Implementation and Evaluation of a Durable Distributed Key-Value Store[*]

Ge Song
Electrical and Computer Engineering
Carnegie Mellon University
gesong@andrew.cmu.edu

Yihua Pu
Electrical and Computer Engineering
Carnegie Mellon University
yihuap@andrew.cmu.edu

ABSTRACT

In-memory key-value store is a common technique to reduce the workload on backend databases. In this project, we implemented a prototype of a scalable distributed key-value store with some level of durability. Our system distributes the storage with consistent hashing and persists the states with write-ahead log.

After implementing a baseline prototype, we evaluated the effectiveness of common optimization techniques widely used in other distributed key-value systems. The first part of evaluation focuses on how the number of virtual nodes affects the load distribution of our system, and we found that a wise configuration of virtual nodes can effectively improve load balancing of storage. In the second stage, we evaluated common optimization techniques on a single node, which could further improve the throughput of the system.

Keywords

Key-value Store, In-memory Store, Internet Services, Distributed Systems, Caching Systems

1. INTRODUCTION

Modern internet services, such as e-commerce and social media platforms, require low latency and high throughput to satisfy users. Traditional disk-based relational database systems, however, cannot always fulfill the latency and throughput requirement in a large scale, because all the requests are directed to a single machine and cause expensive disk I/O. One solution to eliminate this bottleneck is to scale out the storage systems, and distribute the storage and request workload among several machines. Therefore, companies like Amazon, Google, and Meta have developed their own key-value store or caching frameworks, where the system can easily use keys to partition the storage and route the request to proper machines.

There are many researches focused on the optimization of these key-value systems. To review and evaluate the effectiveness of the ideas and techniques covered in previous researches, this paper will present our implementation of a distributed in-memory key-value systems, and evaluate the performance improvement brought by different optimization techniques. Consider the short time we have to implement and evaluate this project, our goal is not to beat any existing framework. The evaluation part of this project aims to provide insights on the effectiveness of different optimizations for future implementations of industry-level scalable systems.

Similar to Memcache[4], we apply consistent hashing in our systems to balance the workload among different nodes and experiment several single-node optimizations to further improve the performance. However, our systems is not only built to serve as a layer of cache between front-end servers and database systems, but also provide some level of durability for non-critical but popular data through write-ahead log. For example, it's common for a video sharing platform to present a list of most popular videos on its homepage. The data of this list will be frequently loaded for every main page request, but the data is not critical, since it can be rebuilt from the records in database, and the application can tolerant some inconsistencies and inaccuracies in this ranking. With our systems, these data can be accessed within memory efficiently, and the failures can be recovered with best effort through the logs without inquiring and merging data from the database.

2. RELATED WORK

Our work is inspired by some open-source frameworks and research papers about distributed key-value stores covered in this semester's discussion. Some systems build NoSQL distributed databases that partition data into shards by the key, while the others keep all items in the memory for efficient lookup and the systems cache the popular data to reduce the workloads on the backed relational databases.

Dynamo [2] is a highly reliable and scalable distributed key-value database developed by Amazon to support the storage of Amazon's E-commerce platform and Amazon Web Service. In Dynamo, consistent hashing with

[*]This report is a summary of our semester project in CMU graduate course *Internet Service (18-845)*

fixed-size partition is used to evenly distribute storage, while a configurable quorum protocol is designed to read and write between replications for applications with different requirements of availability and consistency. Compared to Dynamo, our system stores all data in the memory to achieve high read and write throughput. Also, the original Dynamo paper relied on decentralization protocol to maintain the metadata and membership in order to eliminate a single master keeping all the information about the cluster. In contrast, we implement a centralized master to simplify the protocol of membership change and eliminate the membership metadata needed to be maintained by every node.

Memcache [4] is a distributed in-memory key-value cache developed by Meta to reduce the read-dominant workload generated by their social network applications load on backend database. Since its applications can tolerate occasionally obsolete data, Memcache trades consistency between cache cluster and database for better performance and cache hit rate. Several inspiring optimizations are done at single-node, cluster, and region levels to fit the scale Meta's product. Our assumed workload and upstream applications are similar, but we introduce the logging mechanism to facilitate the failure recovery without touching the database, while Memcache provides no durability and a newly launched cold cluster needs to retrieve data from warm cluster with no guarantee that every key can be recovered.

