H^{2}KV: A Hotspot Awareness based Hybrid Fault-tolerant In-memory Key-Value Store

Lixiao Cui, Yingjie Geng*, Gang Wang†, Xiaoguang Liu†
College of Computer Science, TMCC, SysNet, DISSec, GTIISC, Nankai University, Tianjin, China
{cuilx, gengyj, wgzwp, liuxg}@nbjl.nankai.edu.cn

***Abstract*—In-memory key-value stores play an important role in data-intensive scenarios, becoming one of the infrastructures in modern data centers. With the expansion of data size, the probability of in-memory KV store cluster failure continues to increase. Therefore, it is important to construct fault-tolerant in-memory KV stores to ensure service availability. The widely used primary-backup replication (PBR) strategy can provide fault tolerance while ensuring high performance. However, it has high storage redundancy, which increases deployment costs. On the contrary, the erasure coding can achieve the same fault-tolerant level as PBR with less storage redundancy, but its computational and network overhead cause performance degradation.**

In this paper, we propose H^{2}KV, to balance the performance and storage overhead for fault-tolerant in-memory KV stores. Motivated by the hotspot phenomenon of in-memory KV stores, H^{2}KV applies a hybrid fault-tolerant strategy. For frequently accessed/updated hot key-value pairs, H^{2}KV stores replicas for them to achieve high-performance. While for less accessed cold data, H^{2}KV applies erasure coding mode to reduce storage redundancy. In order to accurately distinguish the hotness of key-value pairs, we propose a three-level hotspot filter mechanism. Moreover, to alleviate the encoding and decoding overhead of cold data, we pack the small-size key-value pairs to large blocks and apply block-granularity for erasure coding operations. We implement H^{2}KV on popular Redis and evaluate it from multiple dimensions. The results show that compared to the replication method, the memory footprint of H^{2}KV is 63% to 77%. Meanwhile, H^{2}KV can maintain comparable performance with PBR and significantly outperforms erasure coding.

***Index Terms*—in-memory key-value store, fault tolerance, hotspot awareness, erasure code**

I. Introduction

In-memory key-value (KV) stores, such as Redis [1] and Memcached [2], have been widely used to support efficient web service and database operations. Large-scale in-memory KV store clusters containing tens of terabytes of data have been deployed on Facebook [3], Twitter [4], and Alibaba [5].

As the cluster size increases, failures become more common [6]. For in-memory KV stores, a node crash will cause all data in memory to be lost, affecting the overall performance greatly. Therefore, it is necessary to provide fault tolerance mechanisms for in-memory KV stores. A naive approach is to utilize the local block device to persist the data and restore it to memory after a failure. This method has been implemented in some systems, like the AOF (Append Only File) of Redis. However, recovering data from block devices to memory can consume much time. According to Facebook, it needs 2 to 3 hours to rebuild 120GiB data from disk to memory [7]. The long recovery times can affect service quality. In this paper, we mainly focus on more efficient fault tolerance methods in distributed in-memory KV stores, i.e., replication and erasure coding.

Primary-backup replication (PBR) [8] is a common fault-tolerant method, where N backup nodes save the data replicas of a primary node. PBR mode can achieve high availability and high performance. When the primary node encounters failures, a backup node will take its place as the new primary node. This switch can ensure the continuity of request processing. However, PBR need many storage resources, i.e. $N+1$ times memory overhead to tolerate N fails. Since DRAM is expensive, PBR mode can greatly increase the deployment cost. Erasure coding is another widely used fault-tolerant method, which uses calculated data redundancy (i.e. parity) to achieve data availability. An erasure coding [9] based distributed system contains data nodes and parity nodes. The data nodes store raw data from users and the parity nodes store parity data calculated from raw data. Compared with PBR, on the one hand, erasure coding has worse performance, which spends more computation and network overhead for encoding and decoding. On the other hand, erasure coding is more space-efficient. It typically requires only several times less storage space than PBR to achieve the same level of fault tolerance.

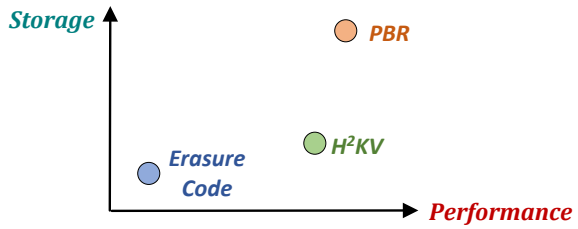


Fig. 1: Theoretical performance vs storage overhead tradeoff.

We seek to build a fault-tolerant in-memory KV store to obtain performance and space efficiency simultaneously, as illustrated in Figure 1. An idea is to combine PBR and erasure coding to build a hybrid fault-tolerant scheme. In this paper, we propose H^{2}KV, which applies a **hotspot-aware hybrid fault-tolerant method** for in-memory KV stores. Different from previous work [10], [11], H^{2}KV determines which fault

* Yingjie Geng is the joint first author.
† Corresponding author.

tolerance scheme should be used for different data according to its hotness. H^{2}KV is motivated by the skewness workloads of real-world scenarios [4], [5]. In a production environment, a small set of data will carry most of the requests. This phenomenon has been reported by Twitter [4], Alibaba [5] and Facebook [3]. **For hot data (i.e. frequently accessed/updated key-value pairs), H^{2}KV applies high-performance PBR.** Due to the small amount of hot data, the use of replicas has little impact on the overall storage overhead. Meanwhile, PBR is more efficient when frequently writing data because it does not require complex calculations. To accelerate read, PBR also allows users to read multiple replica nodes concurrently. **For cold data (i.e. less accessed/updated key-value pairs), H^{2}KV builds space-efficient erasure coding.** Cold data accounts for most of the total data volume. Using erasure coding for it can reduce the overall storage overhead. Besides, there are fewer cold data updates, which effectively reduces the encoding calculation and network transmission overhead.

To implement H^{2}KV efficiently, we still face many challenges. First, how to identify hot and cold data. To maintain low space consumption and high performance, H^{2}KV needs to find the rare, most popular key-value pairs and use a replication strategy for them. To solve this issue, we propose a three-level hotspot filter mechanism that is able to select hot data precisely. Second, how to improve the encoding and decoding efficiency of cold data further. The updates on cold data will not inevitably affect runtime and recovery performance, this is mainly brought by encoding/decoding overhead. Recent studies [3], [4] observe that the key-value pairs are usually small. Encoding them directly will result in poor encoding performance and multiple network communications. To solve these problems, we propose a block structure to organize small KV pairs into big blocks and encode/decode data at block granularity, reducing computational and network overhead. Finally, how to recover data after a crash leveraging hybrid fault tolerance. To solve it, we propose a two-phase recovery mechanism, which uses backup nodes of PBR to restore service quickly and asynchronously recovers the encoded cold data, minimizing the recovery overhead.

We implement H^{2}KV based on widely used Redis. The evaluation results show that H^{2}KV achieves performance close to PBR when uses 63% to 77% of its memory footprint. The main contributions of this paper are as follows.

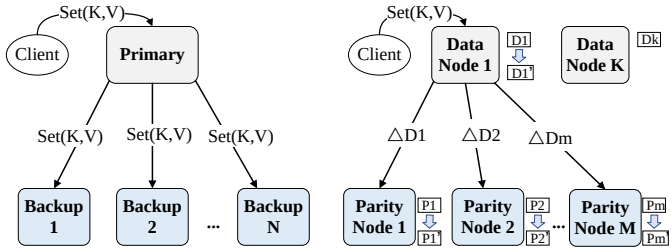
1) We propose to choose different fault tolerance modes according to the hotness of data to balance performance and memory overhead.
2) We design a three-level hotspot filter mechanism to distinguish hot and cold data.
3) We propose to apply block-granularity for erasure coding to improve the efficiency of encoding/decoding.

