Store** | **Caching** | **Request Batching** | **Pipelining** | **Publisher / Subscriber Model** | **Sharding Policy** |
|---|---|---|---|---|---|
| **Riak** | Copy on write | Yes | No built-in pipelining | No such feature | No sharding |
| **Redis** | Write through | Yes | Yes (Mass Insertion) | Yes | Yes |
| **MongoDB** | No caching | Yes | Yes (Aggregation Pipelining) | Yes | Yes. Data is sharded to 64MB chunks. |
| **HBase** | Bucket Cache and LruBlock Cache | Yes | Yes (Pipelined Scanning) | By integration with Kafka | Yes. Per chunk size is 64MB. |
| **Cassandra** | Built in key and row caches | Yes | No built-in pipelining | By integration with Zookeeper | No |
| **Berkeley DB** | LRU | Yes | No built-in pipelining | No such feature | No |
| **Memcached** | LRU | Batched Get but no Batched Set | Clients can build pipelining support. | No such feature | No |

### *C. Key Operations*

Next, we explore the operations with respect to key operations phenomena's like indexing, range keys, auto key expiry (to garbage collect), key mutation operations, link walking or in other words cursors to iterate over a contiguous set of keys.

TABLE V. KEY OPERATIONS SURVEY

| KV Store | Range Queries | Auto Key Expiry | Key Mutations | Link Walking Or Cursors | Secondary Indexing |
|---|---|---|---|---|---|
| Riak | Yes | Yes | No | Yes | Yes |
| Redis | Yes | Yes | No | Yes | Yes |
| MongoDB | Yes | Yes | No | Yes | Yes |
| HBase | Yes | Yes | Yes | Yes(Scanners) | Yes |
| Cassandra | Yes | Yes | Yes | Yes(Auto paging) | Yes. (Not recommended though) |
| Berkeley DB | Yes | Yes | Yes | Yes | Yes |
| Memcached | Yes | Yes | No | No | No |

## VI. EVALUATION

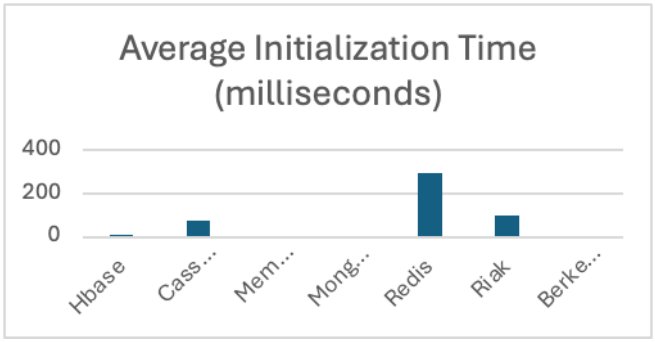
This paper evaluates the key value stores on a cluster of 20 nodes with 8 cores each with a RAM size of . For all above mentioned key value stores, we use 4 nodes as servers and range the number of clients from 16 to 128 to stress the overall system. We measure performance across 4 verticals by scaling clients, scaling the data from small requests of large-to-large number of requests of large sizes. We vary the sizes from 2KB to 2GB per client. We first calculate the initialization time for every key value store to compare the bootstrapping scenarios for each of them, then we perform writes followed by reads. We use MPI infrastructure to scale the clients from 16 to 128 such that each client process is allocated a dedicated core to complete allocation.

### A. *Client Connection*

The time taken to establish a connection to a key-value store is an interesting scenario, as we want to compare how quickly clients can connect and begin performing operations. The following chart shows the average connection time in milliseconds along the X-axis. Notably, Redis, Riak, and Cassandra took significantly longer than other key-value stores. Cassandra shows a spike, as each time a client connects, its information is sent to the seed provider (or master), requiring additional message exchanges before the client receives a connection handle.

Redis experiences delays due to several special checks when a client connects: it changes the client socket to non-blocking, modifies the TCP connection to set the TCP_NODELAY flag, and creates a libevent handle for each connection/socket. These steps contribute to the spike in Redis' connection time.

Riak, being a non-location-aware key-value store, randomly selects a node for each client request. Additionally, Riak communicates with clients over HTTP, which introduces extra overhead, especially considering its design for web applications.




*B. Scaling Clients (Writing Large Requests)*

Following figures represent a scenario where each client is writing 1 key with 2GB and 2 keys with 2GB values each. This experiment is done in order to stress the file system in terms of memory and then measure performance.

Note: HBase, Cassandra doesn't support values of this size and hence is not shown in the figures.

