# Implementation and Evaluation of Log-structured Key-Value Store

Zilong Zhou
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA
zilongz@andrew.cmu.edu

Jiarui Li
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA
jiaruil3@andrew.cmu.edu

## ABSTRACT

Key-value store with log structure is a data storage system that provides high performance, and fault tolerance for large-scale data processing applications. The system's key features include a log-structured data storage design, efficient data compression and encoding techniques, and a key-value store with replication and recovery mechanisms. In this paper, we present the design and implementation of a log structured key-value store (LogKV), including its key features and performance characteristics. We describe the system architecture and the data storage format, which is optimized for fast write and read operations, as well as efficient data compression and encoding. We also discuss the system's fault tolerance mechanisms, which enable it to handle node failures and ensure the high availability of data. We also introduce a useful debugging and evaluation tool for LogKV. To evaluate the system's performance, we conducted experiments using various workloads and benchmarked the system against the standard key-value store. Our results show that, under simulated large-scale workloads, LogKV outperforms the standard system in terms of write and read throughput, and fault tolerance. We also analyze the contention between persistence, performance and memory utilization in the general key-value store. Overall, our work demonstrates the effectiveness of LogKV as a high-performance data storage system for designed workloads. The system's efficient design and fault-tolerance mechanisms make it a viable solution for storing and processing large volumes of data in a performance-critical and fault-tolerant environment.

## Categories and Subject Descriptors

E.2 [**Data Storage Representations**]: Log Store; D.2.8 [**Software Engineering**]: Metrics—*Complexity measures, Performance measures*

## General Terms

Software and its engineering: Key-value Store

## Keywords

Log Key-Value Store, Failure Recovery, Compaction

## 1. INTRODUCTION

Log-structured key-value stores (LogKV) have gained significant popularity as a data management solution for large-scale workloads. The key-value store paradigm, where data is stored in the form of key-value pairs, has become a fundamental abstraction in many real-life scenarios. LogKV stores data in the form of log chunks, persisting operations in an append-only manner, which provides fault-tolerance capabilities crucial for large-scale services. However, the overhead and complexity introduced by the log structure raise questions about LogKV's performance, scalability, and trade-offs compared to traditional key-value stores.

We are interested in evaluating the LogKV's capability, using a naive implementation of the key-value store (NaiveKV) as a baseline. The paper mainly focuses on the following questions: (1) What's the LogKV's performance, in terms of throughput, fault-tolerance, scalability, and memory utilization? (2) What's the trade-off when the persistence feature supported by the log structure is added on top of the key-value store? (3) What's the reason for the performance difference and how does that imply further improvements? In our expectation, NaiveKV should have better performance under light workloads, since it is free from the overhead of maintaining the extra log structure. However, under large-scale workloads, where failure is inevitable, LogKV will significantly outperform NaiveKV, by providing more consistent and available service while introducing acceptable overhead. Additionally, we also examine the tradeoff between persistence, performance, and memory utilization in typical key-value stores.

We implemented LogKV and NaiveKV using c++, details of which can be found in Section 2. The code is open-source on GitHub.[1] We designed many experiments to verify the implementation and evaluate the performance of LogKV. The main technical contributions of this paper are:

- We show the LogKV's performance under simu-

[1]https://github.com/zhouzilong2020/log-kv-store

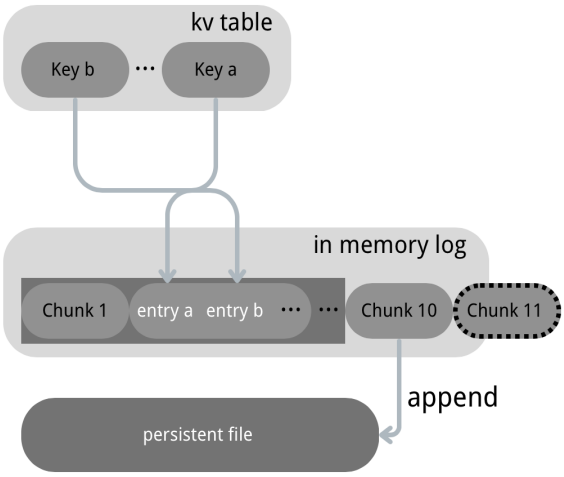


**Figure 1: The data model of LogKV. The in-memory kv table store the key and a pointer that points to the corresponding entry. Entries are organized in a chunk fashion, and a chunk a the smallest memory management unit when allocating and persisting.**