The organization of this paper is as follows. In Section II, we introduce the background of fault-tolerant mechanisms. Then in Section III and Section IV, we explain the key ideas and design details of H^{2}KV. In Section V, we show the evaluation results of H^{2}KV from multiple dimensions. Finally, we discuss related work in Section VI and conclude this paper in Section VII.

II. Background

A. Primary-Backup Replication

Primary-backup replication (PBR) [8] is a popular fault tolerant approach. It has been employed by several widely-used distributed storage systems including Ceph [12], GFS [13], and HDFS [14] to ensure data availability. The fundamental idea of PBR is to store multiple copies of user data on different server nodes, thus achieving data redundancy. As shown in Figure 2a, a fault-tolerant group contains a primary node and N backup nodes, which can to tolerate N node failures. When a node in the cluster fails, the service remains uninterrupted as long as there is at least one complete copy of the data within the cluster. The PBR mechanism can offer fast recovery with simple implementation and easy to scale. However, PBR brings high data redundancy, which spends $N+1$ times storage of original raw data to tolerate at most N failures.



(a) KV-store with PBR (b) KV-store with erasure coding

Fig. 2: Two different schemes of fault tolerance

B. Erasure Code

Erasure coding originates in the field of communication and is introduced into distributed storage systems to achieve data redundancy due to their low storage overhead. In recent years, an increasing number of distributed storage systems such as HDFS-RAID [15], Azure [16] and Ceph [12] begin to use erasure coding to reduce storage overhead while ensuring system reliability while ensuring system reliability.

Reed-Solomon code (RS-code) [17] is the most popular coding scheme. An RS-code scheme can be represented as $\mathrm{RS}(N, K)$, where N denotes the code length, K represents the length of the effective information, and it can tolerate at most $N - K$ node failures. As Figure 2b shows, in a distributed system with N nodes, the raw data is divided into K data nodes, which are then encoded to generate M parity nodes. When a node fails, the data can be fully recovered through decoding operations using the data from any remaining K nodes. Compared with PBR mechanism, erasure coding can offer significantly lower storage redundancy while providing the same fault tolerance capability. For example, for an $\mathrm{RS}(5, 3)$ scheme, the storage redundancy is only 166%.

However, erasure coding also has inherent limitations, as the encoding, updating, and decoding processes all introduce huge transmission and computation overheads.

The encoding process of RS-code can be represented by a matrix calculation. Because RS-Code is a linear coding, it can be uniquely determined by a generated matrix S. We use D and P represent the raw data vector and the parity data vector respectively. Therefore, $S \times D = P$ can represent the encoding process. Similarly, during the recovery process (i.e. decoding), the system can recalculate lost data or parity data by solving equations derived from the aforementioned formula. To improve the performance of data updates, we can apply differential update [18]. In this case, data nodes only need to transmit the XOR delta to parity node rather than transmitting the entire new data. The parity node with update parity data by adding the delta with a predefined coefficient, which is shown in equation 1.

$$\begin{bmatrix} p_0^{'} \\ \vdots \\ p_i^{'} \\ \vdots \\ p_{M-1}^{'} \end{bmatrix} = S \times \begin{bmatrix} d_0 \\ \vdots \\ d_i + \Delta d_i \\ \vdots \\ d_{M-1} \end{bmatrix} = \begin{bmatrix} p_0 \\ \vdots \\ p_i \\ \vdots \\ p_{M-1} \end{bmatrix} + \begin{bmatrix} a_{0,i} \\ \vdots \\ a_{i,i} \\ \vdots \\ a_{M-1,i} \end{bmatrix} \times \Delta d_i \quad (1)$$

III. Key Ideas

1) Using PBR for hot accessed while using erasure coding for cold data. The real-world access pattern of in-memory KV store usually follows zipfian distribution [3], [4], [19]. Only a small amount of data is accessed frequently and the rest of the data is barely visited. As reported by Alibaba [5], in their daily production environment, about 50% of total accesses only touch 1% of total data. In extreme cases, 1% of total data even handles 90% of users' accesses. Recent studies [3], [4] from Facebook and Twitter also confirm the distribution of their workloads fits the zipfian distribution and is even more skewed than the popular YCSB benchmark [19]. For example, the skewness parameter of the extreme data distribution of Alibaba in-memory systems is 1.22, which is much higher than the default skewness (i.e. 0.99) of YCSB.

We design H^{2}KV targeted for the skewed workloads in real scenarios. H^{2}KV introduces high-performance but space-intensive PBR for hot accessed and applies space-efficient but poor-performance erasure coding for cold data, thereby balancing the performance and memory usage.

2) Using block granularity for erasure coding. The size of key-value pairs of in-memory KV stores is usually small. According to a study [4] from Twitter, about 75% of KV pairs are less than 1KB in size. The median size of key and value is 38B and 230B. Eisenman [20] et al. also report the statistics of item size from Memcachier, a commercial in-memory KV cache service. They find the average size of key-value items is 257B. For erasure coding, small data granularity is not performance-friendly. We measure the encode/decode throughput (with SIMD acceleration) of a popular erasure coding library, Jersure [21], under different block sizes. As

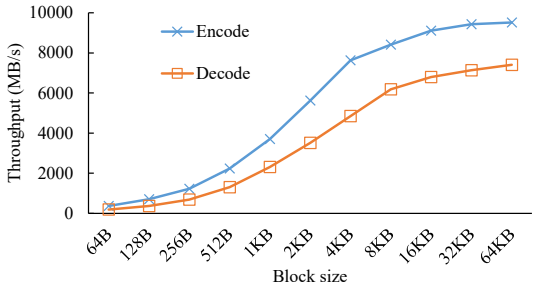


Fig. 3: Throughput of encode/decode varying block size

Figure 3 shows, favorable encode/decode performance can only be obtained when the block size is large. This observation illustrates that small-grained key-value pairs are not suitable for efficient encoding and decoding.

H^{2}KV aims to solve it by packing key-value pairs into large blocks with unified size and using block granularity for erasure coding. *This design can not only improve the computational efficiency of erasure code, but also reduce the number of network communications.* For example, when we pack 128 32-byte key-value pairs into a 4KB block, we only need one network communication. However, it takes 128 times when unpacked. Since network communications usually cause overhead like system calls and data transmission, reducing the number of communications can greatly alleviate overhead brought by erasure coding.

IV. Design

A. Overview

The fault-tolerant mechanism of H^{2}KV is unified and can be applied for distributed in-memory KV stores like Redis [1] and Memcached [2]. In this paper, we implement H^{2}KV based on Redis. There are two commonly used basic interfaces for H^{2}KV: `Get` and `Set`. They handle reads and updates/writes to KV pairs respectively. We only focus on the failures caused by power outages or device errors. Commissions and Byzantine failures are out of this paper.