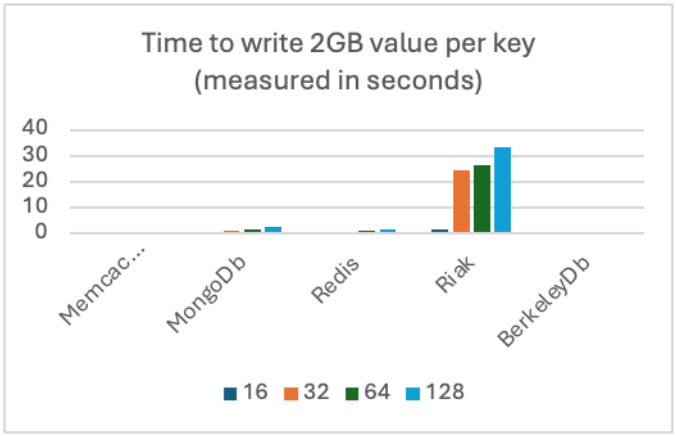



Fig. 1. **Time to write 2GB value per key**

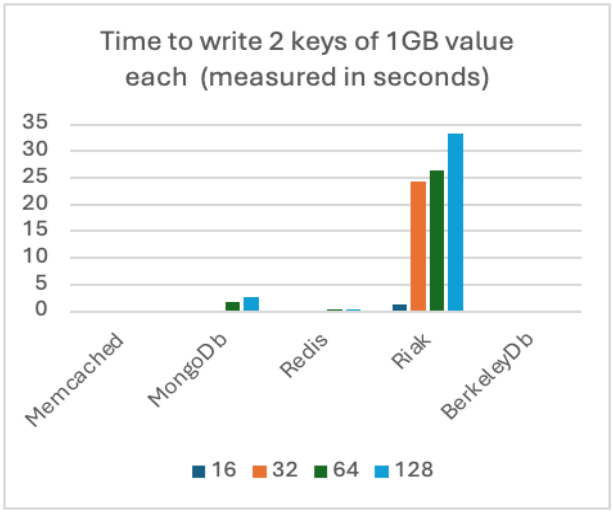



Fig. 2. **Time to write 2 keys of 1GB value per key**

BerkleyDB being a library and real close to the native OS calls, undoubtedly is expected to outperform the other stores as it doesn't involve any fancy overheads apart from replication. Riak on the other hand is not able to scale well in such scenarios. One probable solution to this would be sharding the data to multiple nodes.

### C. *Scaling Clients (Reading Large Requests)*

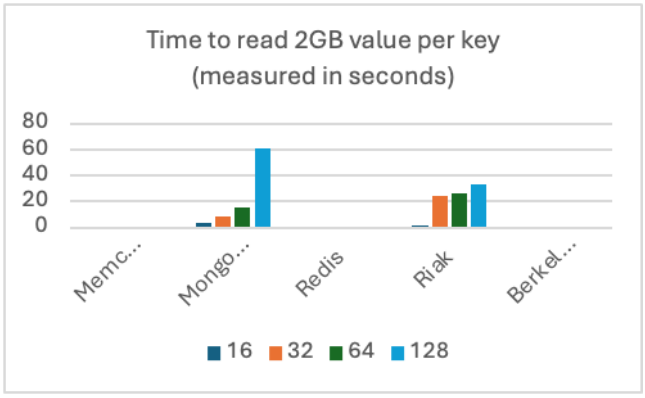



Fig. 3. **Time to read 2GB value per key**

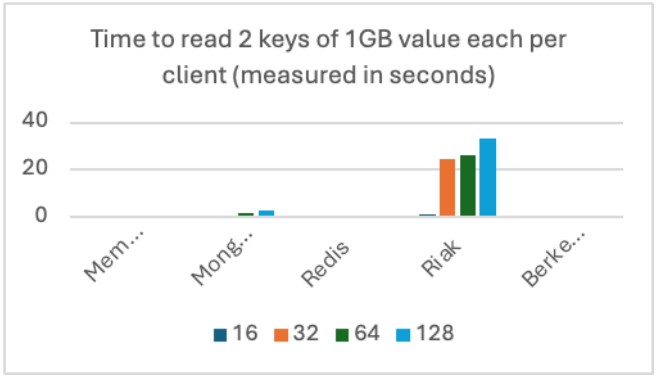



Fig. 4. **Time to read 2 keys of 2 GB value each per client**

In the above tests, MongoDB struggled to scale beyond 2GB workloads but performed well with 1GB requests. Riak, being a library closely integrated with native OS calls, is expected to outperform other stores due to its minimal overhead, aside from replication. However, Riak did not scale well in these scenarios. One potential solution to improve its scalability would be to shard the data across multiple nodes.

