Optimizing Read Performance of HBase through Dynamic Control of Data Block Sizes and KVCache

Sangeun Chae* Sungkyunkwan University
Wonbae Kim Kakao Corp.
Daegyu Han Sungkyunkwan University
Jeongmin Kim Naver Corp.
Beomseok Nam* Sungkyunkwan University

ABSTRACT

LSM-Tree-based key-value stores such as HBase, RocksDB, and Cassandra use a fixed data block size. In this study, we show that using a fixed block size can lead to unnecessary read amplification and cache pollution. To address this issue, we propose a dynamic data block size control method to store small key-values in small data blocks and large key-values in large data blocks to minimize disk I/Os. However, using small data blocks for small key-values can result in performance issues due to increased disk seeks. To mitigate this problem, we implement a two-level cache system, which involves a lower level conventional BlockCache for storing larger, coarse-grained data blocks and an upper level cache, *KVCache*, for storing smaller, fine-grained key-value pairs. Our experiments show that the dynamic data block size control and fine-grained KVCache help effectively reduce read amplification and improve read performance in HBase.

CCS CONCEPTS

• **Information systems** → **Key-value stores**; • **Software and its engineering** → *Operating systems.*

KEYWORDS

Key-value stores, Log-structured merge tree

ACM Reference Format:
Sangeun Chae, Wonbae Kim, Daegyu Han, Jeongmin Kim, and Beomseok Nam. 2024. Optimizing Read Performance of HBase through Dynamic Control of Data Block Sizes and KVCache. In *Proceedings of ACM SAC Conference (SAC'24).* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3605098.3635898

1 INTRODUCTION

Apache HBase [12] is a distributed data store used to store large amounts of unstructured data. It is widely used by web service companies such as Yahoo!, Facebook, Twitter, and Naver, who operate large-scale data centers [2]. HBase's data storage engine is based on the LSM tree, which is implemented on top of HDFS and has a data model similar to Google's Bigtable [8, 10, 12, 17, 19, 28, 32]. LSM-tree has performance characteristics that make it attractive for write-intensive workloads because it buffers writes using an in-memory index called MemStore (a.k.a MemTable in RocksDB and LevelDB), and flushes them to disk in batches to take advantage of the disk's high sequential write bandwidth.

However, LSM tree is known to suffer from write amplification problem due to the repeated merge-sort operation called *compaction*. This compaction process involves flushing in-memory indexes to disk in the form of sorted key-value arrays known as HFiles (also known as SSTables in other KVStores), which are merged incrementally with other previously sorted arrays in batches. This background process requires the same key-value data to be written to disk multiple times, causing *write amplification* problems.

In the past decade, numerous studies have proposed various solutions to mitigate this write amplification problem [3, 7, 13, 16, 21, 23, 25, 26, 30, 33]. However, many previous studies have sacrificed read performance to mitigate the write amplification problems. For example, Fragmented LSM Tree [30] proposed to reduce the number of merge-sort operations by allowing key ranges to overlap between SSTables.

However, improving the read performance of LSM Tree has not received much attention in the community, despite LSM-Tree being an index that inherently sacrifices read performance to improve write performance. The LSM tree has a single sequence of sorted keys at each level, except for level 0 where the key ranges are allowed to overlap. Therefore, multiple sorted arrays need to be searched if data is not found at the first level, and read operations frequently access sorted arrays stored on the disk, resulting in the read amplification problem.

To avoid accessing multiple levels and thereby reduce the read amplification, several recent research works aimed to mitigate the read amplification problem by improving Bloom filters [31]. There also exist some other works that propose to modify the compaction strategies [14], to partition key space [18, 22], to use adaptive indexes [22], and to predict the location of data using machine learning techniques [9, 22].

