SpotKV: Improving Read Throughput of KVS by I/O-aware Cache and Adaptive Cuckoo Filters

Yi Liu, Ruilin Zhou, Yuhang Gan, and Chen Qian
University of California, Santa Cruz

***Abstract*—LSM tree based stores are a popular database design in modern persistent storage systems due to their efficient writes with sorted keys. However, this hierarchical log structure suffers from extensive read amplification because multiple disk accesses are required when it searches for a key. Recent optimizations of LSM trees propose caching hot keys to reduce I/Os mainly based on their access frequencies. However, our empirical studies show that keys are different in I/O costs, which should also be considered in the caching policy: caching key-value pairs with high I/O cost can effectively improve query latency. In addition, false positives incurred by the Bloom filters in LSM trees introduce a large overhead to access SSTables because the queried keys do not exist. In this work, we design and implement SpotKV, which resolves the above two problems in an LSM tree store by proposing two memory-efficient data structures, weighted Count-Min sketch for access and I/O-aware cache admission and dynamic-seed Cuckoo filters for eliminating false positives, to improve data lookup throughput. We implement SpotKV on Google's LevelDB v1.20. From extensive experimental evaluations, SpotKV achieves 1.2-$3.0\times$ read throughput while using the same or smaller memory, compared with several state-of-the-art LSM tree stores under the read-heavy workloads of the YCSB benchmarks.**

***Index Terms*—LSM tree, KV store, Caching, Cuckoo filter, Count-Min sketch**

I. Introduction

Persistent key-value stores are increasingly essential nowadays as a widespread solution for handling extremely large-scale data. Log-structured merge tree (LSM tree) based store is widely used as a fundamental storage infrastructure in a large variety of applications in datacenters, like Google's LevelDB [1], Meta's RocksDB [2], [3], and Apache Cassandra [4]. Based on significant advantages in dealing with write-intensive workloads, the LSM tree store has become the backbone storage backend in production [5]–[7].

LSM tree stores maintain key-value (KV) pairs in two components, one resides in memory to buffer the new KV writes and updates called MemTable and Immutable Memtable. The other is kept in secondary storage, where most KV pairs in the database are packed into SSTables and organized in multiple levels (e.g. from L_0 to L_n) whose sizes increase exponentially. We call the first level with the least number of SSTables as the lowest layer and the last one as the highest layer. Each SSTable contains an array of KV pairs in a sorted ordering and corresponding metadata like indexing data within the SSTable. In each layer, once the number of SStables exceeds its size limit, one of the SSTables would be selected to merge into the next layer with SSTables having the key range overlapping with it. When there is a data lookup request, it has to search the SSTables in each layer until it can find it. During the process, the membership query filters (e.g. Bloom filters) will be used to check if the SSTable contains the targeted KV pair. Note that some read requests for non-update-intensive KV pairs have to go through multiple layers to get satisfied because they are compacted to the bottom as time goes on.

A variety of designs [8]–[11] have been proposed to improve data query throughput in LSM tree stores. These schemes focusing on fast data locating and accessing in SSTable can be categorized into two types – cache-based and filter-based. UniKV [8] uses a Cuckoo hashing table to maintain all KV pairs' positions in L_0, where the KV pairs are not sorted strictly across SSTables. Thus, keeping each key's position in memory could reduce latency for locating SStable in L_0 because binary search does not help in this layer. AC-Key [10], as a representative of hybrid caching work, combines three different caching approaches – key-value, key-pointer and block – to accelerate the process of finding target KV with a "ghost" cache. To lower the false positive rate brought by hashing collision in Bloom filters, ElasticBF [11] tries to assign more bits-per-key to hot data blocks, whose hotnesses are identified by their accessing time intervals. The main thought behind these methods is trading memory space for fast lookups.

However, we realize that existing solutions often use the query frequency to characterize the "hotness" of a key [8], [10], [11]. Based on our empirical studies, presented in Section II, show that a key with a lower query frequency but at a higher level might cause more total I/O times than another key with a higher query frequency but at a lower level. Since the average I/O time is the main metric to predict the query latency, an ideal metric to characterize the hotness should reflect the average I/O time of a key rather than just the access frequency.

On the other hand, improving read throughput by lowering the false positive rate in the Bloom filters for SStables is an effective way. Assigning more bits per key in the filters for SSTables/blocks containing hot keys [11], [12] is a common approach to achieve a low false positive rate because there is a consensus that data accessing workload is usually skewed in real scenarios [11], [13], [14]. Just like the bits-per-key is a key parameter for a single Bloom filter for its memory usage and false positive rate, the fingerprint length is a parameter that can adjust the expected false positive rate when it comes to a Cuckoo filter, and the false positives originate from the fingerprint collisions. Given the fixed memory budget,

ElasticBF [11] varies the bits-per-key in different Bloom filters for different data blocks, and Monkey [12] allocates an exponential distribution of bits for keys in the different layers' Bloom filters. However, increasing bits per key certainly increases memory cost. Is there a way we can lower such membership query filters' false positive rate further besides just allocating more bits?

In this paper, we introduce SpotKV, a solution designed to identify KV pairs' hotness in disk I/O efficiency and further lower false positives in membership query filters by incorporating dynamic seeds within a Cuckoo filter. The newly designed Cuckoo filter can change seed values for each bucket to reduce known false positives. The idea is motivated by the adaptive Cuckoo filter [15], a recently proposed design that allows the filter to eliminate known false positives: when a false positive is detected, the data structure can be adjusted such that the same false positive will not happen again. But the adaptive Cuckoo filter does not have a seed for each bucket like that in SpotKV. We believe this idea is a great fit to eliminate potential false positives of hot queries to an LSM tree: hot queries can be detected. The filters can be changed to correct the hot queries if they are false positives. This combination of strategies can benefit the KV store by caching for hot keys and reducing disk I/Os incurred by filters' false positives.

However, to complete the hotness identification task in an I/O-oriented perspective, and make precise optimization to replace caching with Bloom/Cuckoo filters, the following key problems have to be solved: (1) How to accurately measure and identify the hotness of KV pairs in terms of their I/O costs? (2) How to lower false positives in Cuckoo filters without assigning more bits to the data structure? SpotKV carefully addresses these key issues by developing two pluggable techniques – an I/O-aware Count-Min sketch and adaptive hot-key Cuckoo filters – to improve the LSM tree store's data query throughput in a memory-efficient way based on our observations: (1) the caching admission policy that combines frequency and I/O should be used for the LSM tree based KV store, and (2) there are repeating false positives of hot queries that happen in data stores.

We emphasize that the designs in SpotKV are compatible with most existing optimizations on LSM tree structures. They can replace current relevant modules in different LSM tree variants to achieve higher data query performance.

Our contributions in this paper are summarized as follows:

(1) We conduct empirical studies to identify that including I/O cost is necessary for cache management of an LSM tree store.
(2) We present a fast and memory-efficient structure to select read-intensive key-value pairs according to disk I/O overheads instead of general frequency-based identification in the LSM tree store.
(3) We propose a novel scheme to make a traditional membership query filter to reduce the false positives in an advanced step and improve lookup throughput.
(4) We have implemented a prototype with the two key designs in SpotKV based on Google's LevelDB v1.20 and conducted extensive experiments to illustrate the advantages of SpotKV against several baselines for read-heavy workloads.