### D. *Scaling Clients (Writing large number of small requests)*

We performed three experiments as part of this scenario:

- Each client writing 1024 requests of 1KB values each.
- Each client writing 2048 requests of 256 bytes each.
- Each client writing 20480 requests of 128 bytes each.

The results show that while other key values stores produce competitive timings, Riak is slower by an order of magnitude. Overall, we can conclude that the architecture of riak seems to have an overall overhead when compared with other key value stores.

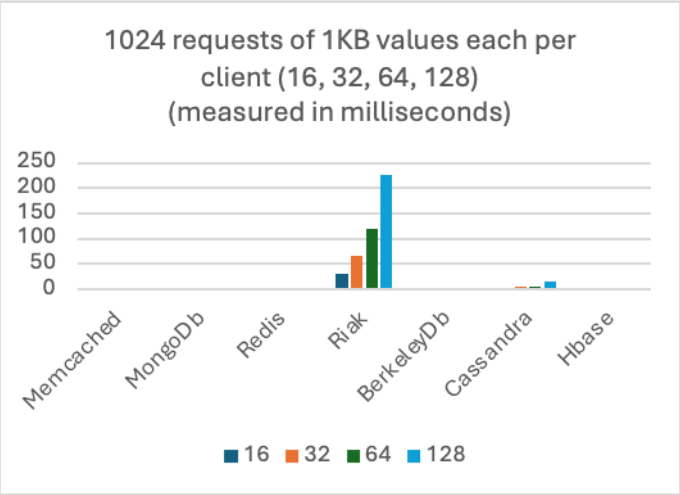



Fig. 5. **Scale testing**

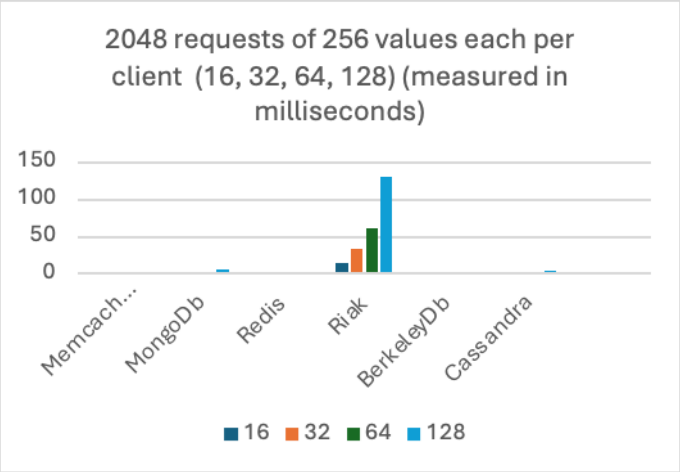



Fig. 6. **Scale testing of values**

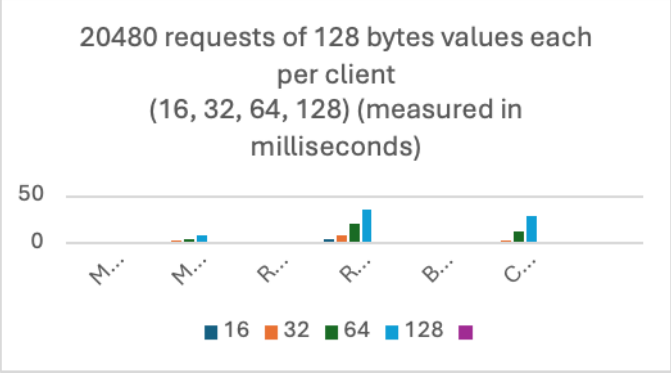



Fig. 7. **Scale testing of 10X**

*E. Scaling Clients (Reading large number of small requests)*

*Like writing, we repeat the following experiments for reading:*

- Each client reading 2048 requests of 256 bytes each.
- Each client reading 20480 requests of 128 bytes each.

While the numbers for Riak were in order of magnitude worse than the order stores, while reading MongoDB took a lot of time. This primarily is because of the way a search is performed on the documents. There is a lot of metadata associated with each small document which wastes time and effort of this document-based store. Note these results were taken with indexing enabled, with indexing disabled MongoDB shows even worse performance.