Our study, which uses real workloads from Naver [1], a leading web portal site in Korea, finds that one of the primary reasons for the read amplification problem in HBase is that disk accesses are performed in large data blocks (64 KB) instead of in smaller units of pages (4 KB) like B-trees. In other words, if an application requests data that is smaller than 4 KB in size, and the data block size in HBase is 64KB, the read amplification factor is 16, even if the query reads only one data block from one HFile. Our study finds read amplification caused by accessing multiple HFiles due to key range overlaps is not significant because Bloom filters in HBase effectively filter out unnecessary file accesses. But we discover that due to the mismatch in size between data blocks and individual key-value pairs, resulting in an increase in read amplification.

*Department of Computer Science and Engineering

SAC'24, April 8 –April 12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04…$15.00
https://doi.org/10.1145/3605098.3635898

The granularity mismatch problem occurs not only when accessing disks, but also when caching data blocks. Most key-value stores utilize large in-memory cache to improve read performance. For example, HBase and RocksDB cache data blocks in an in-memory cache called *BlockCache*. In general, caching is effective in improving read performance by reducing the number of disk accesses, but caching in the LSM tree is not similar to the file system's page caching in that key-value data can vary in size [34]. Also, the degree to which caching improves performance is dependendent on the level at which the cached data is located in the LSM tree.

For the workloads that have temporal locality, i.e., the most recently inserted records are the most popular, LSM tree-based key-value stores rely on small in-memory write buffers called MemStores or MemTables. However, in-memory write buffers represent only a tiny fraction of the overall data stored in key-value stores. As such, read operations frequently access sorted arrays stored on the disk, and the read amplification problem occurs. To reduce the read amplification, efficient caching techniques are necessary for queries that have relatively low temporal locality.

HBase and RocksDB, along with other key-value stores, organize the sorted sequence of key-value pairs into data blocks of fixed size. Each data block is encoded for variable length records and often compressed to reduce the size. Therefore, when reading a single key-value pair, the entire data block needs to be read and decoded. Then, the data block is cached in the block cache. If all key-value pairs in multiple disk pages of the same data block are cached together, it would be ideal if subsequent queries read adjacent key-value pairs from those cached pages. However, in the Naver workload, we observe that there are not many queries such as range queries that access those adjacent key-value pairs. As a result, reading large data blocks leads to cache pollution and degrades cache utilization. Again, this is because queries with high spatio-temporal locality are primarily absorbed by MemStores (MemTables).

In this study, we propose dynamically adjusting the size of data blocks based on the size of the key-values being written. By writing small key-values in small blocks and large key-values in large blocks, we can substantially reduce unnecessary I/Os and improve read performance. Using variable-sized data blocks also helps improve the space efficiency of the BlockCache by preventing large cold data from taking up cache space meant for small hot data. This non-intrusive design is beneficial for improving the performance of queries with spatial locality, such as range queries, by allowing the existing BlockCache to be used as is. However, this design can lead to large blocks containing large key-values occupying a significant amount of cache space, resulting in low cache efficiency if those large key-values are cold data.

To address this issue, we implement *KVCache* in HBase that caches hot key-value pairs on top of the BlockCache. By caching individual hot key-values in KVcache, frequently accessed hot key-values can be separated from cold key-values. This separation allows blocks containing cold key-values to be evicted from the BlockCache, increasing cache utilization. To facilitate this, we propose a novel cache promotion policy based on the relative access frequency and data size of each individual key-value.

The rest of this paper is organized as follows: In Section 2, we provide background and motivation. In Section 3, we detail our dynamic data block size control mechanism. In Section 4, we describe KVCache and its promotion and demotion policy. In Section 5, we evaluate the performance of the proposed methods. In Section 6, we discuss related work. Finally, in Section 7, we conclude the paper.