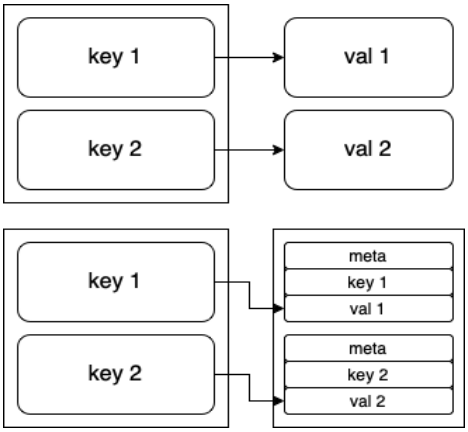


**Figure 2: The difference between the log-structured data store and traditional hash table. The top shows the memory layout of a hash table, the bottom shows the memory layout of log-structure data store.**

lated large workloads in terms of throughput, fault-tolerance, scalability, and memory utilization. We compare the results against the NaiveKV and show that LogKV outperforms NaiveKV under large workloads in almost all aspects.

- We identify two trade-offs, the performance-persistence trade-off and the memory-persistence trade-off, between LogKV and NaiveKV.
- We design and implement a delicate LogKV and an interactive tool for debugging and evaluation.

## 2. DESIGN

### 2.1 LogKV

The LogKV system is designed to store key-value pairs in memory and periodically save them to disk. It mainly provides three commands: **put** for storing a new key-value pair, **get** for retrieving the value of a given key, and **delete** for removing a key-value pair. To support fast read operation, the value stores a pointer to the in-memory key-value entry record in the log. The key and value are stored in general byte representation. It is also allowed that a key carries a null value.

LogKV uses a log-structured approach where updates are written to an in-memory, append-only log. The fast write operation is achieved since each writes sequentially appends a new entry to the end of the log. This approach allows multiple updates for the same key to be stored in separate log entries, reducing memory utilization. When memory utilization falls below a certain threshold, a compaction process will be triggered. The compaction process will remove the stale log entries by copying the active entries to new log chunks and updating the key-value hash table to point to the new entries.

The log is managed in chunks, which are the basic units of memory allocation. When a chunk does not have enough space for a new entry, a new chunk is allocated and the old chunk is appended to the disk. When failure is encountered, LogKV attempts to recover any persistent data from the disk during the boot up. LogKV recovers the in-memory key-value table by replaying the entries in the log sequentially.

### 2.2 NaiveKV

NaiveKV is a simple key-value store that serves as a baseline for our study. It stores data as a hash table, with each key corresponding to a value. NaiveKV uses an in-memory data structure, and thus, does not provide fault-tolerance. Since the hash table is rebuilt from scratch each time the system starts, NaiveKV has limited scalability, especially under heavy workloads. However, it is simple to implement and provides a baseline for comparison with more complex key-value stores. Besides, for better comparison, we enhance the NaiveKV with a simple persist approach, which will write the entire table as a snapshot into the disk, when the predefined persist threshold is reached.

## 3. IMPLEMENTATION

### 3.1 LogKV

LogKV is implemented as a log-structured key-value store, where data is stored in log chunks in memory and

on disk. We designed a custom log format that includes metadata and payload for each chunk. Each chunk has 2MB in total, including its metadata and entries. The metadata stores its create and last update time stamp, total capacity in bytes, used bytes, and entry count. Each entry is a four-variable structure composed of version number, key size, value size, and payload. The version number determines the sequential order or update operations, which is useful in compaction and recovery. The payload contains the serialized key-value pairs in a compact format and is dynamically allocated depending on the size of the key and value. We use 16 bits unsigned integer numbers for version number, key size, and value size, which supports $2^{16}$ character space for keys and values. Our compaction mechanism guarantees that the version number will never overflow.

When the 2MB is filled, this chunk will be persisted into the disk file. We decouple the disk file size from chunk size by making each file 2GB in size, which will hold 1KB chunks. Each file also has its metadata that specifies the create and last update time stamp, total bytes persisted, and chunk count. During the recovery, we used asynchronous I/O and write buffering techniques to optimize write performance and reduce latency. Recovery is performed with a 512MB buffer holding the entries fetched from the persistent file. Thus, we replay 256 chunks with each I/O operation.