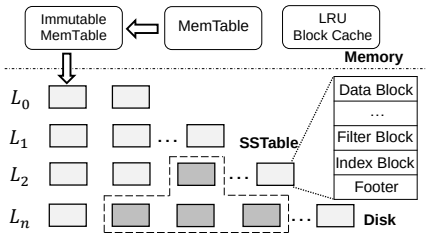


Fig. 1: The LSM tree structure in LevelDB.

II. Background and Motivation

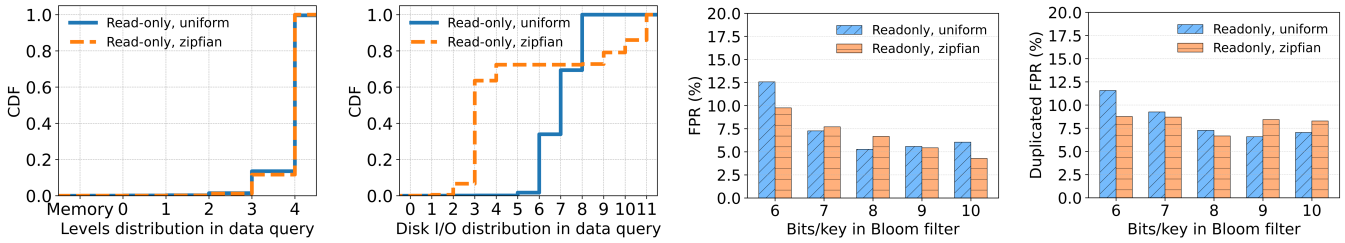
In this section, we first introduce LSM tree based KV stores with an example of LevelDB [1]. We then show our observations from the empirical results and analysis of the data lookup workload.

A. Log Structured Merge Trees

Fig. 1 illustrates the structure of an LSM tree, which consists of both in-memory and on-disk components. A MemTable and an Immutable Memtable serve as the main cache space in the memory. On the disk, a set of SSTable (Sorted string table) groups storing data in a persistent way are ordered and organized in n layers, from L_0 to L_n. The number of SSTables in each layer increases exponentially by a factor named *Layer Ratio* r, where r equals 10 in LevelDB and 8 in RocksDB [2] by default. If we refer the number of SSTables in L_i is m_i, that means $m_{i+1} \approx r \cdot m_i$, where $i + 1 \leq n$. Except for SSTables in L_0, all key-value pairs are sorted strictly inside an SSTable and among SSTables in a layer without any overlapping.

An LSM tree store works as follows. All incoming KV write operations are first served by the MemTable, which acts as a cache buffer. When the MemTable is filled up, it would be converted to an Immutable Memtable on which we could not write more data and then a new empty MemTable will become a new buffer for incoming operation requests. When the Immutable Memtable is flushed from memory to disk, all KV pairs in it are sorted and rewritten again into a new SSTable, which is how the new SSTable is built in L_0. SSTable is a piece of data log where KV are organized on it with metadata. We have m_0 SSTables in L_0, and the key ranges among them are not strictly sorted because the Immutable Memtables are converted to SSTables at different times. Compaction is a merging operation that happens among SSTables in two continuous layers. When the number of SSTables in L_i reaches the set threshold, an SSTable in L_i is chosen and merged with all SSTables in L_{i+1} with key range overlappings. Then all KVs in these SSTables are sorted and written into a new SSTable in L_{i+1}.

The read operation starts to find the target KV from Memtable to Immutable Memtable in memory first. If there is no match, the LSM tree will search the disk layer by layer from L_0 to L_n because the newly updated data are stored in the lower layer (like L_0). The search stops immediately



(a) Number of queries for different level

(b) Number of queries with different Disk I/O times

(c) False positive rate

(d) Repeated false positives ratio

Fig. 2: Observations from YCSB read-only workload.

once it gets the KV in a certain layer. For m_i SSTables in L_i, it uses binary search to locate the SSTable candidates that may contain the target KV. After it locates an SSTable, the Bloom filter block will be accessed first as a membership query structure to check if the target KV exists in it – Bloom filters have false positives.

B. Motivations

Many performance optimizations on LSM tree stores have been proposed [10], [11], [16] by treating hot data (in the unit of key, SSTable, or block) and cold data differently. A hotness identification scheme is a prerequisite for separating hot data from the whole workload set. For example, L2SM [17] uses `HotMap` to calculate the hotness value of all SSTables and optimize the compaction of hot SSTables to save the computation overheads.

An LSM tree store manages all KV pairs through SSTables in multiple layers, which incurs lots of I/O overhead [18] when it comes to point data lookup operation (especially compared to $B+$ tree). Caching hot KV pairs [10] enables the database to access read-heavy data faster and consume less CPU and time resources without going through SSTables stored in disk. However, how the hot keys are identified and selected in an LSM tree store to reduce I/O overhead is a challenging research problem. For example, TRIAD [16] proposes to get the most K popular key-value pairs in each MemTable with a top-K algorithm, which can recognize the update-intensive data instead of read-intensive pairs. Time-based hotness identification is also a common approach used in recent work to complete the task. ElasticBF [11] maintains a table to track all data block's `lifeTime` values which represent the time they stay in the LSM tree. The data segment that is accessed more than two times within a time duration `expiredtime` will be inserted into the cache. The weakness is that all keys in one data segment share one hotness value, which is not accurate for individual KV pairs, especially when the memory size for caching is limited. We know it is a subtle tradeoff because memory cost will increase significantly if we maintain the `lifeTime` values for all single KV pairs. Thus, a memory-efficient hotness identification scheme is required for estimating the hotness value for individual keys.

In addition, data are stored in different levels of the LSM tree. For a KV pair that is stored at a higher level, accessing it requires more I/O cost. Hence we argue that the storage level of a KV pair needs to be considered in the cache strategy: a key with a lower query frequency but at a higher level might cause more total I/O times than another key with a higher query frequency but at a lower level. We run a microbenchmark with 1 million read operations on LevelDB [1] v1.20, with 4 levels and 4 SSTables in L_0. Then, we draw the CDF of SSTables accessing times for the workload (with uniform and Zipfian access patterns) based on YCSB [19]. Fig. 2(a) shows the distribution of levels the data lookup reaches in the log scale. A higher level receives significantly more data requests than the lower levels; for example, data lookup requests reaching the highest level (L_4) make up 88% of the uniform workload. From Fig. 2(b), we can see the number of SSTables accessing distribution varies in different data lookup processes. In uniform workload, the data requests that access 5 SSTables make up around 15%, but the requests that access 8 SSTables are 30.6%. Such disk I/O difference motivates us to differentiate them in caching admission and prioritizes the one that incurs more disk I/Os. Thus, a cache admission rule that can take into account the different I/O costs of the KV pair is ideal for the LSM tree store.

Recently, counting sketches [20], a statistical tool for making hotness identification, have been proposed for multiple network applications [13], [21] as a memory-efficient approach. For example, the Count-Min sketch [13] is used to estimate the frequency of incoming data packets with the hashing value of the IP address in it. Motivated by these techniques, we propose to apply sketches on counting the hotness of queried KV pairs in the LSM tree store. However, caching KV pairs with the most frequent KV pair does not necessarily minimize the disk I/O overheads. If a KV pair has a high hotness value, maybe it is not optimized to cache it in terms of the benefit for the database system. The final objective is to improve the read performance by lowering the disk I/Os incurred by the lookup operation. AC-Key also takes into account the different overheads incurred by caching the KV pairs in different layers and makes the KV pair with a high-efficiency factor stay at the cache for a longer time like the weighted LRU cache [22]. In this work, we focus more on how to select hot keys in the cache admission.