The basic fault-tolerant unit of H^{2}KV is an H^2-group. A group consists of N data nodes, K parity nodes and N*K backup replication nodes, which can tolerate up to K failures. Figure 4 shows an example of H^{2}KV ($N = 3$ and $K = 2$) that can tolerate up to 2 failures. The data nodes are responsible for dealing with `Get` and `Set` requests from clients. In data nodes, H^{2}KV applies a three-level hotspot screening mechanism to distinguish hot and cold data, which we will discuss in Section IV-C. For hot data and warm data, H^{2}KV creates replications on backup nodes to achieve performance-efficient fault-tolerance. The backup nodes also can serve to `Get` to improve the overall performance. Because backup nodes store hot data, most of read requests can be processed successfully. For cold data, H^{2}KV calculates parity data for them and stores redundancy in parity nodes to achieve space-efficient fault-tolerance. Specifically, H^{2}KV first packs the cold key-value pairs into large data blocks, then encode the

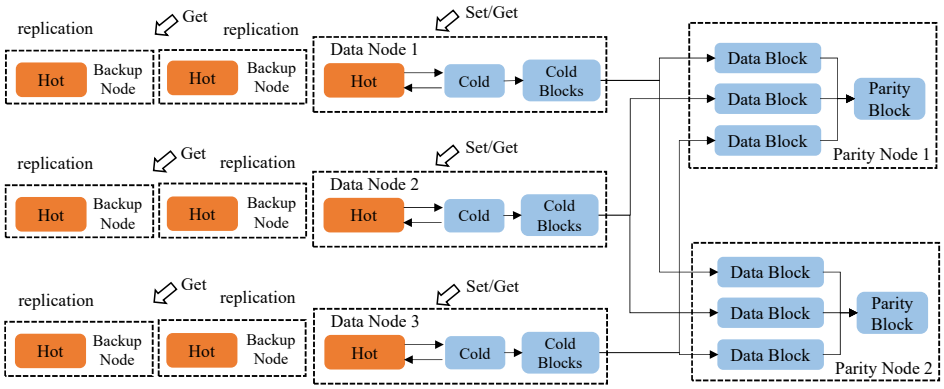


Fig. 4: The overview of H^{2}KV where N is 3 and K is 2.

parity block according to the aligned data block. When the encoded cold data is updated, data nodes only need to calculate the XOR delta between before and after the update and use the delta to update parity blocks. We will introduce the details in Section IV-B.

For recovery, H^{2}KV applies a two-phase strategy. In the first stage, H^{2}KV first switches the backup node to the primary data node, thus guaranteeing the quick restoration of service. In the second stage, H^{2}KV recovers cold data according to parity data to complete full data recovery. Section IV-D will introduce the implementation details.

B. Erasure Code with Block Granularity

1) ***Data organization****:* To alleviate the encode/decode overhead, H^{2}KV packs several small key-value pairs into aligned large blocks. The block structure is shown in Figure 5. The data organization of H^{2}KV is motivated by slab allocator and prior in-memory key-value stores [2], [22]. Like previous work [2], [22], we divide a block into multiple chunks of fixed length. The key-value pairs are stored in chunks. The blocks with the same chunk size are grouped into the same category. The number of categories is limited. As Figure 5 shows, when the block size is 4KB, we divided the blocks into 256 categories in total, the chunk size increases from 16B to 4KB, and the step size is 16B. To quickly find a block that can be stored when writing a key-value pair, we link the blocks of every category into two lists, an available list and a full list. The full list stores the blocks in which all chunks are occupied. On the contrary, blocks in the available list have free chunks where new key-value pairs can be placed. In the current implementation, the default size of a block is 4KB.

Figure 5 also illustrates the metadata layout of a block. To conveniently manage blocks, a `bitmap` is used to mark the usage of chunks in the block. The `category id` records which category the block belongs to. The `ratio` records the proportion of chunks already used to the total, providing a reference for garbage collection. The `prev` and `next` pointers are used to link blocks into corresponding lists. The `bid` and `addr` are used for efficient encoding/decoding. Among them, `bid` is the serial number of the block, which is used

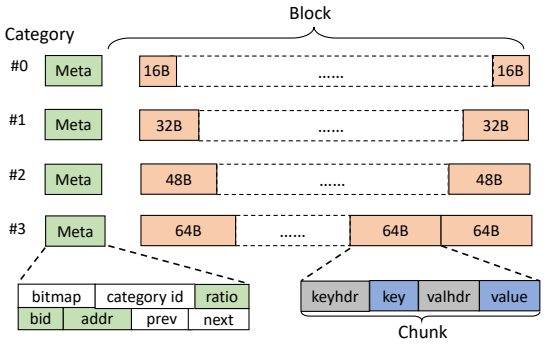


Fig. 5: The structure of block.

to index the block during the encoding and decoding process. The `addr` is used to find corresponding parity blocks. We will introduce the role of both in detail in Section IV-B2. H^{2}KV selects the appropriate category of blocks according to the size of a KV pair and stores it into a chunk. The `keyhdr` and `valhdr` are used to record the size of key and value respectively.

2) ***The calculation of parity data****:* In H^{2}KV, the basic unit of encoding/decoding is a data block. Taking RS(5,3) as an example, each coding stripe contains three data blocks and two parity blocks. Figure 6 shows the block organization structure of a parity node. Another parity node has the same structure. In data nodes, H^{2}KV maintains a hash table that maps `bid` to each block to retrieve the data blocks. When a block becomes a full block (i.e. there are no free chunks), the data node sends this block to parity nodes. In a parity node, H^{2}KV maintains a block list for each data node to save the metadata of the data blocks sent by different data nodes. Every time the parity node receives a data block, its metadata will be added to the end of the corresponding linked list.

When generating parity blocks (i.e. encoding), The data blocks at the same position in the linked lists will be included into the same stripe to calculate the corresponding parity blocks. For example, the $parityblock_0$ is calculated according to three data blocks at the head of the linked lists, that is the $dblock_0$ from node 0, $dblock_2$ from node 1 and $dblock_1$

from node 2. After completing the calculation of the parity data, the memory space occupied by the data blocks will be released, and only the metadata will be reserved. The `addr` of the block metadata will point to the parity block corresponding to the data block. For recovering data after a crash, the lost data blocks need to be retrieved based on parity data (i.e. decoding). To complete this processing, H^{2}KV needs to obtain the surviving data blocks and parity blocks in the stripe. For example, if data node 0 fails, H^{2}KV will traverse block list 1 and block list 2 to get the `bid` of the surviving data block, and request the data blocks from the corresponding data nodes according to the `bid`. Meanwhile, H^{2}KV will check the `addr` field of data block metadata to obtain parity blocks.

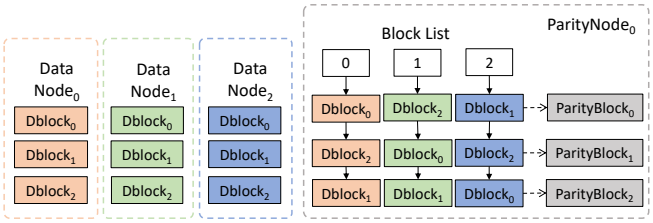


Fig. 6: The encode/decode structure with block granularity.

*3) **Data transfer**:* When performing fault-tolerance operations, in-memory KV stores usually leverage an asynchronous transfer strategy to obtain better performance. For example, the PBR implementation of Redis maintains a replication backlog on both the primary and backup nodes respectively. The backlog is a fixed-length queue that stores the data from primary to backup nodes. The primary node sends data to backup nodes asynchronously. Every time data is sent, the primary node updates the data offset in the backlog. Similarly, the backup nodes also update the offset after receiving the data. To ensure data consistency, the primary node will check whether the offsets of the primary and backup nodes are the same in each heartbeat detection, and resend the corresponding data from the backlog if they are inconsistent.

H^{2}KV also applies an asynchronous data transfer strategy. When sending data blocks to parity nodes, H^{2}KV maintains a linked list to store blocks that have been sent but have not been confirmed by the parity nodes. Similar to the backlog of Redis, the length of the linked list is also fixed, which prevents excessive data differences between data nodes and parity nodes. After the parity nodes receive data blocks, the encoding process will also be performed asynchronously without blocking the data nodes. In the process of calculating the parity, the data blocks are saved in the parity nodes, which is equivalent to using multiple replicas for fault tolerance. Only when the encoding calculation is completed, the data blocks will be released.