### 3.2 NaiveKV

NaiveKV is implemented as a simple in-memory hash table. The put, get, and delete operations are wrapped hash manipulations. On every 2MB update operation on the hash table, a snapshot of the table will be taken. Each snapshot is a complete copy of the current key-value hash table. On recovery, the latest snapshot is used to reconstruct the table.

### 3.3 Test Tools

We have developed a set of tools to facilitate debugging and experimentation in our study. Specifically, we implemented a command line tool that allows users to launch the key-value store service with various options. Depending on the specified command options, the program can run the debugging unit tests or provide the key-value store service by listening to the stdin descriptor. The interactive operations are defined in a generic form as <opt> <key> <val>, where <val> is an optional parameter. Additionally, we have developed a Python script for synthesizing simulation data and supervising experiments, which are further elaborated in Section 4.

To accurately measure the memory utilization of our key-value store, we employed two techniques: the psutil python library and function interposition. The psutil library provides an API that allowed us to retrieve the memory footprint of the subprocesses after termination, providing a reliable estimation of the memory cost incurred by the key-value store. Additionally, we utilize a static link malloc function interposition technique during our evaluation phase to capture detailed data on memory allocation patterns. This technique involves intercepting and redirecting calls to the malloc function, a standard library function in C/C++ used for dynamic memory allocation, using our custom implementation. By employing this interposition technique, we are able to gather information such as the size of allocated memory blocks and the overall memory usage behavior of our log-structured key-value store with fine-grained granularity at runtime. This enables us to obtain accurate insights into the memory utilization of our system, facilitating our performance analysis and optimization efforts.

### 3.4 Development

Our development process begins with the use of GitHub for version control. We create feature branches for every new feature and any changes are made on this branch. Once a feature is complete, we commit it to the feature branch and merge it into the dev branch. This allows us to keep track of every feature and ensure that they are thoroughly tested before they are merged into the main branch. We also have a GitHub workflow that runs built-in tests to check the correctness of the merging code. This ensures that any code that is merged into the main branch is free of bugs and works as intended. When the dev branch is mature enough, we merge it into the main branch. By following this process, we ensure that our codebase is always up-to-date, and that any new features or changes are thoroughly tested.

## 4. EVALUATION

### 4.1 Method

We compare the performance of LogKV and NaiveKV under varying workload conditions. Specifically, we investigate the impact of four workload parameters on the throughput, duration, and memory utilization of each data store. These parameters are **hit rate**, **read fraction**, **hot fraction**, and **hot rate**.

Hit rate refers to the probability that a key is already in the data store and can be retrieved. Read fraction is the proportion of read operations in the workload, while hot fraction is the proportion of keys that are considered "hot" and accessed frequently. Hot rate is the rate at which the hot keys are accessed. By varying these parameters, we aim to simulate different real-world scenarios and evaluate how each data store performs under these conditions. Our hypothesis is that LogKV will outperform NaiveKV in any scenario if there is considerably large data (more than millions of key). Each

**Table 1: Values for Cartesian Product**

| Parameter | Possible Value |
|---|---|
| hit rate | 0.2, 0.6, 0.8, 0.99 |
| read fraction | 0.2, 0.4, 0.6, 0.8, 0.99 |
| hot fraction | 0.01, 0.05, 0.1, 0.2 |
| hot rate | 0.7, 0.8, 0.9 |

experiments are run five times and the results are averaged. For instance, the **read-heavy with hot key** configuration implies that 20% of the keys are responsible for 60% of the access, with 80% of the commands being read commands.

## 4.2 Data Synthesis

We conducted experiments on the performance of LogKV and NaiveKV by synthesizing test commands to simulate real-world scenarios. To explore various parameter combinations, we performed a cartesian product on predefined possible values (as shown in Table 1). Due to the large number of cartesian products, we could not execute all of them under our considerably large dataset (containing millions of keys). Therefore, we generated 100,000 commands for each set of parameters. Although this approach may not fully capture the real-world situation, we still believe it provides valuable insights into the behavior of LogKV and NaiveKV. Additionally, we considered sets of representative parameters that are reasonably close to real-world situations (as shown in Table 2) on a much larger scale of data (millions of keys).