To reduce the disk I/Os to SSTable, different LSM tree store implementations [1], [2] usually leverage Bloom filters to check if the queried key exists in the SSTable. However, the false positives in Bloom filters will cause extra disk I/O and CPU resources. We show the false positive rate in Fig. 2(c) when we set the number of bits per key used in the filter from 6 to 10. There are more than 10% false positives shown in both

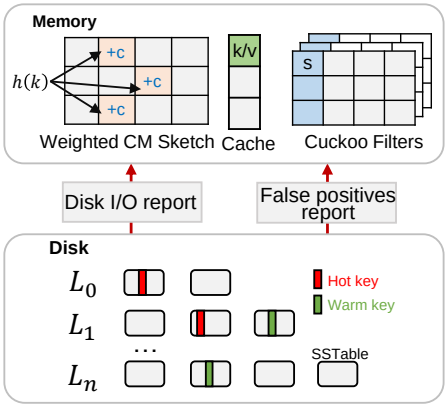


Fig. 3: Overview of SpotKV.

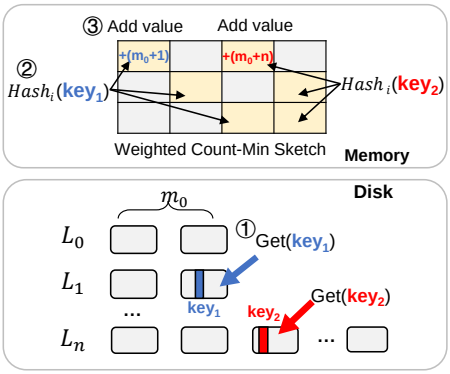


Fig. 4: Weighted Count-Min sketch.

Zipfian and uniform workload when bit-per-key is 6. The more bits we assign to the Bloom filters, the lower the false positive rate is. Given the fixed memory budget, we cannot lower the number anymore. However, we observe that the repeated false positives make up around 7% among all false positives, as shown in Fig. 2(d). Thus, the other motivation of this work is how to reduce/avoid duplicated false positives in a further step. Alternatively, there are many hotkeys in a workload, and it is impossible to cache all of them due to the limited memory space. Is there any way we can avoid their false positives appearing with the membership query filters?

III. Design of SpotKV

We propose SpotKV, a high-throughput and memory-efficient solution for the LSM-tree store, which can potentially save a large portion of I/O overheads with a novel hotness identification scheme. It also can mitigate read amplification by reducing false positives for accessing SSTables.

A. Overview

A key design goal of SpotKV is to leverage a memory-efficient sketch data structure to realize I/O-aware hotness identification for caching hot KV pairs. It also leverages adaptive Cuckoo filters to eliminate false positives on warm keys or the keys where false positives happened before. SpotKV accomplishes this by installing these two optimized modules introduced in this section.

The whole design is illustrated in Fig. 3. In memory, besides MemTable and Immutable MemTable we mentioned in section II, we have a module named *weighted Count-Min (CM) sketch*, which enables SpotKV to identify hot/warm keys based on the I/O overheads they can save for read operations. The hot keys chosen by sketch can be cached for any further applications. The fingerprints of warm keys output by our weighted CM sketch will be kept in an LRU list. Note that only fingerprints will be maintained thus the memory usage will be way less than storing full KV pairs. We also have an array of dynamic seeds-based Cuckoo filters maintained in memory for each SSTable on disk, and they can lower FPR when it tells if the target key is in the SSTable or not. Given the fixed memory budget, our Cuckoo filters can work it out with the list of warm keys' fingerprints as assistance to remove false positives may happen on them.

Note that we put the following two main modules together to work out as SpotKV, they can also be divided and applied to other LSM-tree stores independently without the other on. For example, dynamic Cuckoo filters can be fed with hot keys' fingerprints chosen by a time-based hotness identification scheme.

The workflow of a key lookup operation is as follows. The queried key will be first checked in the cache, MemTable and Immutable MemTable, respectively. If there is no match, then the lookup will go to the SSTables on the disk. Same with LevelDB, it searches every SSTable in L_0 and at most one SSTable in the layer from L_1 to L_n will be checked until the queried key is found. When the lookup process gets the queried KV pair in L_j, after the LSM tree returns the data, the counters in the weighted CM sketch will be updated based on the number of the SSTables this lookup had went through. If the estimation of the key's hotness value in the weighted Count-Min sketch reaches a threshold, then the corresponding KV pair will be cached for future use. If the estimated hotness does not reach the threshold, the fingerprint of the key will be added into the fingerprint LRU list, which will be fed to the adaptive Cuckoo filters as a reference to remove false positives on them.

B. Weighted Count-Min Sketch

To mitigate the disk accessing times consumed by read operations in the LSM-tree structure, we propose to identify read-intensive keys that incurred large I/O overheads from multiple layers and cache them in memory to make future lookups fast. In this section, we are going to illustrate the detailed design of the weighted CM sketch. This sketch provides a fast and memory-efficient hashing scheme to estimate the hotness of each individual key-value pair. Hotness analysis in terms of SSTables or other units of data store is not the focus of our work.

Count-Min sketch. A Count-Min sketch consists of M arrays of W counters, which are initialized to 0. The maximum value each counter can reach depends on the bits assigned to it; if a counter occupies 8 bits and the maximal value it can serve is 255. A set of M hash functions are enabled to hash elements

into random counters in each array. In practice, we usually use one hashing function and M different seeds to realize $H_i(k)$, where $0 \leq i \leq M$. It works as follows. When it comes to a key, it is hashed to different counters in each array with different M hashing functions, and generally a constant value c will be added to them. If you want to estimate the frequency of a key, just access the counter values $(c_0, c_1, .., c_M)$ with M functions again, and the minimal value $min(c_i)$ will be regarded as a close estimation of it. The estimation error e is limited to ϵ with probability of $1 - \delta$, where we assume $M = \lceil e/\epsilon \rceil$ and $W = \lceil ln(1/\delta) \rceil$. In fact, sketch is commonly used to count up data packets with specific IP/MAC addresses in their headers as the keys of hashing functions.

Weighted Count-Min sketch. Provided there are $m_0, m_1, .., m_n$ SSTables in LSM-tree structure from layer L_0 to L_n, where the key-value pairs in L_0 are not strictly sorted across the m_0 SSTables. When data request comes to L_0, it has to search all SSTables because there possibly are keys range overlapping among them. However, it can use binary search to locate that exact one candidate SSTable from L_1 to L_n because all keys are sorted in ascending in layers.