*4) **The update of parity data**:* There are two operations that result in updates of the parity data, updates of key-value pairs, and garbage collection of data blocks. For the updates of a key-value pair, if the size of the new key-value pair does not exceed the size of current stored chunk, it will be updated in place. The delta parity will also be calculated to update the parity block. If the new key-value pair is too large to update in-place, it will be placed in other available data blocks. When the block becomes full, it will be used to generate a parity block along with other full data blocks. This non-in-place update will cause 'garbage' chunks in the old block, which is recorded by the `bitmap` and `ratio` fields of block metadata. When there are too many garbage chunks in a block (the `ratio` is lower than 0.5 by default), H^{2}KV will perform garbage collection.

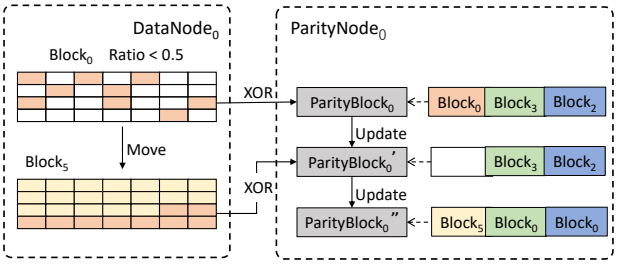


Fig. 7: The updates of parity data when conducting garbage collection.

The garbage collection can incur updates to parity blocks. Figure 7 shows an example. Block 0 in data node 0 has a low `ratio` and is selected to be recycled. H^{2}KV first moves the available key-value pairs in block 0 to other blocks (block 5 in Figure 7) and clears block 0. Then, H^{2}KV obtains an XOR delta between cleared block 0 and original block 0 to update the parity block 0. Meanwhile, the parity node updates the metadata of block 0 to mark it as recycled. Finally, when block 5 becomes full, H^{2}KV uses it to replace the original position of block 0 to update the parity block 0. The XOR delta between block 5 and a cleared block is calculated and sent to parity nodes. The block metadata in the parity node is also updated to record the information of block 5. In extreme cases, all data blocks in a stripe will be garbage collected and become cleared blocks. At this time, the parity blocks in the stripe will also be released.

C. Hotspot Aware Mechanism

H^{2}KV adopts different fault tolerance modes according to the hotness of data. The overview of hotspot aware mechanism is shown in Figure 8. H^{2}KV records hotspot information at the granularity of key-value pairs. Previous work [5], [23], [24] on hotness identification usually divide data into two states, cold and hot. However, this approach can lead to frequent state switching of data between hot and cold. This is unacceptable for H^{2}KV, because state switching will lead to fault-tolerant mode switching, resulting in huge overhead. To filter frequently accessed data precisely and avoid frequent fault-tolerant mode switching, H^{2}KV proposes a three-level filter mechanism. The key-value pairs are divided into three categories, hot, warm and cold. For hot and warm data, H^{2}KV applies PBR for fault tolerance. For cold key-value pairs, they are packed into blocks as described in Section IV-B. H^{2}KV applies erasure code for them.

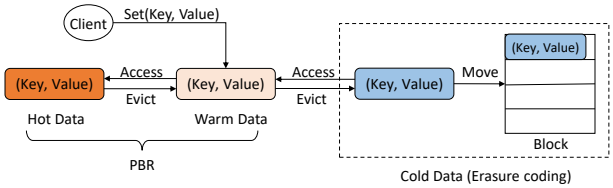


Fig. 8: The overview of hotspot aware mechanism.

*1) **Three-level filter mechanism**:* The selection of hot data and cold data can heavily affect the overall performance of H^{2}KV. The easiest way is to use traditional cache eviction policies like LRU and LFU, but these strategies have two main disadvantages that limit their use on H^{2}KV. First, these strategies are not precise when filtering cold data. Current in-memory KV stores usually do not maintain the LRU/LFU linked list, but uses the sampling mechanism [1], which saves precious memory resource occupied by the linked list pointer. However, limited by the number of samples, the sampled key-value pairs may not contain cold data, so it is easy to evict hot data. Second, hot and cold switching is frequent. These cache strategies simply put a key-value pair to cache once it is read, which can make some data frequently switch between hot and cold data. This phenomenon is exacerbated by imprecise filtering due to sampling. Applying this approach to H^{2}KV will incur significant performance degradation. Other hotness identification work with only two state (i.e. cold and hot) also has similar switching overhead. When a cold kv pair is judged as hot data, H^{2}KV needs to switch the fault tolerance mechanism for it. This requires multiple network communications with backup nodes. Similarly, when hot data changes to cold data, H2KV also needs to pay network and computing overhead to complete erasure code fault tolerance.

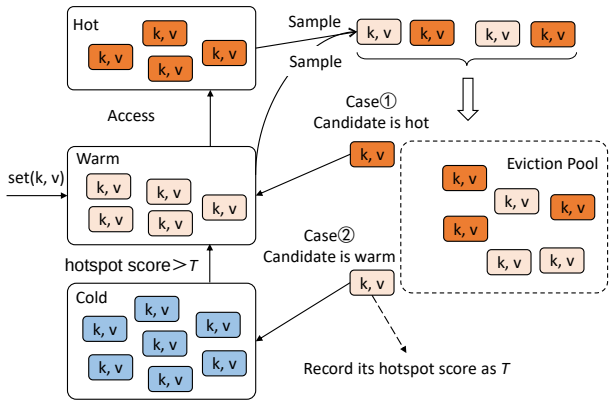


Fig. 9: Three-level hotspot filter mechanism.

The three-level filter mechanism can solve the above issues effectively. As Figure 9 shows, a newly inserted key-value pair is set to warm first. When the key-value pairs in warm state are accessed, they are promoted to hot state. This state change will not cause additional overhead, because both hot data and warm data use PBR to ensure fault tolerance. H^{2}KV can set the maximum capacity hot_{max} of hot data and warm data. When the total capacity of key-value pairs marked hot and warm exceeds hot_{max}, data eviction is triggered. Same with prior work [1], H^{2}KV also uses a sampling method to evict cold data. We maintain an eviction pool to store sampled key-value pairs of the most recent rounds. Each eviction will find the coldest data in the eviction pool, thus improving the accuracy of filtering cold data. H^{2}KV can use any traditional cache evict algorithms like LRU and LFU to select candidates. The biggest difference from the previous methods is that H^{2}KV does not set all candidates to cold. If a candidate key-value pair is mark as hot state, it will be set to warm. This gives the hot key-value pair a second chance, reducing the possibility of it being misidentified as cold. If a candidate key-value pair is marked as the warm state, it will be evicted to cold region. Meanwhile, H^{2}KV records a hotspot score T of the evicted key-value pair, which is the number of accesses of the recent period. For cold key-value pairs, H^{2}KV checks their hotspot score when accessing them. Those key-value pairs whose score exceeds T will be promoted to warm data. This avoids the frequent conversion of cold data to hot data.

*2) **Switching of fault-tolerant mode**:* Changes in the hot and cold state of the key-value pairs can bring the switching of fault tolerance mode. For new key-value pairs, H^{2}KV sets them to warm and leverages PBR to ensure fault tolerance. When a warm key-value pair becomes cold, H^{2}KV moves it into a data block and switches fault-tolerant mode to erasure coding. The encoding process is described in Section IV-B. During this process, the replications on the backup nodes are reserved. Only when the primary node receives the information that the parity nodes have completed the encoding, it will release the replications. Therefore, the data reliability is still maintained when switching modes. Similarly, when cold data is converted to hot data, the parity data will be updated when the replicas are established. The update of parity data also uses an XOR delta strategy as introduced in section IV-B4.