## 4.3 Experiment

Experiments are conducted with an i7-9700 CPU, its detail specification is listed in Table 4. For the small scale experiment, we make the client a process that runs on the same machine with the data store. The command is send to the store server by pipe, and the output is retrieved from the server and dumped (redirect to /dev/null). As it is shown in Figure 3, under the 100,000 command workload. The naive method performs better in terms of the throughput. This is expected since in this set of experiments, the naive method is not performing any persistence, largely reducing the disk overhead. Such service is not reliable in the large scale, when failure happens from time to time.

The large-scale test is executed in the the same way as the small one, except there are 1,000,000 commands to be executed. Given the significant time required to conduct such an experiment, the experiment could only be conducted on empirically important scenarios identified by us.

Meanwhile, we suspect that the experiment results shown in Figure 4 and Figure 5 are buggy in terms of the experiment metrics measurement, since it does not match our expectation. According to our intuition, LogKV should demonstrate lower throughput when the write command fraction is low because in such scenario, persist happens less frequently. LogKV should outperform NaiveKV when the number of write operations goes up, especially under the 400,000 command and 0.2 read fraction experiment setting. However, Figure 4 shows an overlapping curve. Also, the memory utilization does not follow our expectation. Even though it is reasonable that when read fraction is 0.99, in which write rarely happens, meaning that there is little memory used in storing the key-value pair and most of the memory is used for metadata overhead, the memory utilization is low. We do not agree with the fact that when read fraction is 0.2, meaning that write and delete operations happens frequently, the memory utilization can reach the peak. This is because in LogKV, when the key in the table is updated, a new entry is inserted into the log, causing previous entry for this key becoming obsolete. Before compaction happens, this obsolete entry will be overhead, leading to a lower memory utilization metric.

## 4.4 Recovery

Table 3 shows our recovery experiment results on small workloads. Firstly, there are several clarification points to be made. We reported zero recovery time because the time measurement granularity our the current test tools are too rough (in the unit of seconds), which results in the lose of precision in the table. The inf throughput is the same problem. However, despite the mistake, the recovery statistics match our expectations well. LogKV shows a good recovery accuracy while sacrificing an acceptable amount of throughput performance. However, NaiveKV struggle with the tradeoff between recovery accuracy and throughput. When a small persist threshold is used, NaiveKV takes snapshot frequently, providing a low recovery error after the failure on the cost of a significant drop in terms of the throughput. When a larger persist threshold is used, NaiveKV collapse toward the NaiveKV without any persistence support, which has high throughput but low recovery accuracy.

# 5. DISCUSSION

## 5.1 Persistence or Performance

In key-value stores, there exists a tension between persistence and performance. Persistence refers to the ability of the store to ensure that every update is first applied to the local disk before it is acknowledged as successful. Performance, on the other hand, refers to the ability of the store to sustain a high throughput of requests. This tension stems from the additional I/O operations required for data persistence. These operations can significantly slow down the response time of

**Table 2: Empirical Values for a Theoretical Scenarios**

| Scenarios | hit rate | read fraction | hot fraction | hot rate |
|---|---|---|---|---|
| read heavy with hot key | 0.95 | 0.8 | 0.2 | 0.6 |
| read heavy without hot key | 0.95 | 0.8 | 0 | 0 |
| write heavy with hot key | 0.95 | 0.2 | 0.2 | 0.6 |
| write heavy without hot key | 0.95 | 0.2 | 0 | 0 |

**Table 3: Recovery performances of LogKV and NaiveKV**

| KV Method | LogKV | NaiveKV | | |
|---|---|---|---|---|
| Operation Number | 100000 | 100000 | | |
| Persist Threshold | 2MB | $2^6$KB | 2MB | $2^6$MB |
| Recovery Time (s) | 0 | 0 | 0 | 0 |
| Recovery Error | 3.55% | 0.17% | 4.53% | 99.18% |
| Throughput (Kop/s) | 50 | 1 | 17 | inf |