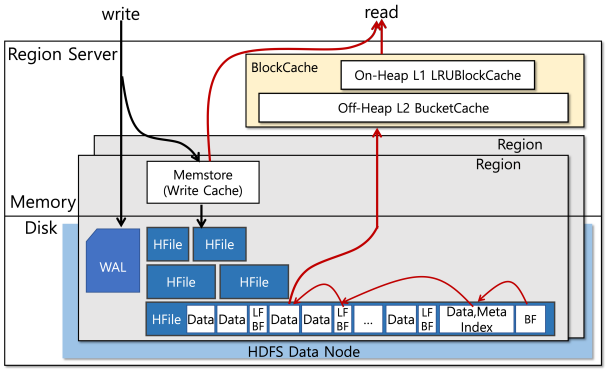


Figure 1: IO Path in HBase RegionServer

2 BACKGROUND AND MOTIVATION

2.1 Apache HBase

HBase is a distributed column-oriented data store that runs on top of HDFS. HBase consists of three major components - *HMaster*, which is the front-end master node, *RegionServers*, which are backend worker nodes, and *Zookeeper*, which helps to coordinate all the distributed nodes in the system.

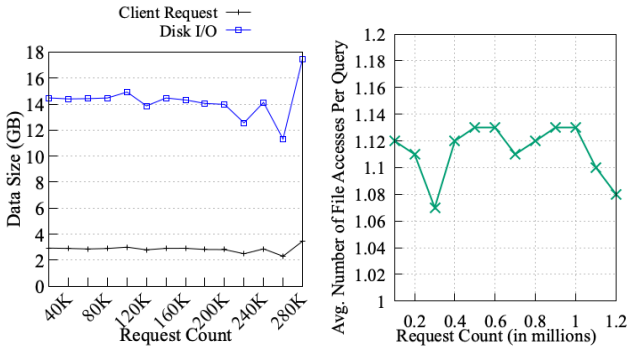
In HBase, keys are partitioned and each partition is assigned to a backend RegionServer. When a key-value pair is inserted into a RegionServer, it is first written to a log file called *HLog* and then buffered in the RegionServer's *MemStore*, which is an in-memory SkipList, as shown in Figure 1. Once the MemStore becomes full, it is flushed to HDFS as a new *HFile*. HFiles contains sorted key-value pairs that are stored in *data blocks*. The first key of every data block is indexed by a B+tree that is stored in *index blocks*, which helps to efficiently navigate through each HFile.

Even though index blocks are employed to find a particular data block in an HFile, it's possible for the key range of one HFile to overlap with those of other HFiles. Therefore, to ensure that queries are performed efficiently without accessing too many HFiles, HBase constructs Bloom Filters for each data block and each HFile. In addition, if the number of HFiles exceeds a certain threshold, the RegionServer triggers a background process called compaction, which performs merge-sort to create a new large HFile and reduce the number of overlapping HFiiles. This helps to avoid searching through too many HFiles when performing a read query.

2.2 Blocks in HBase

HBase, a distributed database, performs I/O with HFiles in blocks like data, index, and Bloom blocks, each severing specific functions. Data blocks store key-values and are generated during MemStore flushes, while index and Bloom blocks improves access and efficiency, optimizing cache and performance.

HBase boosts performance with a dedicated BlockCache for each RegionServer, storing frequently accessed blocks efficiently in memory. There are three main BlockCache options: LruBlockCache, SlabCache, and BucketCache. LruBlockCache, the default, resides in



(a) Read Amplification (b) Number of Accessed HFiles per Query

Figure 2: Workloads in Naver Data Center

the JVM heap and divides memory into *single-access*, *multi-access*, and *in-memory* sections, typically at 25%, 50%, and 25% allocations. LruBlockCache can also serve as a first-level L1 cache in multi-level setups with SlabCache or BucketCache.

SlabCache and BucketCache offer alternative caching approaches in HBase. SlabCache optimizes cache space, dividing it into zones for small and large blocks. It reserves 80% of the total off-heap cache size for small blocks and 20% for large ones, excluding oversized blocks. BucketCache splits cache space into 14 buckets, each with single-access, multi-access, and in-memory areas. It employs an LRU algorithm like LruBlockCache and serves as a second-level L2 cache in multi-level setups. These caching strategies enhance HBase's data access capabilities, optimizing performance in big data and real-time processing contexts.