RAMCloud [5] focuses on how to efficiently recover storage in a DRAM cloud. RAMCloud applies log-structured storage to recover from failures. A masters logs are scattered and replicated on multiple nodes, and multiple masters can reconstruct the logs to maximize parallelism and reduce the recovery latency. The replica nodes and reconstructing nodes are chosen based on a randomization approach with refinement. Compared with RAMCloud, the log of each node will only be stored locally, since we assumed the disk has relatively low possibility to fail and our systems only store non-critical data that can be rebuilt from other sources with more effort even if the logs are lost.

3. IMPLEMENTATION

Many different implementations and interfaces of the distributed key-value store systems are possible, and the right choice can only be verified by the results in a real system. We use *Golang* to implement a prototype of distributed key-value storage system with *PUT*, *GET*, and *DELETE* functionalities, which enable us to evaluate the effectiveness of different optimization techniques popular in key-value storage systems. We also leave generic extension APIs so that we can easily tune our systems for different requirements and workloads. We choose *Golang* as our development language since it support light-weighted coroutines which can work especially well with highly concurrent workload. Compared with *C*, *Golang* might encounter memory overhead because its inefficient garbage collection algorithm, but we believe *Golang* is good enough for our evaluation purpose, and it can dramatically facilitate our development.

This section will cover the overall system architecture, some implementation details, and the trade-off in our design.

3.1 System Architecture

There are three main components of our systems: client, master server, and cache server. The clients send requests to master or cache servers on behalf of the front-end server needs to access the data. Current supported requests include *PUT*, *GET*, and *DELETE*, and the request can be sent via either *HTTP* long connection or *Golang*'s RPC framework. The master server is responsible for maintaining the membership of the cluster and the hash ring, ranging from 0 to 2^{64}, which consists of the hash identifiers of all cache servers. Each cache server can have several hash identifiers, and these identifiers decide which key ranges a given node is responsible for. Every operation on the key-value dataset needs to inquiry the master server to be distributed to the correct cache server. Given a specific key, the master server can quickly pick the cache server owning this key through the hash ring and consistent hashing protocol. Finally, the cache server maintains a thread-safe concurrent unordered hashmap structure in memory, and handle requests from master and clients.

As shown in *Fig 1*, two modes are supported in our systems to access specific data stored in one of the cache server. In the first mode, the master server serve as the proxy. The master receives *PUT, GET, and DELETE* requests from clients, use the key to locate the target cache server, and send requests to the cache server on behalf of the clients. In the second mode, the clients can alternatively send a *ROUTE* request to the master to get the address of a target server, and fetch the data by itself. Clearly, if all the clients try to access the data via proxy mode, all the traffic needs to go through the master server, which can quickly become the bottleneck. The master server can be optimized to merge several requests being directed to the same nodes within a short period (less than 20ms) into a single batch request. This optimization might slightly increase the latency since every request needs to wait at the master to be merged, but both the request number and traffic can be reduced (i.e. multiple `GET/PUT` requests of a same key can be merged into one). However, the proxy mode is still too expensive to handle all the request. We leave the proxy mode as an option in our system, because we believe the proxy mode can be beneficial for the transaction request that we will support in the future. Distributed