Fig. 4 shows the working process of the weighted Count-Min sketch that estimates the hotness of KV pairs. Same as the traditional Count-Min sketch, it has M arrays of W counters initialized with 0. When it comes to a point lookup request of key_1 (as indicated in blue), our SpotKV will ❶ follow the LSM-tree rules to search target KV pair layer by layer; here we find key_1 in the second SSTable of L_1. ❷ Then we hash key_1 with M different seeds to M counters, spreading all rows in sketch. ❸ We add $c = m_0 + 1$ on the hashed counters as an increment of hotness it brings to this counter, where m_0 refers the number of SSTables in L_0. Then we compare all values in all these M counters and choose the least one as a close estimation of key_1's hotness value. If the smallest estimation exceeds a threshold, the KV pair could be regarded as hot and added to the LRU caching list. Correspondingly, we found key_2 (indicated in red) in the third SSTable of L_n, so we add an increment of $c = m_0 + n$ on the counters hashed by key_2. Additionally, if a key is found in the i-th SSTables in L_0, we just add i on associated counters.

In summary, what the weighted CM sketch actually does is to use the SSTable accessing overheads caused by searching the key as the increment added on counters instead of requested times from the database. When it comes to a read-intensive workload, there are read-intensive keys in high layers of the LSM-tree structure with less chance to be updated and appear in low layers again. Thus, we use the I/O cost and access times simultaneously to refer to how worthwhile to identify it as a hot pair. If we just add a constant for each key, then all KV pairs in different positions of the LSM-tree will be treated with no difference, and the I/O cost is not considered.

Configuring hot KV threshold. The weighted Count-Min sketch enables us to estimate the frequency of the individual KV pair in a fixed period. As time goes on, the value in counters will increase, it does not bode well if we set a fixed integer as a bar to screen hot keys. The threshold we select should be relevant to a key's relative hotness in the sketch. Thus, we defined the hotness of a key as the ratio between the estimated value and the sum of all counter values like:

$$hotness = \frac{min(c_0, c_1, .., c_{M-1})}{\sum counter\ value} \tag{1}$$

Then, no matter whether the sketch is just initialized or almost overflowed, a threshold t $(0 \leq t \leq 1)$ could be chosen to tell if the requested key is hot or not.

Overflow of the sketch. When the value of a counter reaches the maximum it can support, how to enable sketch to serve after overflow is a crucial problem. Given each counter is assigned with 8 bits, the largest estimated value it can record is 255. At that time, all counters in the sketch can continue serving, but this overflowed one.

We leverage two common approaches to let sketch work on after overflow. One where as long as one counter overflows, we will empty all counters and reconstruct it with the KV requests in the future. The advantage is it is fast, and this operation does not incur any heavy overhead, but it will lose all information and former estimations recorded in the past. Usually, after we empty the whole sketch, we will add an initial value to the overflowed counter to refer to it as the most popular counter in the last round.

The other approach to solve this problem is using a sliding window. All keys contributing to their popularity in the sketch will be maintained in a sliding window with time order. When a counter reaches the ceiling value, we start from the left of the sliding window and subtract a constant from each key's associated counters until the overflowed counter's value is taken off by 5%. The benefit is that sliding window-based sketch [23] can always keep the relative popularity indicated in a past period. However, it is expensive to implement this policy because it is easy to get the counter to be overflowed again if it got ovevflowed before, and the cost to remove the KV pairs in the tail of the sliding window also incurs large overheads. Additionally, maintaining keys in a list just for the sketch is not necessary in terms of its memory cost. The sliding window-based sketch does not fit our work. Thus, we choose an "empty scheme" as the default way to handle the overflowed counters in our weight CM sketch.

Configure the sketch size. Since we replace KV request times with its corresponding SSTable accessing cost in the sketch, we can understand our weighted CM sketch in another way: If it takes m times on-disk accessing to find a key, it means key appears m times if we put all KV pairs in one single layer. When it comes to configuring the sketch's size, we can follow the rules in traditional CM sketch, too. Assume the sketch size is $M \times W$, where array number M also refers to the hashing functions we used in sketch. To make sure the estimations in the CM sketch follow $P(x \leq \hat{x} \leq x + \epsilon n) \geq 1 - \delta$, where n is the number of all distinct KV pair requests, ϵn is a tolerant error and δ is confidence probability. Since we are going to apply an empty policy(resetting to 0) to deal with the overflowed counter problem, n will be a number of distinct KV requests within a time round, which starts from an

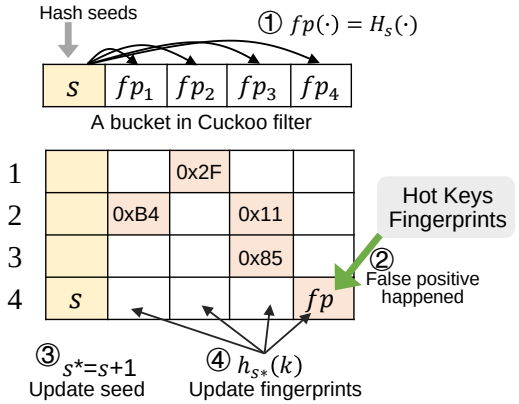


Fig. 5: Dynamic-seed Cuckoo filter.

empty sketch and ends when an overflow happens. Given that the ceiling value in each counter is set to 255 (8 bits), we want to estimate the frequency of 1K KV pair requests in a single time round. With ϵ equal to 0.01, we can calculate the required width W in sketch should be 270. If we hope estimation error could be limited within a difference of $1K \times 10$ with the possibility of 99%, the expected choice for the number of hashing functions M will be 4 or 5.

Overhead. Even if maintaining and managing our weighted CM sketch in memory will cost some resources, but the overheads are very small. Provided our sketch size is $M \times W$, and each counter contains 8 bits for estimation. If we set M as 5 and W as 270, the memory cost will be 1.01KB.

For each point lookup request for k, we will calculate M hashing result with $H_i(k)$. The M memory accessing operations are required to update the value in M associated counters, above which does our sketch incur extra computational overhead. When it comes to an overflowed counter, assigning all values with 0 in the sketch just occupies limited computational resources.

C. Dynamic-seed Cuckoo Filter

To avoid wasted I/O overhead for accessing "wrong" on-disk SSTables, which actually do not have the target KV contained, approximate query structures like Bloom filter are used in LevelDB and Cuckoo filter in RocksDB [2]. LSM-tree stores suffer from the waste of I/O overheads incurred by false positives in filters, even with the help of caching. We propose to mitigate the false positive rate (FPR) of query structures associated with SSTables further with Cuckoo filters and our weighted CM sketch.

Cuckoo filter. SpotKV achieves a lower FPR using the Cuckoo filter for the application of an LSM-tree store, where each SSTable has a membership query structure with it. We introduce (2,4)-Cuckoo filter here because recent works [24], [25] prove it can achieve the maximal load factor in memory. A (2,4)-Cuckoo filter is a table of a number of buckets, each with 4 cells or slots. When we insert a key in the Cuckoo filter, two candidate buckets could be calculated by $b_1 = H(key)$ and $b_2 = H(key) \oplus H(f(key)) = b_1 \oplus H(f(key))$, where $H(\cdot)$ is a hash function and $f(\cdot)$ returns the fingerprint of the input. Then key will be stored in one of the 8 cells of two buckets. If there is no available cell for newly inserted keys, one of the keys in these 8 cells will be kicked out to the other candidate bucket and keep going until all keys could be stored without overflows. Note that what we put in the bucket cells is actually the key's fingerprint instead of the full key, which is the different point with Cuckoo hashing [26]. Thus, when it comes to a key membership query, we just compare the key's fingerprint with 8 possible fingerprints stored in two buckets, as long as there is a match, the Cuckoo filter will return a positive answer that the key may be in it. If there is none of them match, then the key will definitely not be in the filter (no false negative).