2.3 Read Amplification

Figure 2(a) shows the total number of accessed disk pages and the number of disk pages consumed by 20,000 queries per RegionServer over time in the Naver data center's HBase cluster. The HBase cluster consists of 3 Master nodes, 10 Storm supervisor nodes (ver. 2.2.1), 5 Kafka Broker nodes (ver. 2.3.1), and 16 HBase RegionServers (ver. 2.2.7) and HDFS DataNodes (ver. 3.1.4). In the Master nodes, HBase Master, ZooKeeper (ver. 3.4.14), Yarn Resource Manager (ver. 3.1.4), Storm Nimbus, and Ambari (ver. 2.2.1) are deployed. Each RegionServer has dual E5-2630 v3 CPUs (8 cores, 2.4GHz), 128 GB DRAM, and 12 4 TB 7200 RPM HDDs with RAID 1 configuration.

While applications are consuming an average of 150 KB of data per query, HBase reads an average of 700 KB of pages from disk. Even though about 30% queries benefit from cache hits and they can avoid disk IO with the 2 GB BlockCache, the average disk IO is over 4× the size of the data requested by the application, confirming that the read amplification is very severe in HBase.

There are two main reasons why read amplification occurs in HBase. First, as in all LSM tree-based key-value stores, HBase requires reading multiple HFiles to process a single query, as HFiles' key ranges can overlap. We will refer to this problem as *multi-file access amplification*.

Second, HBase's disk IO unit is not a disk page but a data block, so if the size of a data block is set larger than 4 KB, even if an application requests data smaller than 4 KB, the entire data block is read and stored in the BlockCache, resulting in read amplification. We will refer to the read amplification that occurs when the size of the data block and the size of the data are different as *granularity-mismatch amplification*.

2.3.1 Multi-File Access Amplification. Ideally, in HBase, it's best to manage only one HFile per column family to minimize disk access during reads. However, if the workload is write-intensive, more HFiles are flushed to disk, which makes read queries access large numbers of HFiles due to overlapping key ranges. This overlapping key range problem, which we refer to as *multi-file access amplification*, is a fundamental issue with LSM trees. To prevent reading unnecessary HFiles, HBase performs minor and major compactions. Minor compaction merges a configurable number of smaller HFiles into one larger HFile, while major compaction merge-sorts all HFiles in a RegionServer to create a single, large HFile. Since major compaction is periodically performed in HBase clusters, it's common for RegionServers to have one very large HFile of several terabytes for each column family, and several recently created small HFiles.

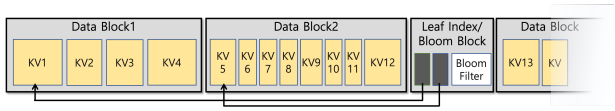
Figure 2(b) shows the average number of HFile accesses per query in the Naver's HBase cluster. Despite having about 340 HFiles per region, recently created HFiles are small, and thus Bloom Filters very effectively prevent unnecessary file access to those small HFiles using the default 131KB Bloom Blocks. With the default configuration, the total size of Bloom Blocks for 1.5 TB HFiles, i.e., a column family, is only about 200 MB. Through this experiment, it was found that small Bloom Filters are effective in reducing read amplification due to multiple file accesses on a real production cluster, even when minor compaction is performed infrequently.

2.3.2 Granularity-Mismatch Amplification. HBase uses fixed-size data blocks per column family. The default size of a data block is 64 KB, which is the smallest IO unit HBase can use to access HFiles. Therefore, even if the requested key-value size by the application is smaller than the data block size, a 64 KB data block is read, i.e., unnecessary disk I/O occurs. To solve this granularity-mismatch issue, a simple and naive solution is to decrease the data block size. However, using smaller, fixed-size data blocks can negatively impact performance.