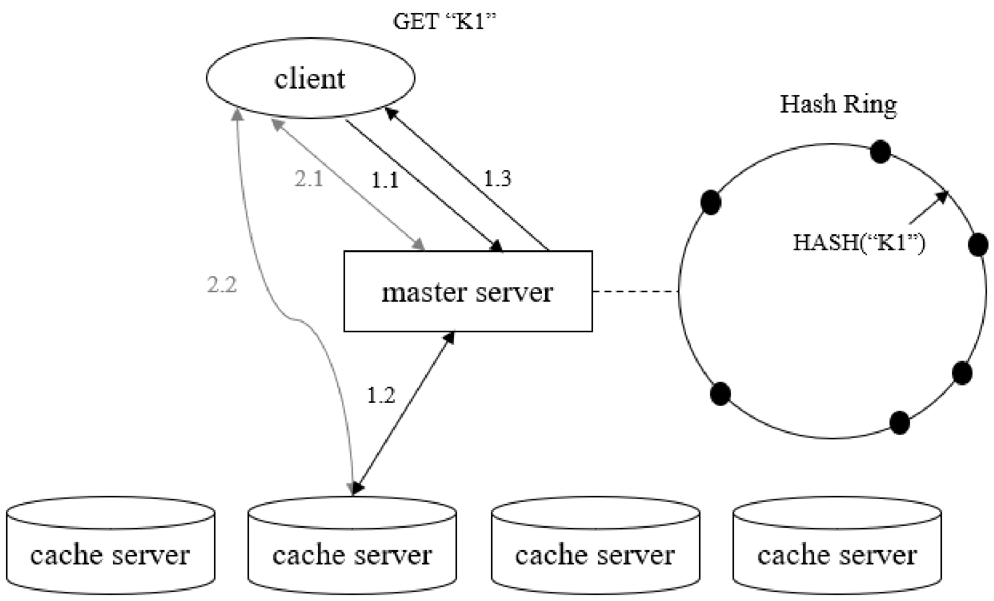


Figure 1: Two access modes are supported: proxy mode and router mode. In the proxy mode, 1.1) the client send request *GET "k1"* to the master; 1.2) the master server use hash ring to pick the proper cache server, and fetch the data on behalf of the client; 1.3) The master return the data to client. In the router mode, 2.1) the client send request *ROUTE "k1"* to master, and the master respond with the address of proper cache server, 2.2) the client bypass the master fetch the item with key "k1" directly from the second cache server.

consensus protocol like two phase commit can be easily implemented in a centralized master without complicating the client library. Also, the proxy mode can let the master server collect the health metrics such as response time from cache server from the normal requests, without sending additional heartbeat signals.

In our design, the cache server is totally unaware of the membership of the cluster. This means any existed key-value cache service with a open network interface can be portable to our systems after registering its service address and supported protocol in the master server. With this design, we can conveniently build a heterogeneous clusters of different cache servers.

3.2 Consistent Hashing and Membership

The master server can map keys to the cache server nodes responsible for them with consistent hashing. The master server maintains a hash ring of the hash identifiers of each cache server. The identifiers can be assigned to each cache server manually by the administrator or randomly by the master server. We believe manual configuration will be a better approach in production, since the administrator can have control over the distribution of the hash identifier, and partition the hash ring evenly with a careful configuration. Meanwhile, a random configuration can simplify the membership changing process, and we use it as the baseline to evaluate the effect of adding virtual nodes. Note that each cache server can be assigned more than one hash identifier on the hash ring, and the additional identifiers are called *virtual node*. For example, in *Fig 2*, there are three cache servers in the original cluster (black nodes on the ring), and each cache server owns two hash identifiers, which indicate the key range each cache server has access to. Cache server *N2* has hash identifiers *vid7* and *vid3*, so *N2* is responsible for the keys in range *(vid2, vid3]* and *(vid5, vid7]* before new cache servers join.

Once receiving a key-oriented request, the master server use efficient non-cryptographic FNV[3] hash function to compute *hash_fnv(key)*, and then execute the binary search on the hash ring to find the hash identifier *vid* next to *hash_fnv(key)* in the clockwise direction. The cache server with identifier *vid* is the target of this request. With proper configuration of the number of identifiers per cache server, it's highly possible that the data are distributed evenly across all cache servers.

