KV Cache Store: Implementation Upgrades

Alice Ao
Yale University

Benjamin Mehmedovic
Yale University

Gabriel Dos Santos
Yale University

1 Introduction

Key-value stores are a fundamental building block in modern distributed systems, providing a simple and efficient way to store and retrieve data. In recent years, the popularity of key-value stores has increased significantly due to their ability to scale horizontally, tolerate failures, and support high-performance read and write operations. However, there are multiple different design strategies and upgrades, which our project aimed to explore.

This paper highlights our implementation of an enhanced key-value store that builds on top of the sharded and replicated store we implemented in Lab 4. Our upgraded key-value store supports various data types such as strings, lists, sets, and sorted sets, and provides two new functions: MultiSet and Compare-And-Swap. This paper will explain the implementation, design decisions, and trade-offs in the development of the upgraded key-value store.

2 Related Work

Our design was heavily inspired by Redis, a popular NoSQL key-value store with support for data structures, such as List, Set, and SortedSet [4]. In addition to having a standard Set function, Redis provides a MSET/MultiSet function that reduces traffic, as a user can set multiple key-value pairs at a time [1].

In practice, key-value stores for large distributed systems, such as Amazon's Dynamo, often prioritize availability over consistency. Fault tolerance and reliability are critical to Amazon's e-commerce operations, as it has millions of customers across the world and tens of thousands of machines that collectively experience frequent failures [3]. Therefore, the priority of availability inspired some aspects of our design, such allowing for partial node failures and copying over shard information.

It is important to note that simple key-value databases are not the only database/data model for handling large amounts of distributed data. Other designs define specific relations and properties between data. Column-family databases, such as Cassandra, organize big data into multi-dimensional tables and allow for more complex querying [2].

3 Implementation Details

3.1 Data Types

We implemented three new data types: List, Set, and Sorted Set, with corresponding methods like Create, Append, Remove, and Check. We also added test cases to confirm the proper functioning of these types.

3.1.1 List

While designing the list data type in the backend, many different data structures were considered. Here are a few of the options with their pros and cons in terms of time complexity.

1. **Slice**: The simplest option would be to use the Golang slice. With no additional optimizations the expected runtime would be $O(1)$ append, $O(n)$ remove, and $O(1)$ get.

2. **SortedSlice**: By using an efficient sorting algorithm such as QuickSort, the list can remain sorted in the key-value store. This comes at an additional cost for insertion but gain on removing elements: $O(log(n))$ append, $O(log(n))$ remove, and $O(1)$ get.

3. **Map**: By using hashing in a map data structure, we achieve theoretical constant time for insertion and deletion of elements within the list but must transform the entire structure if the user desires all entries in list format. $O(1)$ append, $O(1)$ remove, $O(n)$ get.

In the end, we decided to implement the simple Slice version of the data structure. Besides being great for dominantly append workloads, it's also able to maintain the original order of requests. This becomes useful when a user wants to deal with chronologically ordered logs. This also makes it such

that the PopList procedure works like a queue. Having the items sorted by alphabetical order would come with the trade-off of losing chronological order which we believe was not worth it for our users.

3.1.2 Set

The Set ensures that all inserted elements have no duplicates. It does not require the chronological order of elements to be maintained. During the process of designing the set, 2 main options were considered being a SortedSlice or Map.

The design trade-offs are almost identical to those outlined for the List data type. However, we did not consider a regular Slice at all for the set implementation because sets must ensure the constraint that all elements are unique. This would require an $O(n)$ time on insertion to check for duplicates thus making it slower in every aspect.

Ultimately, we decided to use a map as the back-end implementation for the Set as it has constant time Append, Check, and Remove operations. (We don't expect the workloads to use Get requests often.)

Additionally, we wanted to compare the performance benefits of List and Set and see whether the constant-time Append and Check operations result in significant improvements in latency. To do this, we modified the stress tester to run multiple concurrent requests for Append and Check. Using the flags, a user running the stress tester can now disable the Get and Set queries, disable the correctness checking, and choose whether they'd like to test Append and Check with a List or Set.