Dynamic-seed Cuckoo filter. We propose to use (2,4)-Cuckoo filter as a membership query structure associated with each SSTable. As shown in Fig. 5, we make an adaption on a traditional Cuckoo filter that ❶ we attach a seed for each bucket and use it to calculate the fingerprint of 4 keys stored in it. At first, all seeds are set to 0. All four keys in one bucket share the same seed for the hash function to get its fingerprint, whose length is adjustable in our setting. Note that the two candidate buckets are calculated by two distinct hashing functions H_1, and H_2 and any of them cannot be restored by the exclusive-or operation as suggested in the traditional Cuckoo filters. However, we will keep the full keys while constructing a dynamic Cuckoo filter for a new compacted SSTable while merging. The idea was motivated by the recently proposed adaptive cuckoo filter [15] but the detailed design is different in order to be used by the LSM tree.

Recall that we conclude a list of warm keys' fingerprints (hashed with the seed 0) and they are organized in an LRU approach. The warm keys mainly consist of two parts: (1) When there is a key query, we update it in our weighted CM sketch. If it is not identified as a hot key (the estimated frequency is smaller than the threshold but larger than 0), it will be regarded as a warm key. Then, we will calculate the warm key's fingerprint with the seed of 0 and put it in the LRU list. (2) When a false positive happens, and the bucket seed is 0, we will insert the fingerprint of this key into the list. We will also move the compaction offset pointer to the current SSTable and try to compact this table into the next level first.

Every time we finish constructing the Cuckoo filter with all seeds of 0, we test all warm keys in the list to see if the dynamic Cuckoo filter will give a positive answer back.

There are two occasions that make it have a positive answer, (1) The warm key is indeed in the Cuckoo filter. (2) As shown in Fig. 5, ❷ there is a false positive happened. No matter which one is the real reason for that, what we are going to do is ❸ updating seed 0 to 1 in this bucket. Correspondingly, ❹ we update all fingerprints with new seed. After we update all possible seeds and fingerprints in buckets where false positives may occur with our warm keys, a dynamic seeds-based Cuckoo filter is finished, and it could be used as a structure when a data lookup request makes a membership check with extremely low FPR. Then, we can discard all the

full keys to save memory space and use the Cuckoo filter to serve the membership query in the future.

There is a variant based on the dynamic-seed Cuckoo filter. What we put in the list of warm keys is their full keys instead of fingerprints hashed when it is constructed. The weakness of the fingerprint version is that we can only update seeds once in the buckets with possible false positives. Thus, we can use 1 bit to store the seed. However, false positives may still exist because the newly updated fingerprints may conflict with other hot keys' fingerprints with the seed of 1. Suppose we can maintain full keys of warm KV pairs in an LRU list and feed it to the Cuckoo filter. In that case, we can do many rounds of seeds and fingerprint updates to make the FPR lower until we can totally erase all false positives incurred by these warm keys. The weakness is that we have to spend more memory on storing full warm keys. We still use the fingerprint version in the rest of this paper and demonstrate that the throughput improvement brought by the reduction of FPR can replace caching even with a smaller memory budget. Note that we could write the dynamic-seed Cuckoo filters into a disk with SSTables like filter block in LevelDB because all seed updates will be finished before the construction.

Configure the dynamic-seed Cuckoo filter. The critical parameter in (2,4)-Cuckoo filter we need to configure is the size. The number of cells in a Cuckoo filter influences the effectiveness of membership checking for each SSTable. If the load factor is low, lots of memory space will be wasted; if it is too high, then the reconstruction of Cuckoo filters will frequently happen because some keys cannot find an available cell within the limited kicking-out times, then the compaction of new SSTable will be pulled back in this stage.

The key part of setting the Cuckoo filter's size (number of buckets) depends on how to estimate the number of distinct KV in a new SSTable after compaction. Provided an SSTable size is set to p (e.g. 2MB), and the size of each key-value pair is q on average. We can use $\lceil p/q \rceil$ as an estimation for the number of KV pairs. Since we have other metadata blocks (like footer and indexing blocks) in an SSTable besides data blocks, the number of distinct keys in a full SSTable will not exceed $\lceil p/q \rceil$. Note that we do not have to round our dynamic Cuckoo filter's size to a power of 2 because we use two independent hashing functions to locate two candidate buckets for a key. However, we can initialize the Cuckoo filter with the size of $\lceil p/(q\tau) \rceil$ first, where τ is the expected load factor.

Overheads. Constructing and maintaining our dynamic Cuckoo filters for each valid SSTable will incur additional memory of computational costs. The peak memory overhead will be spent on a set of Cuckoo filters and a list of warm keys' fingerprints. Provided we set 20480 cells for each dynamic Cuckoo filter with 5120 buckets, the fingerprint length is 8 bits, and the seed only costs 1 bit in each bucket. Each dynamic Cuckoo filter costs $5120 \cdot (4 \cdot 8 + 1) bits = 20.62KB$. If all cuckoo filters are put into memory, in a 5-layer LSM-tree with 2 SSTables in L_0 and size ratio r as 8, the whole memory cost for Cuckoo filters will be $20.62KB \cdot \sum_{i=0}^{5-1} 2 \cdot 8^i \approx 193MB$. Additionally, we can apply the common optimization for the last layer L_n, removing all Cuckoo filters there to save memory cost [6].

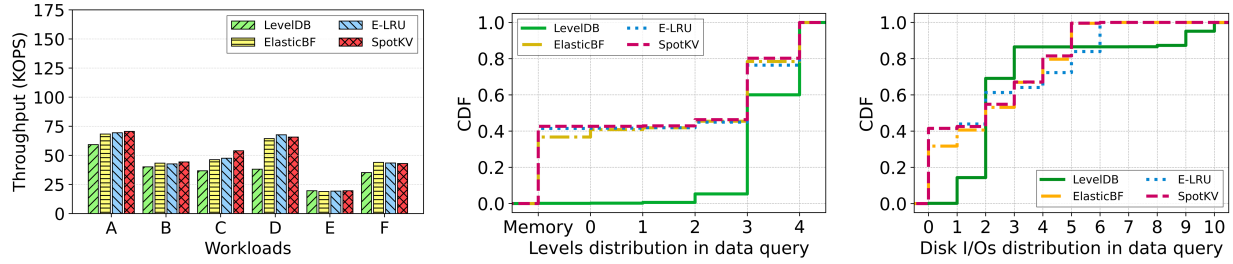
IV. Performance Evaluation

Setup. Our experiments are running on a workstation that maintains the LSM tree key-value store service, with Intel Xeon Silver 4314 CPU @ 2.40GHz, 160GB 2133MHz DDR4 memory, and 48MB LLC. The SSD we equipped is a Samsung 990 PRO NVMe SSD with a user capacity of 1T.