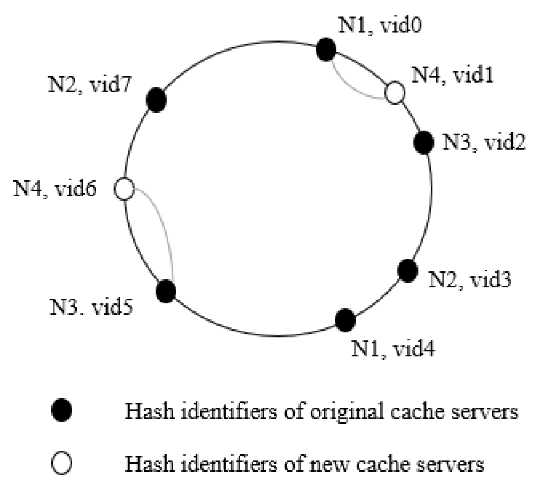
One important advantage of consistent hashing is that only a small fraction of keys need to be migrated when the node joins or leaves since each cache server is only responsible for continuous key ranges before its hash identifiers. *Fig 2* shows when a new cache server joins the cluster, only the keys in two segments of the hash space need to be migrated. Generally, when the *Nth* node joins the cluster, only *1/N* fraction of data needs to be placed to the new node on average. Similarly, when a node is removed from the cluster, only the keys owned by this node are migrated to successor nodes on the ring.

When the system administrator registers a new node, the migration will be executed in the background repeatedly until the migration succeeds. Before the migration is finished, the involved keys can still be accessed on the old nodes, and the old nodes will remove these redundant keys only after they receive a signal from the new node indicating the migration is finished. This design can avoid blocking the new *GET* requests while migration, but the clients might get an outdated cache service address when their *ROUTE* request is served just before the old node remove the involved keys. For *GET* requests, it will not be a big issue, since the clients can inquire the master again when the previous request is rejected by the original node. However, for the *PUT* requests, it will cause keys to be added to the wrong node. We propose two solutions besides blocking *PUT* requests for this subtle scenario. First, the original cache server can record the keys being updated in the first non-blocking migration iteration, and then migrate updated keys during the first iteration while blocking the new *PUT* requests for a short period. Second, we also plan to implement the lazy migration in the future: an involved key can be kept on the original node until there is a *PUT* or *GET* request trying to access it. This lazy migration can also avoid too many keys being migrated at the same time, which can potentially cause network traffic outrage during migration. The challenge is that the master server should be able to track this process to correctly direct the request to the latest address.

3.3 Single Node Cache Server

The cache server maintains an in-memory hashmap, and provide REST APIs that can handle request from clients and the master server to access key-value pair concurrently. We use *Golang's map* as our primary data structure, which solves collision through bucket hashing and resizes based on the load factor. To improve the throughput of the hashmap against concurrent request, we implement a map structure with fine-grained segment lock. The map is partitioned into several segments, and each segment has its own independent readers-writer lock, so that access to a key in a specific segment will not blocking operations assigned to other segments.

Our example cache server is a http server built with *Golang's net/http* package, which is based on multiplexing and coroutines to receive connections and handle requests. We also switch to *HTTP/2*, and we find features like header compression, request pipelining, and



N1, vid0
N4, vid1
N3, vid2
N2, vid3
N1, vid4
N3, vid5
N4, vid6
N2, vid7
● Hash identifiers of original cache servers
○ Hash identifiers of new cache servers

Figure 2: The cluster originally consists of 3 cache servers *N1-N3*, each cache servers are assigned 2 identifiers *vid0-vid7*. When a new cache server *N4* joins, only the keys from *(vid0, vid1]* and *(vid5, vid6]* need to be migrated from *N2* and *N1* to *N4*.

long connection improve the throughput of our cache server by about 12%. Clearly, there is huge space for us to improve the efficiency of our cache server, but we believe the example cache server functional well enough to provide a prototype for us to verify and evaluate different ideas.

Regular cache service also evict less important data to control the usage of memory. However, exploring cache replacement strategies is out of the scope of this report, since we believe the discussion of cache replacement should always be based on different real workloads in production [2, 4]. For example, the widely-used LRU cache replacement policy might works poorly the workload scanning the whole map in a fixed order, since the recently accessed keys is not likely to be accessed again until the next scan. However, we propose that APIs like *PIN* and *UNPIN* for upstream applications can be helpful for in most cases. *PIN* will notify the cache server not to evict a key until this key is *UNPIN*ed. This functionality can explicitly keep several hot keys always in the cache, to avoid a huge number of requests on hot keys being passed directly to the backend databases.