The results of our stress tester experiments are displayed in Figure 1. We had Checks as about 75% of the QPS in each experiment, and we tested a variety of total QPS for both lists and sets. We found that the latency first decreased for both lists and sets as QPS increased, up to about 2,000 total QPS. This was consistent with our experimental results from Lab 4 and may be simply due to how requests can be run concurrently. However, when the total queries begins to exceed 12,000, there is a significant increase in latency for both list Checks and Appends. There is only a modest increase in latency for set Checks and Appends, even as we increased the total QPS to nearly 40,000. Therefore, if there are many queries per second, or the system is handling a large volume of data accesses and additions, it is beneficial to use a set data type. Our results also revealed that surprisingly, the Append latency is higher than the Check latency for lists with high workloads. This could be due to a variety of reasons, such as scheduling or unexpectedly high runtimes for the append function. The process of allocating new memory for each element and adding it to the list may increase overhead and latency.

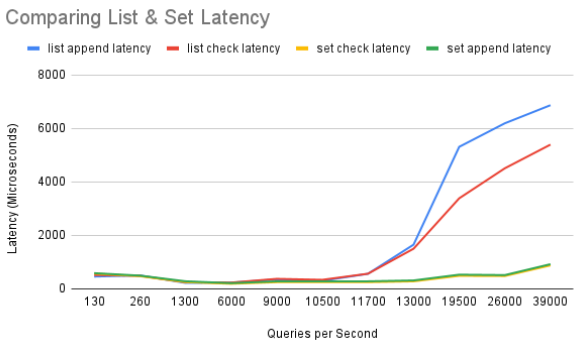

Figure 1: Stress Tester List and Set Latency Results

3.1.3 Sorted Set

The Sorted Set implementation was heavily inspired on the Redis sorted set APIs. We imported a SortedSet Golang library which implements similar functions such as AddOrUpdate and GetRangeByRank. At high-level, users are able to store elements with an associated score. Users can then retrieve elements based on their ranking order.

1. **AppendSortedSet**: Adds an element into the sorted set with specific value and score. If the element is added or updated, this method returns true. This is a $O(log(n))$ operation since the elements are maintained in order.

2. **GetRange**: Get elements within specific rank range [start, end]. Note that the rank is 1-index with 1 being the first node with lowest score. Time complexity of this method is $O(log(n))$ as well.

3.2 System Design

The system optimizes and reduces lock contention by separating out the key-value pairs based on data type and shard. When first processing a request, we determine which shard the key belongs to using a common hashing function. Each shard has separate sections for strings, lists, sets, and sorted sets that can be independently accessed. Therefore, each node can theoretically process $M * N$ concurrent requests where M is the number of data types and N is the number of shards.

3.2.1 Enhanced Shard Migrations

The Key Value store was improved by adding streaming during shard warm-ups. Previously, warming up involved sending the entire key-value store in unary RPC requests. By converting to streams, we can process smaller components as they come. We decided to stream the database based on a key-by-key basis. For every key in the shard that needs to be copied,

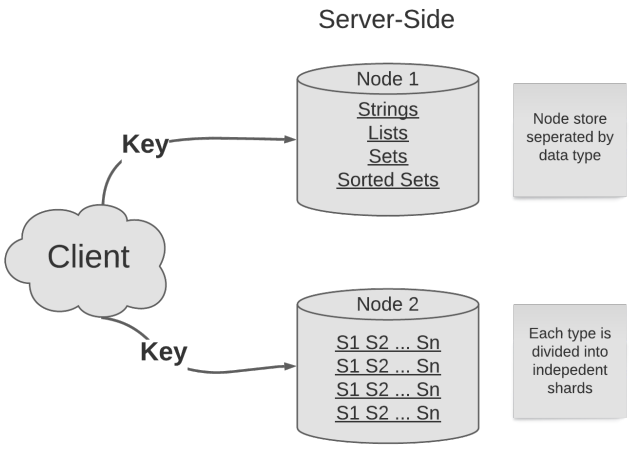

Figure 2: This outlines the structure of the database. Keys can map to multiple nodes for fault tolerance. Each node's store is divided into 2 levels. The first level is based on datatype and the second level is based on shard.

`GetShardContents()` sends the corresponding value immediately without waiting for the remaining keys on the shard to be discovered.