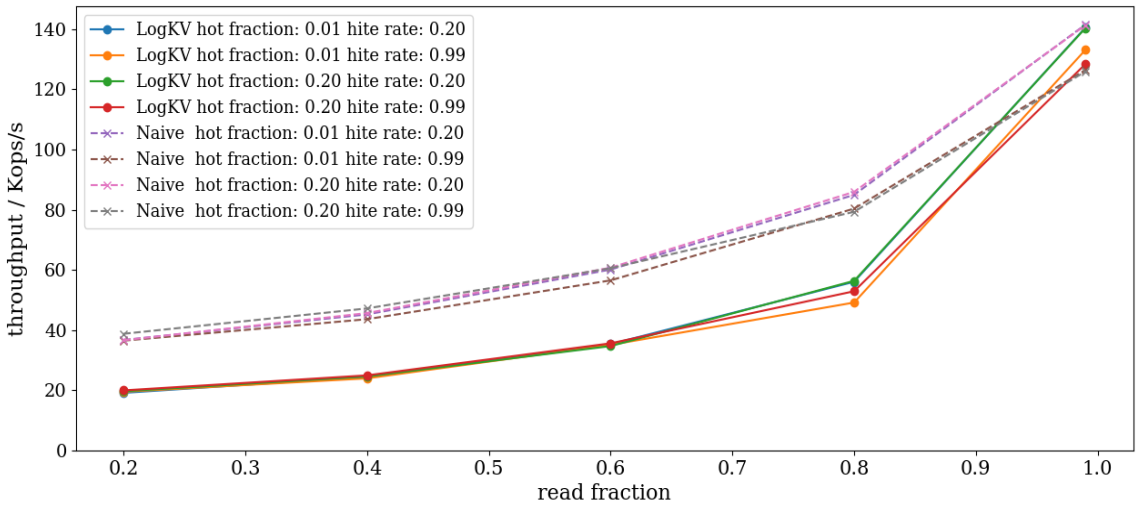


**Figure 3: Throughput of executing 100,000 commands with 0.9 hot rate in the unit of thousand operations per second (Kops/s). The dotted lines represents NaiveKV (without persist), and the solid lines represents the LogKV.**

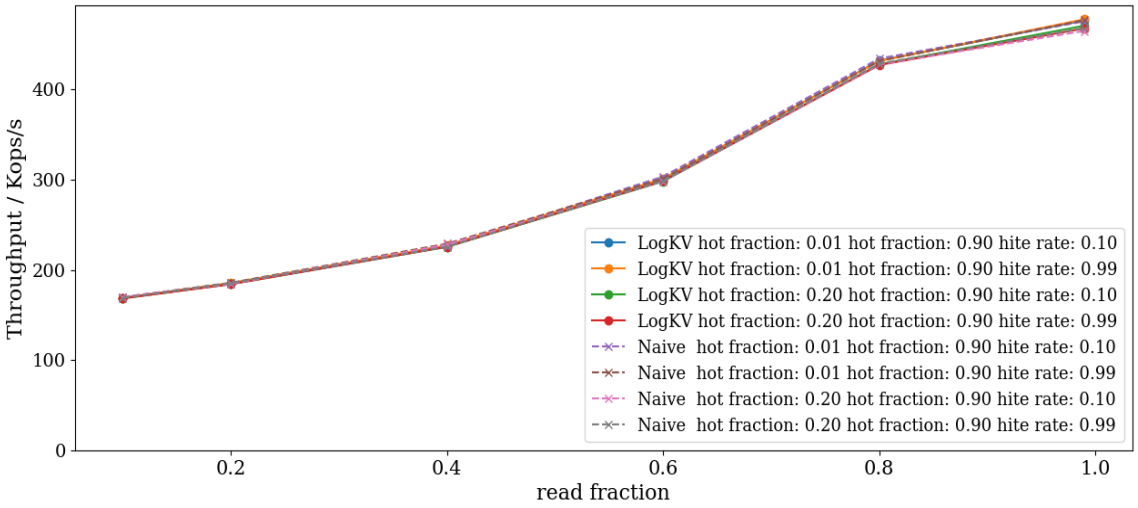


**Figure 4: Throughput of executing 400,000 commands with 0.9 hot rate in the unit of thousand operations per second (Kops/s). The dotted lines represents NaiveKV (with persist), and the solid lines represents the LogKV.**