3.4 Fault Tolerance

We mainly consider two ways of achieving fault tolerance in our system: replication and logging.

3.4.1 Replication

Replications is not only a good way of fault tolerance but also a effective way of load balancing, especially for keys with large popularity. We can potentially implement replications at the level of key in the future. One potential solution is let clients *PUT* new key with configuration of replicate factors *r*. Then, the master generating shadow keys by appending delimiter and numbered suffixes to the key until *r* can be distributed to *r* different cache servers. For example, when the master receive request *PUT(key: "k1", value: "v1", replica_factor: 3)*, it will generate shadow keys *"k1_1", "k1_2", "k1_3"*, where delimiter "_" will never appear in normal keys. If any two of these three shadow keys belongs to the same cache server, the master server will keep trying increasing suffix number until keys can be distributed on different cache servers or the number of retry hits the a given limit. In this situation, for a specific key, the system is able to tolerant the failure of at most two cache nodes that contain this key.

When the clients wants to access a given key with replications, the master server will randomly pick a server from cached metadata of shadow keys to parallelize the request to this key, so that the large load on this specific key is evenly distributed to all three cache nodes. If a cache server fails, the master will try to get key from the other two nodes. Generally, with *R* replications for a key, the system can tolerate *R-1* without waiting for the cache server to be rebuilt, and evenly distribute the workload on this key to *R* nodes when all of them are healthy. The disadvantage of this approach is all the *PUT* with replica factor needs to be executed by the master, and master needs additional memory to cache the metadata of the hot keys.

3.4.2 Logging and Recovery Mechanism

Logging provides fault-tolerance to in-memory storage systems. Upon various failures, log data can be used to avoid data loss and efficiently recover the system. This could be very helpful if the cache is used as a middle layer between databases and front-end services, through which hot keys can be fast loaded in to the cache, so the warm-up time is largely reduced.

Our logging mechanism is based on Write-ahead Logging (WAL). While running, each of the cache server will create a log file that tracks all of updates, including their timestamp, key, and value. Upon a *PUT* operation, the update is first recorded in the log file on disk and then applied to in-memory hashmap. However, this synchronous way of writing log introduces a problem: every request that modifies the data needs to first write to the disk to have the log persisted, and this actually contradicts with the nature of a cache, where all of operations should be fast (in our case, all of the operations should happen in memory).

Our workaround is to use asynchronous method for logging, which means logging may not strictly happens before the actual update. Upon a update request, the server first makes a asynchronous call to a log writer thread and continue to update the hashmap without waiting for the log writing. In our implementation, this is achieved through *Golang's channel*, which is basically a pipe that connects concurrent goroutines. The server goroutine is the producer of the pipe, adding logs into it; the log writer goroutine is the consumer of the pipe, taking logs out and writing them to the log file.

In asynchronous WAL, the request doesn't need to wait for disk operation to be handled, so its performance should be comparable with non-logging implementation. However, the performance gain comes with a cost: as the logging is not strictly persisted before the actual update happens, upon a failure, the log and update may both get lost. Therefore, it's really a trade-off between fault-tolerance and performance.

4. EVALUATION

This section focuses on the evaluation of three parts of our design:

- How does the number of virtual nodes (hash identifiers per node) influence the distribution of the data?
- How does the write-ahead log impair the performance?
- How can fine-grained lock improve the throughput of a single cache server?

For the first problem, we simulated the key distribution by deploying both the master server and cache servers locally, adding many new keys to the cache cluster through proxy mode, and getting the numbers of keys being stored on all nodes. Since we only needed to examine the distribution of data after inputting enough key-value pairs, the local simulation would be enough. For the last two problems, we deployed the cache server on the host *AWS EC2 t3.xlarge instance*[6], with 4 vCPUs, 16 GB memory, 5 Gbps network burst bandwidth, and up to 2,780 EBS disk burst bandwidth. The workload was generated concurrently by our local clients, and the average latency between clients and cache service was about 80 ms.