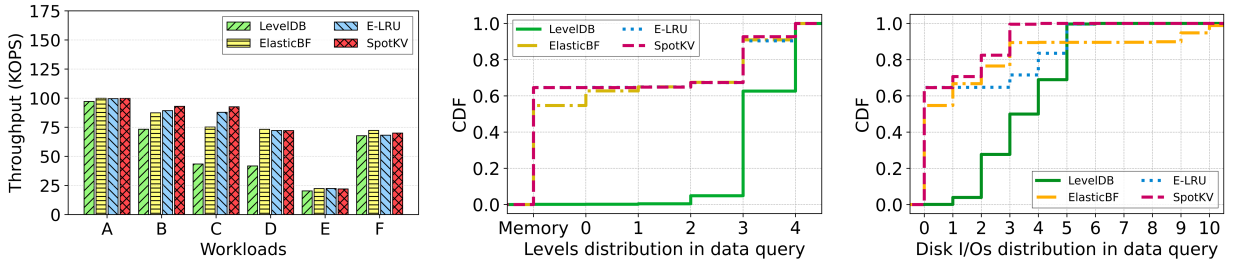
We used the C++ version of Yahoo! Cloud Serving Benchmark (YCSB) [19], [27] as workload by default. We adapt it to generate the KV pairs for SpotKV and run 12 million transactions each time to get the throughput and stat disk I/Os. We use 100M KV pairs (around 12GB) to warm up the database with 20B keys and 100B values, which is a common setting [28] in existing studies. Unless specified, all workloads follow a 0.99-Zipfian accessing pattern. We use workload C of YCSB as the default workload to evaluate and analyze our weighted CM sketch and dynamic Cuckoo filters. We developed SpotKV based on Google's LevelDB v1.20 [1], which is an LSM tree implementation that maintains bloom filter blocks on disk for each SSTable. For a fair comparison, a version of LevelDB that keeps bloom filters in memory [17] is implemented. We configured the SSTable size to 1MB (default setting in LevelDB) and the number of levels to 5, and set the original `block_cache_size` to 100 and `max_open_files` size to 64. Besides LevelDB, we use two other recent works for comparison. (1) E-LRU [10], which assigns an efficiency factor to each cached KV pair. In the implementation we use, the potential number of disk I/Os is assigned as the efficiency factor. When the LRU cache is full, the system checks the 16 least used KV pairs and evicts the one with the smallest efficiency factor. Generally, the higher the level of the KV pair lies, the harder to be evicted from the cache. For cache admission, the system directly inserts the new queried KV pair in the LRU cache. (2) ElasticBF [11], which calculates hotness based on SSTable block's `lifetime`. We maintain a table to record the `lifetime` of each data block in an SSTable. If the block is accessed more than once within the `expiredtime`, we cache all the KV pairs in that block. We use the data operation sequence as time. We set the `expiredtime` as the same as the existence time of the SSTable. If an SSTable is compacted to the next level, the time record for each data block it contains will be removed. We also implement ElasticBF on top of LevelDB due to the lack of open-source code.

A. Overall performance in YCSB

We show the performance of SpotKV under six different YCSB workloads (A, B, C, D, E, F) against E-LRU [10], ElasticBF [11] as well as LevelDB v1.20 [1]. Data read operations make up 50%, 95%, and 100% in workloads A, B, C, respectively. Workload E consists of 100% data scan requests, and there are 50% read-modify-write operations in workload F. Each benchmark performs 12M operations with Zipfian accessing patterns (skewness as 0.99 and 1.1) defined by the benchmark on a pre-loaded key-value data store.



(a) Throughput on YCSB workloads (Zipfian const: 0.99). (b) Level distributions (Zipfian const: 0.99). (c) Disk I/O distribution (Zipfian const: 0.99).



(d) Throughput on YCSB workloads (Zipfian const: 1.1). (e) Level distributions (Zipfian const: 1.1). (f) Disk I/O distribution (Zipfian const: 1.1).

Fig. 6: Evaluation with YCSB workloads.

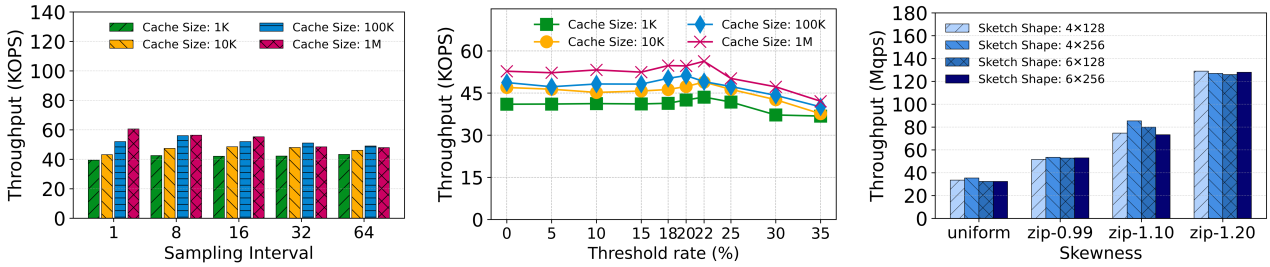
We have configured the cache size as 100K (0.1% size of the total database), and the default threshold for hotness identification in the weighted CM sketch is set at 0.19. The size of the weighted CM sketch is set to 4×256. The resulting throughput data is presented in Fig. 6(a) and Fig. 6(d), corresponding to data access patterns of 0.99 and 1.1, respectively. In Fig. 6(a), SpotKV consistently demonstrates the greatest benefits using an equivalent cache size among various baselines. Note that SpotKV can achieve 1.03-1.93$\times$ the throughput of LevelDB except workload E (100% scan). The throughput performance benefits read-intensive workloads more because our scheme is mainly optimized for data lookup operations.

Then, we give a comprehensive analysis of the impact of the weighted CM sketch on data queries, accompanied by pertinent statistical insights. To this end, we present Figures 6(b) and 6(e) illustrate the CDFs of data lookup operation levels reached under Zipfian access patterns with constants of 0.99 and 1.1, respectively. Remarkably, SpotKV consistently achieves comparable ratios for data queries in memory, whether they stem from the cache or the MemTable. Even with a modest cache capacity of 0.1% entries of the total database, the hit ratios can be mounted to around 40% and 65%, respectively. However, ElasticBF achieves inferior hit ratios when compared with both SpotKV and E-LRU. This difference arises from ElasticBF's reliance on block-level hotness identification, as opposed to the more granular individual KV pair approach used by SpotKV. Also, the `lifetime` updates for each data block maintained by ElasticBF is another reason resulting in low throughput. We implement E-LRU in the granularity of individual KV pairs so that the hit ratio can be similar to the SpotKV. However, the no selective admission of caching KV pairs makes the ratios for cached high-level KV pairs a bit lower than SpotKV does. Compared with LevelDB (in green), the caching-based schemes can make more than 50% of data operations be completed in memory without reaching into the disk. Even if LevelDB may not access any data block in each level due to many reasons (e.g., the queried key does not fall into the key range of all SSTables), Caching-based approaches can make those data operations completed without checking metadata or filter blocks and saving software overheads.