D. Recovery

*1) **Crash of data nodes**:* When the data nodes crash, H^{2}KV adopts a two-stage recovery mechanism to quickly restore services. In the first stage, H^{2}KV conducts a simple primary/backup switch to select a backup node as the new primary node. Since the backup node stores hot data, the switched primary node can respond to most user requests. Since in-memory KV systems are generally used as a cache, for requests to cold data that cannot be processed in the switched primary node, the underlying persistent system (e.g. databases) is responsible for handling them. This stage can ensure fast performance recovery of H^{2}KV after failures. In the second stage, H^{2}KV recovers cold data from parity nodes. This process is slow because the decoding of the erasure code requires a lot of calculations and multiple network transmissions. However, since the first stage has recovered hot data, the decoding of cold data can be performed in the background. Compared to previous work [10], [11], [25] using erasure codes for fault tolerance, the two-stage recovery

mechanism of H^{2}KV does not need to wait for slow decoding, achieving fast recovery with small storage redundancy (i.e. the hot data using PBR).

*2) **Crash of backup and parity nodes**:* The crash of backup and parity nodes only reduces the fault tolerance level of H^{2}KV but does not affect service availability. H^{2}KV will restore the backup nodes and parity nodes respectively according to the hot and cold information recorded on the data nodes. For a backup node crash, H^{2}KV directly sends hot data to the node to complete the recovery. For a parity node crash, H2KV will recover the lost parity blocks according to the information of the parity strips on the other surviving parity nodes. If all the parity nodes crash, H^{2}KV will re-encode the parity based on the cold key-value on the data nodes as described in Section IV-B.

E. Implementation

We implement H^{2}KV on Redis 4.0.14. For hot KV pairs, H^{2}KV uses `jemalloc` to manage them. For cold KV pairs, H^{2}KV leverage block structure as introduced in Section IV-B to store them. H^{2}KV also adopts a hash table as the index like other in-memory KV stores. We add a field to the hash entry of H^{2}KV to mark the status of the KV pairs (hot, warm or cold). In the current implementation, H^{2}KV applies reed–solomon (RS) coding for erasure coding. But other more efficient coding mechanisms, like RDP code [26], [27], can also be easily implemented in our work. For PBR implementation, H^{2}KV still uses the same mechanism as Redis. H^{2}KV uses LFU in the three-level filter mechanism to filter candidates by default.

V. Evaluation

A. Experimental Setup

An H^2-group is the basic fault-tolerant unit. Therefore, in this section, we measure the performance of an H^2-group. The evaluated H^{2}KV is configured to tolerate at most two failures, which contains 3 data nodes, 6 backup nodes and 3 parity nodes. We create 12 virtual machines to host an H^{2}KV fault-tolerant group. The hardware configuration of evaluation platform is equipped with two Intel(R) Xeon(R) Gold 5220 CPUs (2.2GHz, 18 cores, 32KB L1i cache, 32KB L1d cache, 1MB L2 cache and 25MB L3 cache) and 128GB memory. It is connected with 1Gb network.

*1) **Workloads**:* We evaluate H^{2}KV by testing various settings for value size, read-write ratio and degree of hotspot skewness. The size of key used in our paper is 16B. For value size, because the value is small in the production environment, we apply 32B to 1KB for evaluation. The total number of key-value pairs in our evaluation is 20M. Our H^{2}KV is designed for hotspot workload. Therefore, we use the popular YCSB benchmark, which is confirmed to match the hotspot access characteristics in real scenarios. The key popularity of YCSB is zipfian distribution and the default hotspot skewness is 0.99. We also evaluate H^{2}KV with other skewness degrees. We measure performance with different read-write ratio. Specifically, we use three patterns including (1) 50% `Set` + 50% `Get`, (2) 5% `Set` + 95% `Get` and (3) 100% `Get`. They correspond to A, B, and C workloads in YCSB benchmark [19] respectively.

*2) **Test tools**:* We use YCSB benchmark to generate workloads. Because the original YCSB client does not support evaluating multiple instances at the same time, we implement an efficient client using `hiredis` [28]. The client has 16 threads and is connected with H^{2}KV through TCP. To improve performance, we use batched sending strategies. The batch size is 10.

*3) **Compared systems**:* We compare H^{2}KV with four counterparts. (1) *Cluster*, a cluster without fault tolerance. Same with H^{2}KV, we deploy 3 data nodes for it. (2) *PBR*, a cluster that uses PBR for fault tolerance. To maintain the same fault-tolerant level with H^{2}KV, we use 3 data nodes and each data node is equipped with 2 backup nodes (6 backup nodes in total). (3) *Block-EC*, a cluster that uses erasure coding for fault tolerance. Same with H^{2}KV, it also applies block-granularity erasure coding. (4) *Basic-EC*, a cluster that also uses erasure coding for fault tolerance. But the encoding/decoding granularity is key-value pair. Both *Block-EC* and *Basic-EC* leverage Reed-Solomon code. To match the fault-tolerant level with H^{2}KV and *PBR*, they use RS(5,3) which contains 3 data nodes and 2 parity nodes. For a fair comparison, these four systems are also based on Redis. For *Cluster* and *PBR*, we use the official implementation in Redis. For *Block-EC* and *Basic-EC*, we implement them ourselves as efficiently as possible.

B. Memory Consumption

In this section, we will evaluate the memory consumption of H^{2}KV compared with other fault-tolerant methods. We first calculated the theoretical storage redundancy of PBR, erasure coding and H^{2}KV as shown in Table I. In theory, H^{2}KV has obvious advantages compared with PBR and increases acceptable storage redundancy compared to erasure coding. In actual scenarios, the amount of hot data is usually small, therefore, we set the maximum amount of data (i.e. hot_{max}) using PBR to 15% and 10% respectively and calculate the corresponding memory usage. when hot_{max} is 10%, H^{2}KV theoretically saves 120% redundancy than PBR.

TABLE I: Theoretical memory redundancy when tolerating at most 2 failures. H^{2}KV (a%) represents a% of total data in H^{2}KV using PBR while 1-a% of data using erasure coding.

Methods	PBR	RS(5,3)	H^{2}KV(15%)	H^{2}KV(10%)
Redundancy	300%	167%	187%	180%

To prove the redundancy advantage of H^{2}KV, we measure the real memory consumption and the results are shown in Figure 10. Apart from *PBR* and *Block-EC*, we also measure Cocytus [10] (denoted by *Cocytus-EC*), which uses erasure coding for value and PBR for key. The memory usage of H^{2}KV is between PBR and erasure coding. When value size is 64B, the real memory redundancy of H^{2}KV is 191% (when hot_{max} is 10%). This result is close to theoretical value and the increased part is mainly brought by metadata (e.g. block

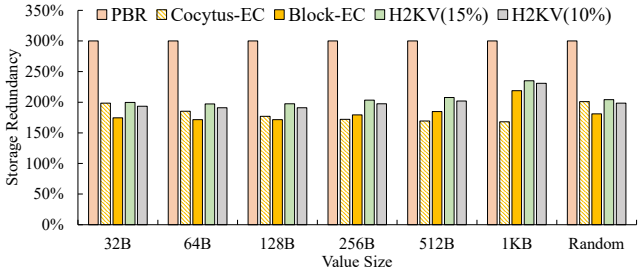


Fig. 10: Memory consumption.

metadata). Compared with *PBR*, H^{2}KV only uses 63% to 77% memory when value size ranges from 32B to 1KB. *Block-EC* has low memory usage when value size is small and its memory consumption is less than Cocytus-EC. When value size is 32B, the memory redundancy of *Block-EC* is 170%, which is almost same with the theoretical value. *Cocytus-EC* leverages PBR to ensure fault-tolerant for keys. Therefore, it needs more memory space than pure erasure coding. This phenomenon is especially obvious under the small granularity value. As the growth of value size, the storage redundancy of *Block-EC* increases. This is brought by the memory hole of block management. For example, when value size is 1KB, the space required to store a key-value pair is 1044B, including 1KB value, 16B key, 2B *keyhdr* and 2B *valhdr*. Because the block size we used is 4KB, a block can only hold 3 key-value pairs, incurring a memory hole close to 1000B. The overhead caused by memory holes can be eliminated by increasing the block size and dividing blocks into more categories. Because H^{2}KV also use block granularity erasure coding, it also has more memory consumption for large value size. But H^{2}KV still has an advantage compared to *PBR* when value size is 512B and 1KB. We also test random value size ranging from 32B to 1KB. H^{2}KV(10%) uses 31% less memory than *PBR* and it only uses 13% more memory than *Block-EC*.