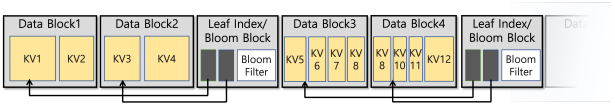
If small data blocks are used, random reads occur more often, which increases the number of disk seeks and degrades read performance. In contrast, if large data blocks are used, a large number of unrequested key-values are cached in BlockCache. If cache hits occur when neighboring key-values in the same data block are requested by subsequent queries, caching large data blocks can potentially reduce the number of disk I/Os. However, if there's no spatio-temporal locality or if the cache size is smaller than the dataset, even if a large data block is used, random reads still occur.

Additionally, the data block size determines the number of data blocks and index blocks. Consequently, utilizing small data blocks may result in an increase in index size, leading to the inability to cache all index blocks in the block cache.

The HFile layouts in Figure 3 provide two examples - one with a large data block size and the other with a small data block size. When an HFile contains numerous small data blocks, the B+tree height increases to three, and leaf node index blocks are interspersed between the data blocks. In this example, each leaf node can index

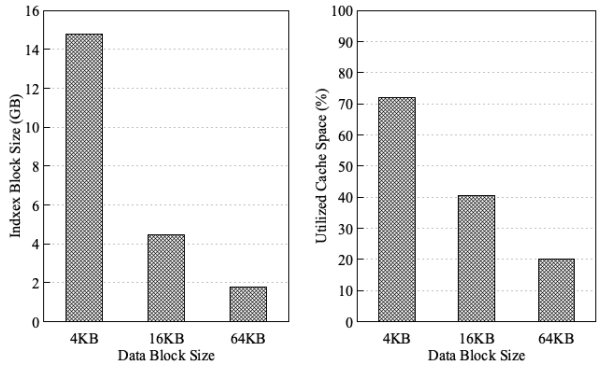


(a) Static Large Block



(b) Static Small Block

Figure 3: Data Blocks and Index Blocks in HFiles



(a) Total Size of Index Blocks with Varying Data Block Size (1.5 TB)

(b) Cache Pollution with Varying Data Block Size

Figure 4: Trade-offs in Small and Large Data Blocks

two data blocks, resulting in a leaf index block and its Bloom filter block being saved after every two data blocks.

Figure 4(a) illustrates the total size of index blocks for HFiles of 1.5 TB with varying data block sizes. When the data block size is configured to 4 KB, the total size of index blocks increases to 14.8 GB, which is 8.3× larger than the index block size when the default 64 KB data blocks per RegionServer are used. To enhance indexing performance, index blocks must be small enough to be cached in memory. If index blocks are not in memory, they must be retrieved from disk, causing a substantial increase in disk I/O because index blocks are accessed more frequently than data blocks.

When the total size of index blocks increases, the likelihood of index block cache misses rises. As a result, using a 4 KB data block size may lead to further problems of read amplification and random disk access. Hence, adjusting the size of the data block to 4 KB is not a viable solution.

2.3.3 Granularity Mismatch and Cache Pollution. HBase employs the LRU replacement policy for both LRUBlockCache and BucketCache on a per-block basis. However, frequent access to a data block does not guarantee that all key-values in the block are frequently accessed. When only specific key-values are frequently accessed, adjacent key-values may remain in the BlockCache even if they are not used. In the experiments shown in Figure 4(b), we measured the percentage of key-values that were cached in the multi-access area of BlockCache but never accessed. As the data block size increases, the number of key-values that unnecessarily occupy the cache space due to neighboring hot key-values increases, leading to decreased cache utilization, as low as 20%.