Fig. 6(c) and Fig. 6(f) illustrate the CDF for the number of the potential disk I/Os incurred in 100% data read workloads. We can see the disk I/Os benefits in our SpotKV with the red dotted line that there is nearly no data read operation that needs access to more than three SSTables when the Zipfian constant is 1.1. Further, SpotKV achieves a more left-shifted CDF. This result benefits from SpotKV's priority of caching KV pairs that incur larger disk I/Os, thereby reducing the potential number of disk I/Os. Even if the E-LRU prioritizes the KV pairs lying at a high level, the cache admission rule does not work well for the case where the KV pairs are at a high level and may not be accessed so frequently, but it is still worth catching. Also, the efficiency factor checking causes time latency when deciding which KV pair is supposed to be evicted. Note that LevelDB benefits from the block cache for improving data lookup throughput when the skewness increases from 0.99 to 1.1. From Fig. 6(b) and Fig. 6(e), we can see the level distribution is almost the same but the CDF of disk I/Os are different with the help of the block cache. LevelDB cached more hot keys in the block cache (LRU) in the case of the skewness as 1.1.

B. *Evaluation of weighted CM sketch*

1) Sampling interval.: We leverage lazy sampling to update sketch counters for data queries to reduce the overhead and



(a) Throughput with different sampling rates. (b) Throughput with different hotness thresholds. (c) Throughput with different sketch shape.

Fig. 7: Evaluation of different settings in the weighted CM sketch.

latency incurred by frequently updating the weighted CM sketch. To evaluate its influence, we vary the sampling interval from 1 to 64 and show the performance with four different cache sizes. If the sampling rate equals 8, we update the counters once for every eight key queries. As we can see from Fig. 7(a), the throughput exhibits upward first and downward trends when the cache size is relatively small (e.g., 1K, 10K and 100K). When the sampling interval is large (e.g., 64), the throughput is the lowest because it cannot update the statistical information to cache in time. If we update the weighted CM sketch too often (sampling frequency is high), the extra latency will be incurred. However, if we update the counters with a very low frequency, then our sketch cannot get the fresh statistical results for queried keys. From the results, the read throughput turns out to be the highest when the rate is set to 8 when the cache size is 100K.

2) Threshold.: The hotness threshold is an important parameter for weighted CM sketch to identify hot keys. In our experiments, we evaluate the impact of the threshold on throughput by adjusting its value within the range of 0 to 0.35 used for cache admission. As depicted in Fig. 7(b), the throughput initially experiences an increase with the incremental threshold value, followed by a subsequent decline. We can observe this trend in all different cache size settings. With a threshold set to 0, the weighted CM sketch totally functions as an LRU cache. In this configuration, each newly queried KV pair enters the cache immediately right after being queried. As the threshold value increases, the cache admits more frequently queried KV pairs. Additionally, those pairs lying at higher levels are prioritized for getting into the cache. Thus, the throughput climbs to its peak. However, with a continuous increase of the threshold, a KV pair has to be very hot to be chosen, which cannot fully use the cache's capacity. Consequently, the throughput experiences a downturn.

3) Sketch shape.: In our evaluation, we also conducted an analysis of the influence of the sketch size on its performance. To assess the sketch's capabilities, we explored four distinct shapes with 4×128, 4×256, 6×128, and 6×256, where 6 corresponds to the number of hashing computations (rows) within the sketch, and 256 denotes its width. as depicted in Fig. 7(c) To provide a comprehensive view of its performance characteristics, we subjected the sketch to varying data query distributions, ranging from uniform to Zipfian with a constant parameter of 1.2 and the throughput remains stable. The shapes with more rows (e.g., 6×128, and 6×256) are supposed to incur extra latency. However, the hashing computation costs less overheads with *hash chaining*, where the new hashing result can be generated from the last hash result with fewer computations. Further, the sketch with a large width can lower the estimated errors in hotness identification. Some keys may be selected and cached because the hash collision happened in the sketch instead of its own actual hotness because other KV pairs also contribute to the estimation value toward its hotness counters. In the evaluation, the shape (4×256) can win with a slight advantage compared with other combinations among the listed four configurations.

C. Impact of dynamic-seed Cuckoo filter

This section evaluates the dynamic-seed Cuckoo filters with different SSTable sizes and fingerprint lengths. We can see from Fig. 8, the throughput of SpotKV increases as we use more bits as fingerprints in dynamic Cuckoo filters. In this evaluation, there are 16K KV pairs in each SSTable. The memory cost for each dynamic Cuckoo filter is 12.5KB when the fingerprint length is 6 bits, and a total space of 78.35MB is used for all SSTables with zero false positives for hot keys. However, the bloom filters used in LevelDB set `bits_per_key` as 10 bits, and there will be 100MB for 1M KV pairs.

D. Impact of workload settings.

Impact of different workload skewness. In this section, we evaluate both SpotKV and LevelDB with 100% read workload in the YCSB benchmark. With 100M KV pairs warmed up, we varied the query skewness from uniform to Zipfian-1.2 in the following 10M read operations. As shown in Fig. 9(a), when the skewness rises, both SpotKV and LevelDB experience an increase in read throughput. LevelDB benefits from the block and table cache to reduce the time for accessing queried KV pairs. SpotKV and LevelDB share the same throughput of around 37.5 KOPS when the workload is uniform. However, when the skewness degree equals 1.2, SpotKV can reach 139.7 KOPS, which is almost three times that of LevelDB.

Impact of different read/write compositions. In this section, we evaluate the performance of SpotKV and LevelDB with different read/write compositions in the YCSB workload. We varied the makeup of read operations in workload from 20% to 100% with Zipfian-0.99, as shown in Fig. 9(b).

When the read operations makeups are small (e.g., 20% to 40%), SpotKV achieves a throughput similar to LevelDB.

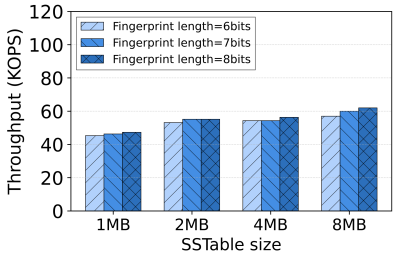
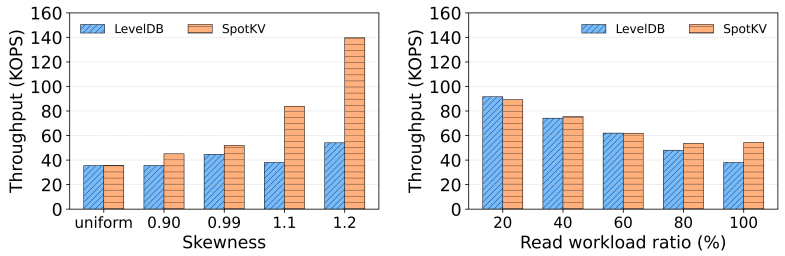


Fig. 8: Throughput with different fingerprint lengths.



(a) Throughput with different skewness in workload.

(b) Throughput with different Read/write ratio.

Fig. 9: Evaluation of the different workload settings.

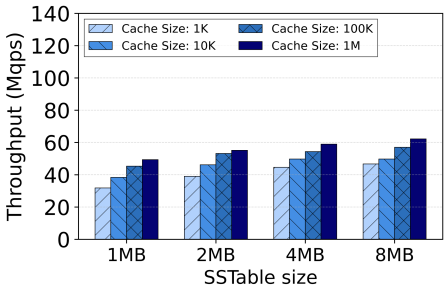


Fig. 10: Evaluation of the different SSTable sizes.