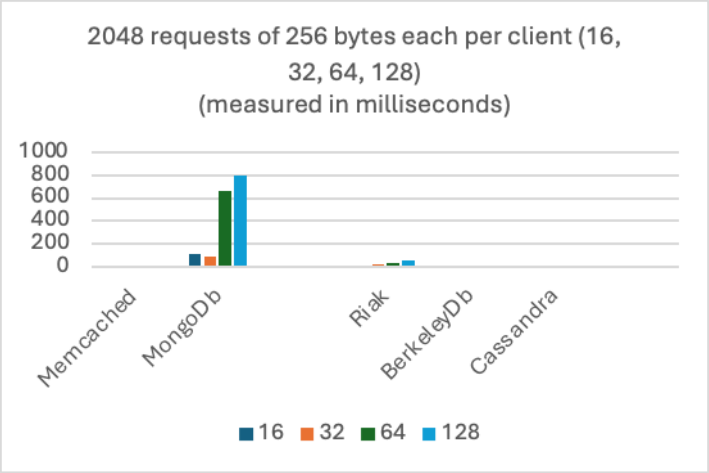



Fig. 8. **Scale testing constrained at bytes**

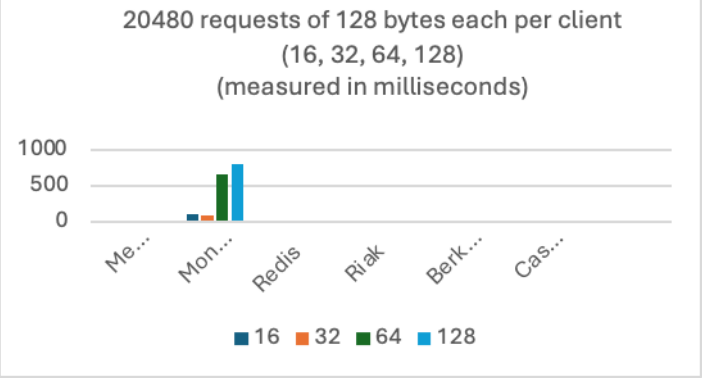



Fig. 9. **Scale testing at 128 bytes**

## VII. CONCLUSION

Key-value stores are becoming increasingly important for the workloads generated by modern applications. Choosing the right store based on data access patterns is crucial. After reviewing the characteristics of various key-value stores, we can identify key features that form the foundation of a highly available and scalable distributed key-value store. These features include sharding, load balancing, key-level and column-level indexing, the use of bloom filters to check if a node contains a key, support for batch requests, eventual consistency, and intelligent load balancing. Allowing clients to customize

the infrastructure based on their needs (such as in the cases of BerkeleyDB and Memcached) can simplify the system and reduce overhead.

Reliability and scalability are essential, as increased data and operational traffic lead to more frequent failures. Over time, key-value stores have evolved to handle these challenges and offer distributed, replicated infrastructures that can withstand faults. Another notable feature is flexibility—clients avoid delays caused by POSIX I/O, and applications can run the same code across heterogeneous systems. With the rise of REST-based frameworks, many key-value stores are adopting standardized client interfaces. The next wave of key-value stores will focus on providing user-friendly interfaces (e.g., stores that run web servers for browser-based visualization) and offering solutions tailored to the specific needs of different application types.

## REFERENCES

[1] A. Lakshman and P. Malik, "Cassandra - A Decentralized Structured Storage System," *Facebook Inc.*, 2009.

[2] K. Seguin, *The Little Redis Book*, 2011.

[3] "Memcached," [Online]. Available: https://memcached.org/. [Accessed: May 5, 2025].

[4] "Redis," [Online]. Available: https://redis.io/. [Accessed: May 5, 2025].

[5] "Riak," Basho Technologies. [Online]. Available: http://basho.com/. [Accessed: May 5, 2025].

[6] *Apache HBase Reference Guide*, Version 2.0.0, The Apache Software Foundation, 2018.

[7] *MongoDB Architecture Guide*, MongoDB Inc., Version 3.2, 2016.

[8] M. Seeger, "Key-Value Stores: A Practical Overview," *Computer Science and Media*, 2011.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," in *Proc. ACM SIGMETRICS*, 2012, pp. 53–64.

[10] G. DeCandia *et al.*, "Dynamo: Amazon's Highly Available Key-value Store," in *Proc. 21st ACM SOSP*, 2007, pp. 205–220.

[11] T. Cowsalya and S. R. Mugunthan, "Hadoop Architecture and Fault Tolerance based Hadoop Clusters in Geographically Distributed Data Center," *International Journal of Computer Applications*, vol. 89, no. 3, pp. 1–6, Mar. 2014.

[12] N. Yuhanna, "The Forrester Wave™: NoSQL Key-Value Databases, Q3 2014," *Forrester Research*, Sep. 2014.

[13] D. Cronin, "A Survey of Modern Key-Value Stores," Stanford University, Technical Report, 2013.

[14] *Berkeley DB Tutorial and Reference Guide*, Version 4.1.24, Oracle Corporation, 2003.