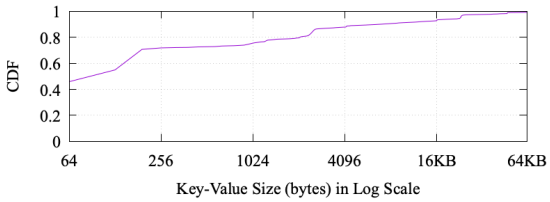


Figure 5: Data Block Size Distribution in Naver's HBase Cluster

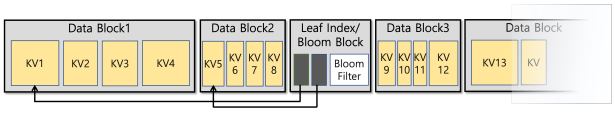


Figure 6: Dynamic Data Block Size

Algorithm 1 *BlockWrite(position, MinBlockSize, MaxNumData)*

```
1: WrittenSize, EntryCount = 0
2: while true do
3:     KeyValue.append(position)
4:     WrittenSize+ = KeyValue.size()
5:     EntryCount+ = 1
6:     if WrittenSize > DataBlockSize then
7:         break
8:     end if
9:     if WrittenSize > MinDataBlockSize AND EntryCount > MaxNumData then
10:        break
11:    end if
12:    position+ = KeyValue.size()
13: end while
```

3 DYNAMIC BLOCK SIZE CONTROL

When determining the optimal data block size, it is important to consider the performance trade-offs that we presented in Section 2. To strike a balance between the advantages and disadvantages of large and small data blocks, we have developed a *dynamic block size control* scheme for HBase.

The underlying principle of this scheme is to store small key-values in small data blocks and large key-values in large data blocks. This approach helps to reduce read amplification for small data blocks while large data blocks benefit from higher sequential read throughput and a smaller index size, resulting in fewer disk accesses.

The distribution of key-value sizes stored in Naver's HBase cluster is shown in Figure 5. This distribution shows that the cluster stores key-values of various sizes, with over 80% of key-values smaller than 4 KB and less than 10% of key-values exceeding 16 KB. If a fixed data block size is used for an HBase cluster with such a wide variety of key-value sizes, significant read amplification or IO performance degradation due to random reads may occur. To address this issue, we propose a *dynamic block size control* scheme that stores small key-value data in small blocks and large key-value data in large blocks. In other words, the data block size is dynamically configured based on the size and number of key-values to be written to each data block.

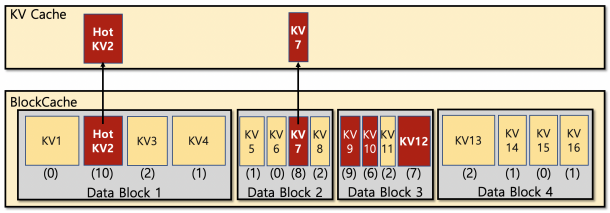


Figure 7: Promoting to KV Cache

In vanilla HBase, key-values are appended to a data block in sorted order, as HFile is written sequentially. The size of the current data block is checked against the default block size (`DataBlockSize`) to determine when to close the data block. Specifically, if the current data block size reaches or exceeds the default block size, the current data block is closed. Then, its corresponding Bloom filter and leaf node of the multi-level index are constructed and appended to the file. Afterward, the next data block is opened, and the process of writing key-values continues. Finally, when all data blocks have been written, the upper-level tree nodes and global per-file Bloom filter are written to the end of the file, along with other miscellaneous metadata.

Algorithm 1 presents the dynamic block size control algorithm that enhances the default behavior of vanilla HBase. This algorithm adds two more conditions to vanilla HBase's default conditions for closing data blocks. The first condition is the `MaxNumData` parameter, which limits the number of key-value data that can be stored in a single data block. If the number of key-value data in the current block exceeds this parameter, the data block is closed (Line 10). This condition mitigates read amplification problems by preventing too many small key-values from being stored in a large data block. The second condition is the `MinDataBlockSize` parameter, which prevents the data blocks from becoming too small. If the number of key-values in the current data block exceeds `MaxNumData`, but the size of the data block is smaller than `MinDataBlockSize`, the data block is not closed, and more key-values are added to increase the block size. The rationale for using this second parameter is to avoid creating too small data blocks, which can lead to frequent random IOs and an excessively large index size.