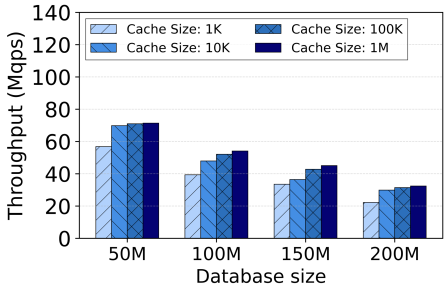


Fig. 11: Evaluation of the different database sizes.

This is because SpotKV is mainly optimized for accelerating data read operations. Further, the dynamic-seed Cuckoo filter construction will incur latency in changing seeds for potential false positives. As the data read workload makeup increases, the cache maintained in SpotKV can improve the whole throughput, especially when there are more than 80% read operations.

E. Impact of varied SSTable sizes.

Fig. **??** delves into the effects of varying SSTable sizes on read throughput. We observe that the read throughput reaches its lowest point when the SSTable size is 1MB. This is because small SSTables lead to the generation of more SSTables in the database with the same number of KV pairs, triggering more compactions that can stall read operations and ultimately reduce read throughput. As the SSTable size increases, the data lookup throughput improves from 30 KOPS to 45 KOPS when the cache size is 1K.

V. Related Work

KV pair hotness identification. TRIAD [16] separates hot keys and cold keys with the statistical function of `getTopKHot` recording the frequency of each key in the current Immutable MemTable. L2SM [17], [29] proposes HotMap, which is like a stack of multiple Bloom filters of each SSTables, and one key's hotness value could be obtained from the number of positive results of all bloom filters. The more positive results bloom filters indicate, the hotter the key would be. Even if L2SM does think of the IO difference caused by the key's position in the tree, the calculation of summing all filters' results up is a time-consuming process for hotness identification. Our SpotKV flexibly combines the Count-Min sketch and layer difference to achieve a quick hotness keys separation scheme.

Effective query-agnostic filters. Chucky [30] constructs a large Cuckoo Filter in memory to locate every single key's SSTable. The updating cost incurred by the associated change would drag the lookup throughput. Monkey [12] exhibits an optimal balance between the costs of updates and lookups with a specific memory budget. It models the worst costs on point lookup and update operation with different size ratios in the LSM trees and shows that we can assign the different number of bits per key to Bloom filters in different layers to achieve a lower false positive under the same memory budget. ElasticBF [11] manages all KV pairs in the unit of the data segment and assigns different bits per key parameter for bloom filters in different hotness segments. In a nutshell, Monkey [12] compromises more space to bloom filters in shallow layers, while ElasticBF [11] grants a larger memory usage for identified hot key-value segments' Bloom filters. However, both of them are trying to give more bits for hot keys in Bloom filters, and our SpotKV further aims to lower the false positive rate.

VI. Conclusion

In this paper, we design and implement an LSM tree store called SpotKV with a novel caching scheme, aiming to set a disk I/O-aware cache admission rule by incorporating a weighted CM sketch and dynamic-seed Cuckoo filters. The core idea of this scheme is to assign varying priorities to queried KV pairs across different SSTable layers. We also develop mechanisms to mitigate false positives arising from the membership query structure by changing dynamic seeds. SpotKV has been implemented on top of LevelDB, and the extensive experiments reveal its ability to deliver higher throughput compared to recent works.

ACKNOWLEDGMENT

We sincerely thank our three anonymous reviewers for their insightful suggestions. The authors were partially supported by NSF Grants 1750704, 2114113, and 2322919, and DoE Grant DE-SC0022069.

REFERENCES

[1] "Leveldb. https://github.com/google/leveldb."
[2] "Rocksdb. https://github.com/facebook/rocksdb."
[3] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb." in *CIDR*, vol. 3, 2017, p. 3.
[4] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.
[5] "https://docs.ceph.com/en/quincy/rados/configuration/bluestore-config-ref/."
[6] Y. Matsunobu, S. Dong, and H. Lee, "Myrocks: Lsm-tree database storage engine serving facebook's social graph," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3217–3230, 2020.
[7] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang *et al.*, "Tidb: a raft-based htap database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3072–3084, 2020.
[8] Q. Zhang, Y. Li, P. P. Lee, Y. Xu, Q. Cui, and L. Tang, "Unikv: Toward high-performance and scalable KV storage in mixed workloads via unified indexing," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 2020, pp. 313–324.
[9] H. H. Chan, C.-J. M. Liang, Y. Li, W. He, P. P. Lee, L. Zhu, Y. Dong, Y. Xu, Y. Xu, J. Jiang *et al.*, "HashKV: Enabling efficient updates in KV storage via hashing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 1007–1019.
[10] F. Wu, M.-H. Yang, B. Zhang, and D. H. Du, "AC-Key: Adaptive caching for LSM-based Key-Value stores," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 603–615. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/wu-fenggang
[11] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "ElasticBF: Elastic bloom filter with hotness awareness for boosting read performance in large Key-Value stores," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 739–752.
[12] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable Key-Value store," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 79–94.
[13] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing Key-Value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 121–136.
[14] Y. Li, Z. Liu, P. P. Lee, J. Wu, Y. Xu, Y. Wu, L. Tang, Q. Liu, and Q. Cui, "Differentiated Key-Value storage management for balanced I/O performance," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 673–687.
[15] M. Mitzenmacher, S. Pontarelli, and P. Reviriego, "Adaptive cuckoo filters," 2020.
[16] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: Creating synergies between memory, disk and log in log structured Key-Value stores," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 363–375.
[17] K. Huang, Z. Jia, Z. Shen, Z. Shao, and F. Chen, "Less is more: De-amplifying i/os for Key-value stores with a Log-assisted LSM-tree," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 612–623.
[18] "https://tikv.org/deep-dive/key-value-engine/b-tree-vs-lsm."
[19] J. Ren, "Ycsb-c. https://github.com/basicthinker/ycsb-c." 2016.
[20] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," in *LATIN 2004: Theoretical Informatics: 6th Latin American Symposium, Buenos Aires, Argentina, April 5-8, 2004. Proceedings 6*. Springer, 2004, pp. 29–38.
[21] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang, "Cocosketch: High-performance sketch-based measurement over arbitrary partial key query," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 207–222.
[22] D. Yao, H. Wang, H. Xu, and M. Zhang, "Lightweight per-flow traffic measurement using improved lru list," *IEEE Transactions on Network Science and Engineering*, 2023.
[23] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 741–756.
[24] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, 2014, pp. 75–88.
[25] S. Shi, C. Qian, and M. Wang, "Re-designing compact-structure based forwarding for programmable networks," in *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. IEEE, 2019, pp. 1–11.
[26] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
[27] G. Xanthakis, G. Saloustros, N. Batsaras, A. Papagiannis, and A. Bilas, "Parallax: Hybrid Key-Value placement in LSM-based Key-Value stores," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 305–318.
[28] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Kroth, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "From WiscKey to bourbon: A learned index for Log-Structured merge trees," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 155–171.
[29] K. Wang and F. Chen, "Catalyst: Optimizing cache management for large in-memory key-value systems," *Proceedings of the VLDB Endowment*, vol. 16, no. 13, pp. 4339–4352, 2023.
[30] N. Dayan and M. Twitto, "Chucky: A succinct cuckoo filter for LSM-Tree," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 365–378.