3.3 New APIs

3.3.1 MultiSet

Similar to the Redis function, MultiSet allows users to set multiple key-value pairs with a single function call. The advantage of MultiSet could be similar to the "batching" advantage in Lab1, as multiple requests to an underlying call can be combined into a single call. Regular Set can result in lock contention when several key-value pairs belonging to the same shards/nodes are set. The node's map must be locked, updated, and unlocked for each shard. MultiSet therefore reduces contention by allowing the map to be locked and unlocked just once for every shard-node pair, no matter how many key-values there are.

Our implementation of MultiSet guarantees *per-shard atomicity*, as our helper MultiSet function sets all of the key-value pairs belonging to the same shard in the same call. Guaranteeing complete atomicity, as redis does, would involve significantly more overhead and latency, as we would need to have some sort of mechanism for undoing MultiSets and restoring their former values if a MultiSet to one shard fails. Alternatively, we would have to lock the entire shard set each time MultiSet is called, using a coarse-grained lock that could result in significant contention.

One possible strategy we considered is enforcing that all keys in a MultiSet call belong to the same shard. However, we felt that this could be difficult, as the user would need to have some knowledge of which keys belong to which shards. We could possibly have implemented a client-facing function that provides information on the key-to-shard mappings, but this would still complicate the client side, as each time the client calls MultiSet, they would have to call this function as well.

We reasoned that it would be much simpler and more consistent with the original lab to have this logic occur in the client and server functions themselves. Therefore, our internal client and server functions are able to handle calls with keys stored on different shards. While this does have the trade-off of not guaranteeing complete atomicity, and therefore resulting in partial failures, it allows the key-shard mappings to be hidden from the user. It also still allows us to keep our current fine-grained locking technique that only locks the current shard.

Additionally, we decided to return the list of failed keys to the user, which can help them address partial failures on their own end. The user can call MultiSet again with only this list of failed keys. Our implementation of MultiSet includes duplicates in the list of failed keys, but given that calling MultiSet with duplicate keys does not change its behavior or the key-value store, we decided to leave the list as-is.

There are some edge cases associated with MultiSet that we decided to handle in the following ways:

1. *Same keys in a MultiSet call*: If duplicates of the same key are given to the same MultiSet call, only one of the values will be saved as the value associated with the key. The values will overwrite one another. We decided to just allow overwriting given that the regular Set function also overwrites keys.

2. *Different number of keys and values*: If the client does not provide an equivalent number of keys and values, the MultiSet call will return an error. This was the most intuitive way of handling this case, as there is no good way for the server to determine which key should be associated with which value.

3. *Partial failures for MultiSet*: In the case of partial failures, in which some nodes may be unavailable and therefore fail to set the key-value pairs, the pairs will still be set on the available nodes. This is consistent with how Set handles partial failures. Additionally, given that GetShardContents helps restarted nodes copy over the key-value pairs they've missed, this implementation strategy may improve the overall availability of the system at the tradeoff of consistency.

Finally, we also wrote both unit and integration test cases to check the correctness of our MultiSet implementation and confirm that it displayed expected behavior under various conditions and edge cases.

3.3.2 Compare-And-Swap

Finally, an additional Compare-And-Swap API was added to strings in the key-value store.

In one atomic operation, CAS swaps the value of the key given that the current value matches with the expected value request parameter. It returns true when the expected value matches with the present value and returns false when either no value exists or values don't match. CAS has the same time complexity as the original set operation.

The CAS API serves as a great advantage to clients in the face of concurrent requests and unexpected failures. For example, many clients may send concurrent requests to modify the same key. A particular client may only want to change the key-value pair under certain conditions when the client's expected value match with the current key's value. The benefits are similar in the instance where a method fails to execute and future code execution relies on the failed value being set. Overall, CAS works to provide clients in the distributed environment.

4 Proof-of-Concept Deployment

Instead of just running the key-value store locally on our machines, we also wanted to deploy our store and test it on multiple nodes. At first, we attempted to deploy it through the Zoo, but due to encoding and permissions issues, we could not properly compile the files. Instead, we successfully deployed the system on two different machines, with one hosting the client and one node and another hosting a different node, using hard-coded IP addresses.