C. Overall Performance

In this section, we will measure the read/write OPS (operations per second) of H^{2}KV (10% PBR). We do not compare Cocytus because it is based on memcached [2]. The architectures of memcached and Redis are different, resulting in large differences in the performance of their systems themselves. Directly comparing Cocytus and H^{2}KV cannot reflect the efficiency of their fault tolerance mechanisms. Actually, Cocytus needs to encode and decode every time data is updated, so its write performance is close to *Block-EC*. Figure 11 shows the performance under YCSB A. The performance of *Block-EC* decreases as the value size becomes larger. There are two reasons. First, in our implementation, we use a fixed-size sending list (see Section IV-B3) to limit the number of blocks for asynchronous encoding. By default, the link list can save up to 4MB of data. With a larger value, the sending list becomes full more easily. In this case, the system needs to wait for the previous encoding to complete, degrading performance. Second, with a larger value, the total size of dataset increases. This requires more encoding calculations and also reduces performance. Our H^{2}KV obviously outperforms *Block-EC*. With a 32B value, it is 51% higher than *Block-EC*. For the same reasons as *Block-EC*, the performance of H^{2}KV also decreases at a large value. However, it is still several times better than *Block-EC*. This demonstrates the effectiveness of using PBR for hot data. To improve the performance of H^{2}KV, we can increase the size of sending list, which can alleviate data waits in asynchronous encoding. We set the size to 80MB and measure it (denoted by *H^{2}KV-Improve*). *H^{2}KV-Improve* has comparable OPS with *PBR* and is far superior to *Block-EC*.

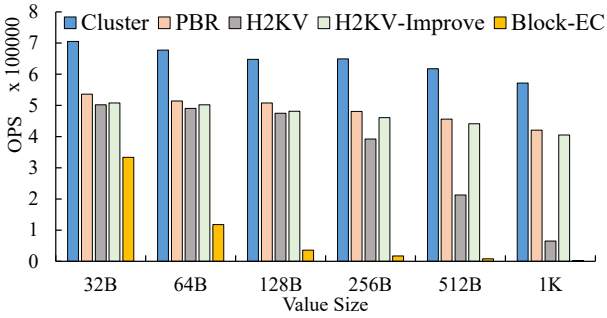


Fig. 11: Performance under YCSB A.

Then we discuss the observations under YCSB B as shown in Figure 12. With a read-heavy workload (only 5% `Set`), *PBR*, H^{2}KV and *Block-EC* have close performance when value size is smaller than 256B. The slightly lower performance of H^{2}KV is because we modified the hash index for storing hotness state. When the value becomes larger, *Block-EC* suffers from performance degradation for the same reasons under YCSB A.

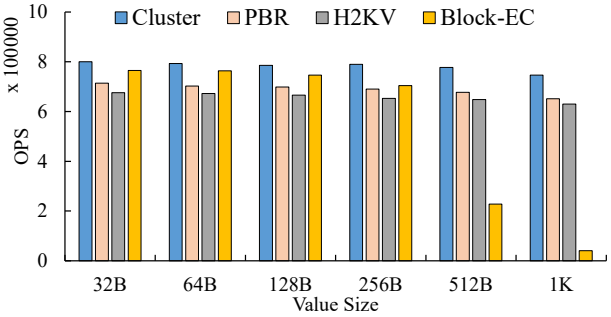


Fig. 12: Performance under YCSB B.

Under YCSB A and YCSB B, we measure performance of only using the data nodes to process the requests. In fact, the backup nodes can also be used to handle `Get`, that is, multi-node parallel read. Both *PBR* and H^{2}KV support this optimization. Under YCSB C (Figure 13), we measure H^{2}KV with multi-node parallel read. Because the backup nodes of H^{2}KV store most of hot data, they can serve most requests, improving performance significantly. In detail, H^{2}KV outperforms *Cluster* by up to 79% and *Block-EC* by up to 81%.

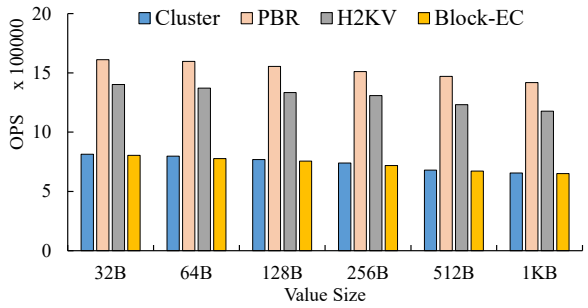


Fig. 13: Performance under YCSB C.

Both *Cluster* and *Block-EC* can only use data nodes to handle read requests. Compared with *PBR*, H^{2}KV is just 16% lower than it. The performance gap is caused by the inability of the backup nodes to handle cold data read requests.

In summary, by leveraging hybrid fault tolerance based on hotspot awareness, H^{2}KV can achieve much higher OPS than *Block-EC*. Meanwhile, the performance of H^{2}KV and PBR are comparable.

D. Efficiency of Block-granularity Erasure Coding

In this section, we evaluate the performance of *Basic-EC*, to explore the efficiency of using blocks for erasure coding. Table II shows the normalized OPS of *Basic-EC* to *Block-EC*. Because the overhead of erasure coding only impacts `Set`, we conduct evaluations with two `Set` ratios.

TABLE II: Performance of *Basic-EC* normalized to *Block-EC* (i.e. *Block-EC* is 100%).

Value Size	32B	64B	128B	256B	512B	1KB
100% `Set`	0.2%	0.3%	0.5%	1.9%	8.3%	52.6%
50% `Set`	0.8%	0.8%	0.9%	1.9%	8.0%	56.9%

The performance of *Block-EC* far exceeds that of *Basic-EC*. This trend is more obvious when the value size is small. There are two reasons for this. First, the encoding efficiency is poor at small granularity (see Figure 3), dragging down the overall performance of *Basic-EC*. On the contrary, *Block-EC* leverages block for large-grained encoding. Second, for *Basic-EC*, it generates a network communication between parity nodes and data nodes when a key-value pair is encoded, while *Block-EC* packs multiple small key-value pairs into a block and sends them in one network communication. Therefore, the number of network communicates of *Basic-EC* far exceeds *Block-EC*, causing performance degradation.

As the value size increases, the performance gap between *Block-EC* and *Basic-EC* continues to decrease. on the one hand, the larger value improves the encoding efficiency of *Basic-EC*, making it close to using big blocks. On the other hand, the larger value reduces the number of key-value pairs loaded in a block. The number of network communications generated by the two also becomes close. In extreme cases, *Block-EC* and *Basic-EC* will achieve the same performance when the value size and block size are the same.