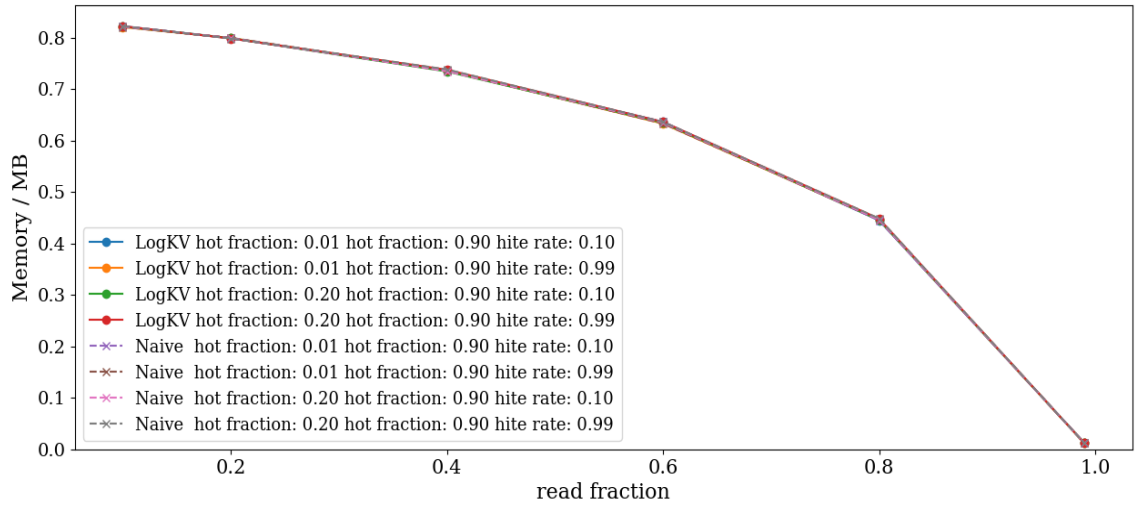


**Figure 5: Memory utilization of executing 400,000 commands with 0.9 hot rate. The dotted lines represents NaiveKV (with persist), and the solid lines represents the LogKV.**

Table 4: CPU Specifications

| Architecture | x86_64 |
|---|---|
| Address sizes | 39 bits physical<br>48 bits virtual |
| CPU(s) | 8 |
| Model name | Intel(R) Core(TM)<br>i7-9700 CPU @ 3.00GHz |
| CPU family | 6 |
| Model | 158 |
| Thread(s) per core | 1 |
| CPU max MHz | 4700.0000 |
| CPU min MHz | 800.0000 |

the store, as each write operation can take a considerable amount of time. As a result, many stores [10, 8] prioritize performance over persistence by acknowledging writes before they are persisted to disk. However, this approach increases the risk of data loss in the event of a system failure.

One way to balance persistence and performance is to use write buffering, where writes are first stored in memory and then flushed to disk periodically or when a certain threshold is reached. This technique allows the store to sustain high throughput while also ensuring that data is eventually persisted to disk. However, it also has its limitations, such as increased memory usage and potentially slower recovery times in the event of a failure. In addressing the growing memory consumption and recovery time, snapshots have become a common solution, as they offer a condensed version of logs. Achieving a balance between persistence and snapshots that ensures minimal data loss and sustained throughput in data stores. For instance, Redis [10] promotes the adoption of AOF (append-only file) to periodically transfer in-memory logs to disk on a second basis.

## 5.2 Persistence or Memory Utilization

Log means extra data structure beside key and value, and there might be duplicated entries when multiple updates are applied to the same key. As figure 2 suggests, there are extra meta info and duplicated key stored in log-structured data store. However, balance could be made between memory utilization and persistence by reverting to the normal store when the data is persisted to the disk, saving the extra memory introduced by log structure. Again, this does not come free, extra effort may needed to manage them.

## 5.3 Code Complexity

The implementation of LogKV introduces additional complexities compared to NaiveKV due to the need to manage an extra list of log entries, both in-memory and on disk, involving log meta data, dynamic memory allocation, and compaction. In our design, logs are organized into sections of chunks, with the log object maintaining an index for the chunks that specify the head and tail of the log, and a table for on-disk persist file paths to facilitate persistence and compaction operations. The compaction process in LogKV involves identifying stale entries, merging entries with copying, updating pointers in the key-value table, and redumping persist files, which adds computational and code complexity. Additionally, recovery from the byte representation in persist files, which requires reconstructing key-value pairs and object metadata sequentially and with corner cases handling, is challenging and nontrivial in coding. Since a chunk is the smallest logical unit for log management, we encapsulate it into a class that maintains metadata such as size, occupied bytes, and entry count, and handles dynamic memory allocation and freeing for storing key-value entries in sequence.