We ran simple commands to test our key-value store and then the original full stress tester from Lab 4. We found that the system successfully was able to compute between the client and the nodes, and also migrated shards when necessary (e.g. in the case of one node coming online after some downtime).

5 Conclusion

5.1 Limitations

While our KV store exhibits significant upgrades from the original KV store in Lab 4, there are important limitations and areas of improvement to note. One limitation is that all elements within the collection data types (list, set, sorted set) have the same time-to-live. There may be use cases in which some elements of a list should have shorter or longer TTLs, but right now, our implementation removes the entire list simultaneously.

Another minor limitation is that we could not unit test shard migrations. The original testing suite mocks the servers and doesn't actually send RPCs, but with our implementation, the Golang protobuf creates a stream only when RPC is sent which proves complicated to replicate in a mocked setting. Additionally, our shard migration with multiple data types serves as a proof of concept for future additions. Right now strings, lists, and sets are migrated. However, due to the complexity of the RPC call that would be required to migrated a sorted set, we decided to omit it for the scope of this project. In the future, manipulations to the protobuf and `GetShardContents()` function would enable copying sorted sets.

Finally, while we did implement streaming during shard migrations as opposed to unary RPC requests, our implementation does not completely utilize the potential benefits of this feature. Since we lock all RPCs until a migration is completely, we cannot serve requests pertaining to keys that are actively being copied over. However, streaming would enable us to serve requests for a key right after it is copied over, even before the shard migration is complete. A refactoring of the code would easily implement this and would undoubtedly lead to a performance boost.

5.2 Future Work

Considering our limitations, there are multiple potential tasks for future work.

The primary extension that would complete our key/value store and further optimize it would be to enable sorted set copying. Additionally, given that the shard migration framework now supports gRPC streams, refactoring the code to support servicing requests on partially copied shards would decrease waiting time for clients and also minimize server load after it completes a shard migration.

Other future work could include adding better support for users working with advanced data types. For example, right now, it is up to the user to properly create lists and sets and add/remove nodes. While we do return the TTL using the GetList and GetSet functions, it would be better to create separate functions that allow the user to check whether a key is still valid, and if so, how much time is remaining for the list/set. Updating the TTL of the list/set may also be a worthwhile addition, as the user may add elements to the list and want the list to persist longer.

As mentioned before, our implementation of MultiSet differs from redis's MSet because it does not guarantee full atomicity. While it would likely be a significant design challenge to restore previous data in the case of a partial failure, the benefits of full node atomicity could make it worth implementing stronger atomicity guarantees for the future. Finally, we could also modify the stress tester to run experiments for the performance of MultiSet vs. Set.

5.3 Final Thoughts

Overall, this project helped us think more critically about designing key-value stores for distributed computing. We con-

sidered multiple data structures as we added new data types, and we compared the performance of two of our data types by modifying an existing stress tester. This component of our project therefore allowed us to not only confirm our existing ideas about the best use cases for different data types but also led to unexpected findings, such as the time complexity of our Append function for lists. We also learned how to modify the existing stress tester for the purpose of running experiments and comparing latency.

Additionally, we gained experience in thinking about tradeoffs between design strategies and guarantees by implementing our new APIs, as well as defining our own RPC functions in protobufs. Our challenges with the shard migrations unit tests also helped us learn how to closely review tests and see how mocks work in unit testing. Ultimately, this project was a great way to not only combine our newfound collective knowledge on distributed systems but also learn new practical skills for distributed design.

6 Acknowledgements

We would like to thank Professor Yang and the ULAs, Ross and Matt, for their help throughout this entire course. We would also especially like to acknowledge Xiao for his extremely detailed feedback and answers to all of our questions. For instance, many of our implementation details, such as per-shard atomicity for MultiSet, were inspired by his advice.

References

[1] Cristian Andrei Baron. NoSQL Key-Value DBs Riak and Redis. *Database Systems Journal*, 6(4):3–10, May 2016.

[2] Artem Chebotko, Andrey Kashlev, and Shiyong Lu. A big data modeling methodology for apache cassandra. In *2015 IEEE International Congress on Big Data*, pages 238–245, 2015.

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[4] Dirk Eddelbuettel. A brief introduction to redis, 2022.