E. Efficiency of Hotspot Aware Method

In this section, we evaluate the efficiency of our proposed hotspot aware method. H^{2}KV applies a three-level filter mechanism (denoted by *TLF* in our evaluation). We compare it with the original cache evict algorithm (denoted by *base*). We also compared two variants of the *TLF* mechanism, *TLF (w/o cold back)* and *TLF (hot start)*. *TLF (w/o cold back)* does not allow convert cold data to warm data. *TLF (hot start)* initializes new data to hot state instead of warm state. We record the access frequency hit rate under different skewness degrees. The access frequency hit rate indicates the ratio between the number of accesses can be processed by the filtered hotspot data to the number of accesses that can be processed by the actual top-hot_{max} data. We conduct an evaluation under two skewness, 0.99 and 0.88. With 0.99 skewness (default setting of YCSB), the hot_{max} is set to 10% and for 0.88 skewness (80% of accesses touch 20% of data), the hot_{max} is 15%.

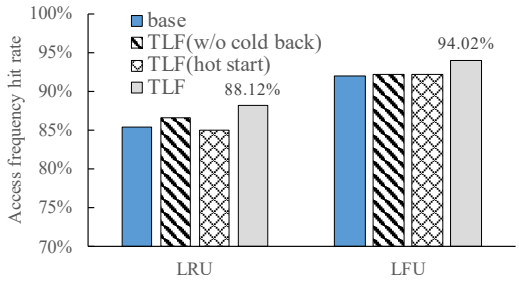


Fig. 14: Access frequency hit rate with 0.88 skewness.

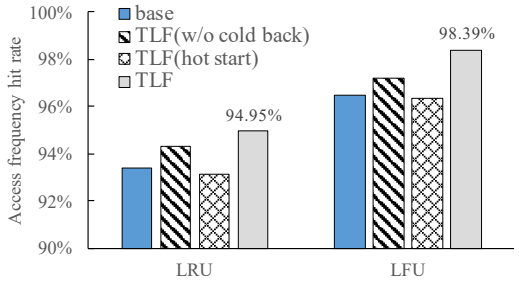


Fig. 15: Access frequency hit rate with 0.99 skewness.

Figure 14 presents the measurement results. Applying LFU as the basic candidate selection method in H^{2}KV has an advantage over LRU. This is because the sampling LFU of Redis uses an attenuation mechanism of access frequency, which improves the hit rate compared with LRU. The access frequency hit rate of *TLF* is higher than base algorithms varying different skewness. When skewness is 0.99, *TLF* screened out almost all the hotspot data. In this case, the access frequency hit rate is 98.39%. *TLF (w/o cold back)* and *TLF (hot start)* have lower access frequency hit rate than *TLF*. *TLF (w/o cold back)* lacks data conversion from cold to hot. Once a hot item is misjudged and enters into cold state, it cannot be promoted when revisited. This degrades the efficiency of hot

data filter. As for *TLF (hot start)*, because it places new key-value pairs into hot state, the filter policy is unable to rapidly convert less-accessed data into cold state. *TLF* initializes new data as warm state that can quickly identify cold data and demote them. Meanwhile, it promotes the data whose hotspot value is greater than T from cold state, improving performance further. The above evaluation and analysis imply the efficiency of three-level filter mechanism.

F. Recovery

In this section, we measure the recovery performance of H^{2}KV. Because the amount of hot data in H^{2}KV affects recovery, we apply two skewness degrees with different hot_{max}. They are 0.99 skewness with 10% hot_{max} and 0.88 skewness with 15% hot_{max}. We simulate a single point of data node failure in H^{2}KV with 16B key and 32B value.

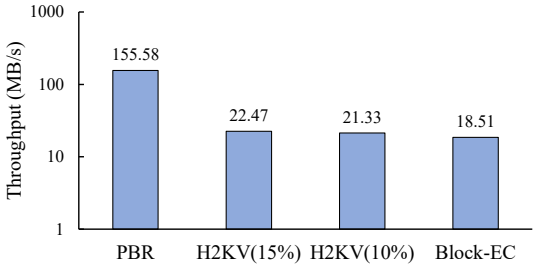


Fig. 16: Recovery throughput.

We first illustrate the results of recovering total data. The recovery throughput is shown in Figure 16. The performance gap between *Block-EC* and *PBR* is large. The Data can be reconstructed in *PBR* with only one network transfer. However, *Block-EC* not only needs more network transmission, but also decoding calculation. The recovery throughput of H^{2}KV is between *PBR* and *Block-EC*. H^{2}KV (15%) has a higher throughput than H^{2}KV(10%). This implies the more data H^{2}KV uses PBR for fault tolerance, the faster the recovery speed. We also notice that H^{2}KV is still much slower than *PBR*, this is because most data within H^{2}KV is cold and it is restored by erasure coding.

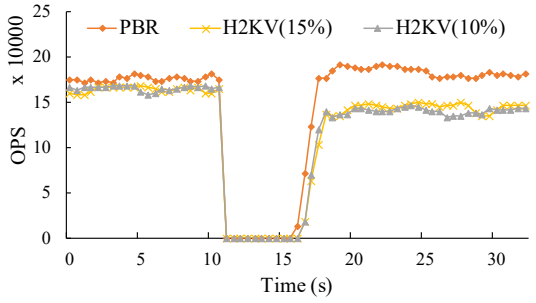


Fig. 17: Real-time OPS when conducting recovery.

Although the recovery throughput of H^{2}KV has a great gap with *PBR*, the two-stage recovery mechanism of H^{2}KV can make it restore service quickly. Figure 17 presents the real-time read-only IOPS before and after recovery. In the 10th second, the system has a single point of failure, and the IOPS quickly drops to 0. Then *PBR* will elect a new node as the primary node and continue request processing. H^{2}KV also performs a simple primary-backup switch on the first recovery stage. Because the backup nodes in H^{2}KV store frequently accessed data, it can handle most of users' requests. For example, for H^{2}KV (10%) under 0.99 skewness, after recovering in the first stage, it can achieve 85% of IOPS before the crash. At the same time, the recovery of cold data in the second stage is performed in the background without affecting service performance.

VI. Related Work

Fault-tolerant storage systems. Some in-memory KV stores apply fault-tolerant methods. Redis [1] and RAMCloud [29] use local disks to save data replicas so that the in-memory data can be recovered after a crash. However, this fault tolerance method breaks down when the local node fails. A better way to tolerate faults is distributed fault tolerance. Cocytus [10] applies replication for keys and leverages erasure coding for values. However, it can suffer performance degradation when the key-values are updated frequently. Both MemEC [30] and EC-Cache [11] apply erasure coding for all data. Although MemEC optimizes performance for small granularity encoding, the network and calculation overhead still limit the efficiency. BCStore [25] and GU [31] also use erasure coding, they leverage batch processing to improve the write performance. ERP [32] proposed a replication placement strategy based on data popularity, reducing the data transmission cost. H^{2}KV optimizes fault tolerance through hotspot awareness, retaining the advantages of both PBR and erasure coding. There are some fault-tolerant work on other systems. ER-Store [33] optimize distributed databases with hybrid fault-tolerant mechanism. WarmCache [34] proposes a hybrid fault-tolerant method based on distributed file systems. Our work focuses on KV stores. Unlike the above-mentioned systems, the size of key-value pairs is very small, resulting in low encoding efficiency. H^{2}KV solves this problem using block structure.

Hotspot awareness. There are numerous prior studies that apply hotspot awareness mechanism. Hotring [5] and VIP-hashing [23] optimize hash tables for hotspot workloads. They promote the frequently accessed items to the head of collision chains to reduce memory access to hot data. To filter hot items, a sampling method is applied. ZExpander [35] distinguishes cold data and compresses them to save memory footprint. Jin et al. [24] proposed an in-memory KV store using hybrid memory, which uses cheap persistent memory to store cold data and places hot data in DRAM. The three-level filter mechanism of H^{2}KV can accurately distinguish hot and cold data, making it use hotspot awareness to improve performance.