4.1 Workload Pattern

Our dataset [1] is the Google Trends of countries all around the world from 2001 to 2020, which is presented by Kaggle as a public dataset. For preprocessing, we extracted only the search keywords and removed all of the duplicates. The final datatset is a list of 18,430 unique strings in a JSON file.

We synthesized two workload patterns with the key set: Read-Intensive and Read-Write-Intensive. For both patterns, a clients was assigned with a random subset of the dataset to read or update. In the Read-Intensive pattern, the cache cluster is preloaded with a configurable number of keys, and each client will read all keys in a random order. The Read-Write-Intensive pattern was aimed to mimic the common execution orders of a key updates in the real world. A client would firstly read the current value of a key, update the old value to new value, and submit the new key-value pair to the cache cluster. Finally, the client might read the key again to check whether the update is made successfully. It is worth to mention that this read-write-read operation is not atomic. When multiple clients shared interleaved key set and a single key is updated concurrently, the last updated value will be visible to all clients.

4.2 Virtual Nodes on Hash Ring

Adding Virtual nodes is an efficient way of achieving better load balancing in consistent hashing. Real node are located in the hash ring in the locations indicated by their virtual IDs. In our implementation, virtual ID can be configured manually or randomly. The master server maintains a map between real nodes (a pair of IP and port number) and virtual nodes (virtual IDs). Upon a request, the server first calculates target virtual ID based on consistent hashing, and then translates the virtual ID to real node IP&port through the hashmap, and finally routes the request to the target cache server. A larger number of virtual IDs should divide the hash space more even, so that load between cache servers are more balanced.

The experiment cache system consists of 8 cache nodes with number of virtual IDs per node ranging from 1 to 8. All of the virtual IDs in this experiment is randomly generated in the space of unsigned 64-bit integer. In each experiment, we made 1,000 *PUT* requests with distinct keys to the master server and collects the number of keys in each cache node. The distribution of number of keys in shown in *Fig 3*; the standard deviation of each experiment is also shown in *Fig 4*. Both of the figures show that the load among different cache nodes become more balanced as the number of virtual IDs per node increases.

This result aligned well with our assumption: virtual node effectively alleviate the hot spot in the cache system, acting as a great way of load balancing for consistent hashing.

4.3 Effect of Write-ahead Log

We implemented the logging mechanism by redirecting the log output from Golang's `log` to a file attached to cache server. We use Golang's `os.File.Sync` to bypass OS buffer and flush the logs directly to disk, so that all the logs are timely persisted on disk.

Table 1 shows the average throughput and latency when standalone cache server handling 500 requests from

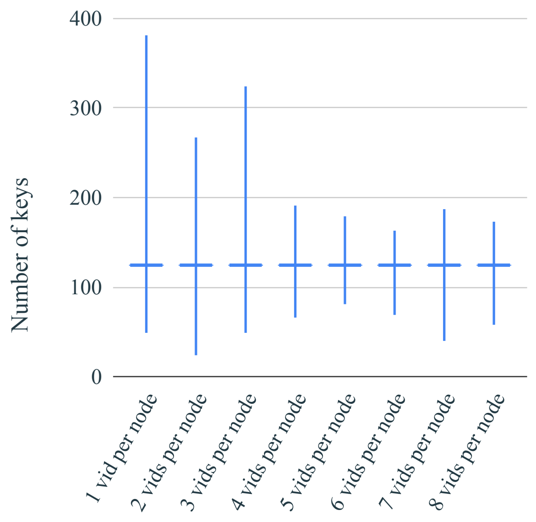


Figure 3: The min, mean, max value of the number of keys per node as a function of the number of virtual IDs per node. The cache system consists of 8 real nodes and stores 1,000 keys.