On the other hand, NaiveKV does not require the management of log metadata, dynamic memory allocation, or compaction. During compaction, NaiveKV simply appends the entire current key-value table to a file without any additional bookkeeping or maintenance, resulting in a simpler codebase compared to LogKV.

# 6. RELATED WORK

Log-structured key-value stores have been widely studied in the field of file systems and databases. In this section, we review related work on log-structured storage and key-value stores, and highlight the contributions of our research.

## 6.1 Log-Structured Storage

Log-structured storage has been proposed as an efficient way to organize data on disk, especially for write-heavy workloads. The concept of a log-structured file system was introduced by Rosenblum and Ousterhout in the early 1990s [11]. Since then, various log-structured storage systems have been proposed, including log-structured merge trees (LSM-Trees) [9], log-structured database [12], and log-structured file system [7]. These systems use different data structures and algorithms to organize and manage data in logs, and they have been shown to achieve high write throughput and efficient space utilization. However, they also have limitations in terms of read performance, memory usage, and fault tolerance.

## 6.2 Key-Value Stores

Key-value stores have gained popularity in recent years due to their simplicity, scalability, and performance characteristics. Many key-value stores have been proposed for various use cases, such as distributed databases [3], caching systems [8, 13], and storage systems for HPC supercomputers [5]. These systems provide a simple interface for storing and retrieving data based on keys,

and they are often designed to handle large-scale data and high write loads. Popular key-value stores include Apache Cassandra [6], Amazon DynamoDB [4], Bigtable [1], and Spanner [2].

# 7. LESSON LEARNED

## 7.1 Experiment Design

We spent too much time running wrong experiments. The results look wrong at the first glance, and it always turns out to be wrong experiment implementations (e.g., accidentally killing the store server before it receives and processes the commands). A sanity check on small data could save tons of time.

## 7.2 Testing is Critical

Implementation should always come with relevant test cases that can prove the basic correctness of the newly added code, which means each new code path should at least be executed by some test cases. During our development, we added new code paths that no test cases can actually execute them, which causes us lots of effort debugging it. The time spent on debugging could be 10x more than writing some simple test cases.

# 8. FUTURE WORK

The current version of LogKV exhibits stop-the-world behavior during kernel operations, such as compaction or persistence, which could be alleviated by implementing multi-threading. Besides boosting kernel operations, multi-threading may also be implemented for get and put operations, which could help to improve performance. Additionally, the naive approach of appending in-memory chunks to the same log file and only persisting to another file when the former exceeds 2GB could cause issues. Advanced log persistence techniques could be applied to expedite the persistence process. Furthermore, the current implementation lacks support for snapshots, which may be problematic for long-running services. Our experiments are conducted synthetically and the communication between the client and our service is based on pipes, which may not accurately reflect the actual performance of our store. A comparison between our store and commercial stores could be presented to demonstrate our performance. Most importantly, this implementation has several hyper-parameters that can be set, such as the chunk size, on-disk file size, and compaction policy. Further experiments are required to identify the optimal parameter set for achieving the best performance. Currently, there is not much optimization for hot keys, which is a common phenomenon in key-value store services. It would also be very interesting to add hot key optimizations to our existing framework.

# 9. CONCLUSION

The key-value store usually balances among persistence, performance and memory utilization. As our implementation of LogKV and experiment results show, log structure could help to have a good level of persistence while achieving high throughput and snapshot could help to reduce memory utilization.

# 10. REFERENCES


[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), aug 2013.

[3] 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):205–220, oct 2007.

[4] 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. 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.

[5] H. N. Greenberg, J. Bent, and G. Grider. Mdhim: A parallel key/value framework for hpc. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'15, page 10, USA, 2015. USENIX Association.

[6] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

[7] A. Merritt, A. Gavrilovska, Y. Chen, and D. Milojicic. Concurrent log-structured memory for many-core key-value stores. *Proc. VLDB Endow.*, 11(4):458–471, dec 2017.

[8] 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 385–398, USA, 2013. USENIX Association.

[9] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[10] Redis. Persistence - redis. https://redis.io/docs/management/persistence/. Accessed: April 24, 2023.

[11] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, feb 1992.

[12] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. Logbase: A scalable log-structured database system in the cloud, 2012.

[13] F. Wu, M.-H. Yang, B. Zhang, and D. H. Du. Ac-key: Adaptive caching for lsm-based key-value stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.