Erasure code. Apart from RS code used in this paper, there are various erasure coding types, including EVENODD [36], RDP [26], and STAR code [37]. These codes also can be applied in H^{2}KV. Some work conducted efficiency optimization

for erasure coding. Plank et al. [38], Gibraltar et al. [39] and Zhou et al. [40] optimize the computation of erasure coding using technologies like SIMD, GPU and etc. Some previous works [41]–[44] improve the transmission efficiency of erasure coding. These optimizations can also be adopted by H^2KV.

VII. Conclusion

In this paper, we propose a hybrid fault-tolerant in-memory key-value store, H^2KV. H^2KV utilizes the hotspot aware method to separate hot data and cold data. By adopting different fault tolerance schemes for data with different hotness, H^2KV can achieve both space efficiency and high performance. We implement H^2KV based on popular Redis and evaluate it with various workloads. The results show that H^2KV uses about 30% memory less than PBR and performs close to it. Compared with using erasure coding, H^2KV has more than 51% performance improvement.

Acknowledgements

This research is supported by National Science Foundation of China under Grant 62272252 and 62272253, the Key Research and Development Program of Guangdong under Grant 2021B0101310002, and the Fundamental Research Funds for the Central Universities.

References

[1] "Redis." https://redis.io/.

[2] "Memcached." http://memcached.org/.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, p. 53–64, 2012.

[4] J. Yang, Y. Yue, and K. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 191–208, 2020.

[5] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, "Hotring: A hotspot-aware in-memory key-value store," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pp. 239–252, 2020.

[6] G. Wang, L. Zhang, and W. Xu, "What can we learn from four years of data center hardware failures?," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 25–36, IEEE, 2017.

[7] A. Goel, B. Chopra, C. Gerea, D. Mátáni, J. Metzler, F. Ul Haq, and J. Wiener, "Fast database restarts at facebook," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 541–549, 2014.

[8] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," *Distributed systems*, vol. 2, pp. 199–216, 1993.

[9] H.-Y. Lin and W.-G. Tzeng, "A secure erasure code-based cloud storage system with secure data forwarding," *IEEE transactions on parallel and distributed systems*, vol. 23, no. 6, pp. 995–1003, 2011.

[10] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang, "Efficient and available in-memory kv-store with hybrid erasure coding and replication," *ACM Transactions on Storage (TOS)*, vol. 13, no. 3, pp. 1–30, 2017.

[11] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "Ec-cache:load-balanced, low-latency cluster caching with online erasure coding," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 401–417, 2016.

[12] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 307–320, 2006.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 29–43, 2003.

[14] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pp. 1–10, Ieee, 2010.

[15] "HDFS RAID." http://wiki.apache.org/hadoop/HDFS-RAID.

[16] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pp. 15–26, 2012.

[17] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[18] C. Canudas-de Wit, F. R. Rubio, J. Fornes, and F. Gómez-Estern, "Differential coding in networked controlled linear systems," in *2006 American Control Conference*, pp. 6–pp, IEEE, 2006.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, 2010.

[20] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti, "Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification," in *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, pp. 65–78, 2019.

[21] K. M. G. James S. Plank, "jerasure." https://github.com/tsuraan/Jerasure.

[22] L. Cui, K. He, Y. Li, P. Li, J. Zhang, G. Wang, and X. Liu, "Swapkv: A hotness aware in-memory key-value store for hybrid memory systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 1, pp. 917–930, 2023.

[23] A. Kakaraparthy, J. M. Patel, B. P. Kroth, and K. Park, "Vip hashing: Adapting to skew in popularity of data on the fly," *Proc. VLDB Endow.*, vol. 15, p. 1978–1990, sep 2022.

[24] H. Jin, Z. Li, H. Liu, X. Liao, and Y. Zhang, "Hotspot-aware hybrid memory management for in-memory key-value stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 779–792, 2020.

[25] S. Li, Q. Zhang, Z. Yang, and Y. Dai, "Bcstore: Bandwidth-efficient in-memory kv-store with batch coding," *Proc. of IEEE MSST*, pp. 14–16, 2017.

[26] J. Feng, Y. Chen, D. Summerville, and Z. Su, "An extension of rdp code with parallel decoding procedure," in *2012 IEEE Consumer Communications and Networking Conference (CCNC)*, pp. 154–158, IEEE, 2012.

[27] L. Xiang, Y. Xu, J. C. Lui, and Q. Chang, "Optimal recovery of single disk failure in rdp code storage systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 1, pp. 119–130, 2010.

[28] "Hiredis." https://github.com/redis/hiredis.

[29] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, aug 2015.

[30] M. M. Yiu, H. H. Chan, and P. P. Lee, "Erasure coding for small objects in in-memory kv storage," in *Proceedings of the 10th ACM International Systems and Storage Conference*, pp. 1–12, 2017.

[31] J. Xia, J. Huang, X. Qin, Q. Cao, and C. Xie, "Revisiting updating schemes for erasure-coded in-memory stores," in *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pp. 1–6, IEEE, 2017.

[32] B. Xu, J. Huang, X. Qin, and Q. Cao, "Traffic-aware erasure-coded archival schemes for in-memory stores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 12, pp. 2938–2953, 2020.

[33] Z. Li, C. Xiao, and J. Qian, "Er-store: A hybrid storage mechanism with erasure coding and replication in distributed database systems," *Sci. Program.*, vol. 2021, jan 2021.

[34] B. A. Ignacio, C. Wu, and J. Li, "Warmcache: A comprehensive distributed storage system combining replication, erasure codes and buffer cache," in *Green, Pervasive, and Cloud Computing*, (Cham), pp. 269–283, Springer International Publishing, 2019.

[35] X. Wu, L. Zhang, Y. Wang, Y. Ren, M. Hack, and S. Jiang, "Zexpander: A key-value cache with both high performance and fewer misses," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, 2016.

[36] M. Blaum, J. Brady, J. Bruck, and J. Menon, "Evenodd: An efficient scheme for tolerating double disk failures in raid architectures," *IEEE Transactions on computers*, vol. 44, no. 2, pp. 192–202, 1995.
[37] C. Huang and L. Xu, "Star: An efficient coding scheme for correcting triple storage node failures," *IEEE Transactions on Computers*, vol. 57, no. 7, pp. 889–901, 2008.
[38] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast galois field arithmetic using intel simd instructions," in *FAST*, pp. 299–306, 2013.
[39] M. L. Curry, A. Skjellum, H. Lee Ward, and R. Brightwell, "Gibraltar: A reed-solomon coding library for storage applications on programmable graphics processors," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 18, pp. 2477–2495, 2011.
[40] T. Zhou and C. Tian, "Fast erasure coding for data storage: A comprehensive study of the acceleration techniques," *ACM Transactions on Storage (TOS)*, vol. 16, no. 1, pp. 1–24, 2020.
[41] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE transactions on information theory*, vol. 56, no. 9, pp. 4539–4551, 2010.
[42] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads.," in *FAST*, p. 20, 2012.
[43] Y. Hu, L. Cheng, Q. Yao, P. P. Lee, W. Wang, and W. Chen, "Exploiting combined locality for wide-stripe erasure coding in distributed storage," in *FAST*, pp. 233–248, 2021.
[44] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, "Partial-parallel-repair (ppr) a distributed technique for repairing erasure coded storage," in *Proceedings of the eleventh European conference on computer systems*, pp. 1–16, 2016.