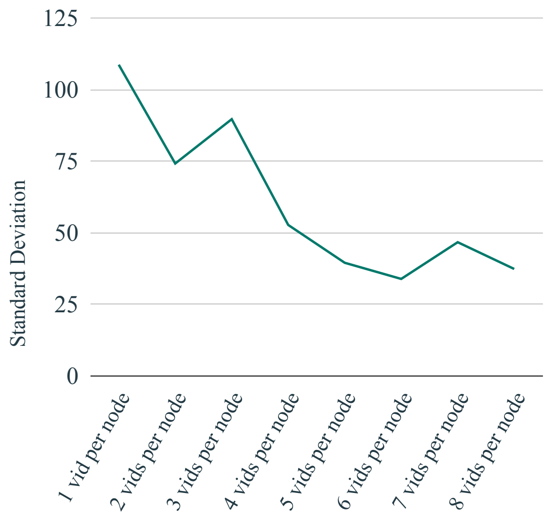


Figure 4: The standard deviation of number of keys per node as a function of number of virtual IDs per node.

	baseline	sync WAL	async WAL
thruput (rps)	298.7	56.2	294.3
latency (ms)	32.6	175.9	33.1
thruput change	100%	-81.2%	-1.5%
latency change	100%	+439.6%	+1.5%

Table 1: Throughput and latency of baseline (no logging implemented), Synchronous WAL, and Asynchronous WAL. The benchmark is done by creating 10 client goroutines with each client writing 50 records to a standalone cache server goroutines.

10 concurrent clients. The benchmark result is consistent with our understanding about WAL. Sync WAL suffers from a huge performance reduction, throughput and latency metrics are largely affected. Regrading async WAL, with a little overhead introduced from queuing logs into the pipe, it has almost the same performance as the baseline implementation. Despite the performance gain of async WAL, it actually trades the risk of potential data lost for better performance.

4.4 Optimizations on Cache Server

A well designed in-memory thread-safe hashmap might improve the efficiency of the cache server. To avoid concurrency issue, our baseline cache server blocked incoming *PUT* and *GET* requests and handle only one request at the same time. Then, we implemented hashmap with readers-writer lock and fine-grained segment locks. *Table* 2 shows the throughput comparison of three implementations evaluated on a single machine with two different workloads. Our test program created 50 coroutines to concurrently access the local hashmaps and computed the throughput of requests. The throughput of The hashmap with fine-grained segment locks was twice that of the baseline implementation, since it had better utilization of CPU parallelism and avoided unnecessary blocking. Fine-grained segment locks performed worse than reader-writers lock in read-only lock, probably because of the overhead of computing the hash value of each incoming key to put it on the right segment.

After evaluating three thread-safe hashmap on a single machine, we deployed cache servers with different hashmap implementation on the same *AWS EC2 instance*, and generated read-write workload with concurrent clients to measure the influence of hashmap optimization on a running cache server. *Fig 5* shows the improvement of remote cache server was much less significant. This result met our expectation, since the decisive factors might be the network bandwidth and efficiency of server implementation, which made the efficiency of hashmap implementation unimportant.

	Read Workload	RW Workload
Baseline	1x	1x
RW Lock	3.25x	1.05x
Fine-grained Lock	2.70x	1.97x

Table 2: Throughput comparison of baseline, readers-writer lock, and fine-grained segment locks. The benchmark is done by creating 50 coroutine on a single machine to access the hashmap structures concurrently.

4.5 Discussion

After implementing and evaluating our key-value store, we learned a few lessons. 1) Virtual node is a great idea for load balancing in consistent hashing. Increasing the number of virtual IDs assigned to each cache node could make the load distribution more balanced, even when all virtual IDs are just randomly generated. It also has great scalability: the lookup from virtual ID to physical address is achieved by a hashmap, so its efficiency would not be largely influenced by the number of nodes. 2) Logging is a classic and effective way of achieving fault-tolerant, but it is still restricted by the trade-off between performance and fault tolerance. We have evaluated two different logging strategies, which are almost the two ends of the spectrum of WAL logging: the most preferment one and the most fault-tolerant one. In the industry, it's the duty of engineers to choose their strategy from the spectrum and balance the trade-off. 3) A good way of understanding a thing is to build such a thing, but there would be some pitfalls in the process of building, and we also learned a lot from solving those pitfalls. For example, when we were evaluating different logging strategies, we found out there was no performance difference between those strategies. After we spent a few nights looking into the source code of Golang, we finally located the root cause: for each opened disk file, there will be an in-memory buffer to handle changes to this file, and changes will only be persisted to the disk when the buffer is full. That's the reason why we didn't observe the expected performance difference, as all of the logging operations actually happened in memory, no matter how we decide to write to the file on disk. Our final solution is to include the operation of flushing the in-memory buffer whenever writing to disk, and it just worked out as we expected.

5. FUTURE WORK

In addition to the work we have done, there are several interesting features deserve further exploration in the future. First, our system can apply fixed-size hash slots instead of consistent hashing to decouple the keys distribution from the assignment of hash identifiers. This approach is claimed to be better by the paper of Dy-

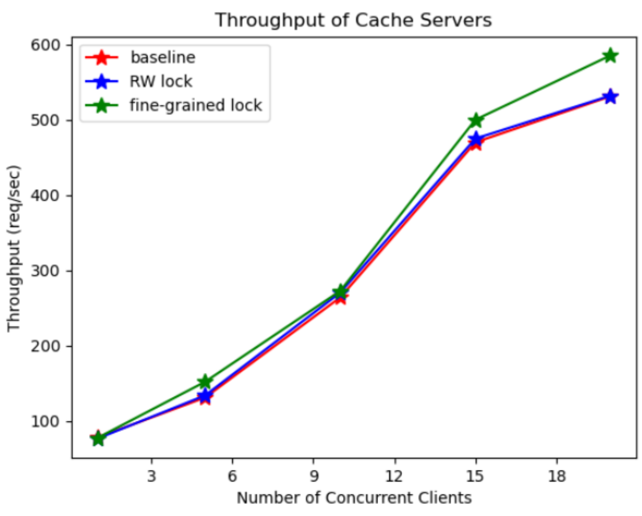


Figure 5: Benchmark of the cache servers with different thread-safe hashmap implementation. The cache servers were deployed on *AWS EC2 t3.xlarge instance*[6].

namo[2] since it can have better control over the distribution of the data and facilitate the rebuilding of Merkle tree. Second, we believe lazy migration will be beneficial for the current system, since it can mitigate the peak of network traffic within the data center caused by data migrations. Third, more data structures can be implemented to support more use cases. For example, skiplist-based ordered map and balanced binary tree can be helpful for the scenarios that require retrieval of ranked data items. Finally, we found there are few high-quality key-value workload traces available in the community for us to make a more solid evaluation of our systems. Researches[2, 4] use the real workloads in their products, which have never been made public. Thus, we believe it will also be a meaningful work for the industry to build key-value traces that can be equivalent to the real workloads in different scenarios.

6. CONCLUSION

We implemented a durable distributed key-value store with nearly 2,000 lines of Golang code. Our baseline implementation focused on master-slave architecture, consistent hashing, membership management, and data movement. We further implemented and evaluated various techniques that are widely used in the area of distributed key-value stores, including virtual node for load balancing, read-write lock & fine-grained lock for concurrent control, synchronous & asynchronous WAL for logging. The durability brought by WAL lets the systems become a reliable additional storage layer to store some frequently-requested data. Our experiments prove the effectiveness of adding virtual nodes into the hash ring and the efficiency of asynchronous WAL in the trade of losing absolute durability. Even though the fine-grained lock can dramatically improve the throughput of a single node hashmap, it doesn't bring to much performance gain because it's not the bottleneck.

7. REFERENCES

[1] D. Dave. Google trends dataset. https://www.kaggle.com/datasets/dhruvildave/google-trends-dataset. Accessed: 2022-04-18.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205220, oct 2007.

[3] D. Eastlake, T. Hansen, G. Fowler, K.-P. Vo, and L. Noll. The fnv non-cryptographic hash algorithm. https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-17.html. Accessed: 2022-04-18.

[4] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, page 385398, USA, 2013. USENIX Association.

[5] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 2941, New York, NY, USA, 2011. Association for Computing Machinery.

[6] A. W. Service. Amazon ec2 t3 instances. https://aws.amazon.com/ec2/instance-types/t3/. Accessed: 2022-04-19.

APPENDIX

Github Repo https://github.com/puyihua/meme-cache