These two parameters ensure that the data block size remains within the `MinDataBlockSize` and `MaxDataBlockSize` ranges and that each data block size is determined by the size of key-values stored in it. Furthermore, this algorithm enables dynamic control of the block size by adding two simple conditional statements, thereby avoiding any additional overhead in the write process.

4 KEY-VALUE CACHE

The efficiency of the cache is essential in reducing the read amplification that results from the granularity-mismatch problem. However, in HBase, the block-level LRU replacement policy leads to unused key-values occupying valuable cache space, as discussed in Section 2.3.3. Although the dynamic block size control scheme can reduce the number of unnecessarily cached data, further improvements in space utilization can be achieved by modifying the cache management policy.

To address this issue, we introduce the *KVCache* as a level 1 cache that stores data in key-value units. The KVCache complements the legacy level 2 BucketCache, which caches data in blocks, resulting in a more efficient cache management system.

4.1 Promotion to KVCache

In the multi-access area of BucketCache, we select data blocks with low cache space utilization to promote hot key-values that are frequently accessed. If a data block contains a small number of hot key-values that are accessed more frequently than other key-values in the same block, those hot key-values are promoted to KVCache.

To identify data blocks with low space utilization, an array called `AccessCount[]` is created when a new data block is cached in BucketCache. The numbers in parentheses in Figure 7 represent the number of accesses to each key-value data, which are the elements of `AccessCount[]`. Whenever a key-value data is accessed, the corresponding element in `AccessCount[]` is incremented by 1. Then, we calculate the mean and standard deviation of the array to identify outliers. If the number of accesses to a particular key-value is greater than a certain threshold and also greater than the sum of the mean and standard deviation, it is determined that the frequency of access to key-value data in the data block is not balanced. Consequently, the key-value data with high access frequency is promoted to KVCache.

In the example shown in Figure 7, `KV2` and `KV7` are promoted to KVCache due to low cache space utilization. However, `KV9`, `KV10`, and `KV12` stored in `Data Block 3` are not promoted to KVCache because their data block has high cache space utilization, indicating that they are evenly accessed. Although the number of accesses to `KV13` in `Data Block 4` is greater than the sum of the mean and standard deviation, it is not promoted to KVCache because its number of accesses is not large enough. That is, the key-values in `Data Block 4` have not been accessed enough times to be promoted to KVCache.

When a key-value is promoted to KVCache, its access count is reset to 0, and the average access count of the data block is recalculated without including the access count of the promoted key-value. By initializing the access count of the promoted key-value to 0, other key-values in the same data block have a chance to be promoted to KVCache.

The data block containing the promoted key-value is automatically added to the eviction candidate list and replaced if it is not accessed subsequently. However, if any key-value in the same data block is accessed while on the eviction candidate list, it is removed from the list and retained in the BucketCache. If another key-value of the same block is accessed on the eviction list, it's removed and stored in the BucketCache.

BucketCache manages its cache space in data block units. Therefore, reading a key-value requires navigating index blocks, finding a data block, decompressing it, parsing it, and doing a binary search within the sorted array to find the key-value. In contrast, KVCache manages key-values using HashMap indexes, allowing for fast exact match queries.

However, when a range query scans key-values in key order, it increments the number of accesses to all key-values in the data block equally. Therefore, key-values frequently accessed by range queries

are better cached in BucketCache rather than KVCache, while key-values frequently accessed by point queries are better cached in KVCache. Thus, depending on the workload characteristics, it is necessary to determine which cache to use to store key-values and dynamically adjust the memory size allocated to each cache.

To dynamically adjust the cache sizes, we allocate a small 10 MB of memory to KVCache when a RegionServer starts. As more key-values are promoted to KVCache, we increase its size by taking memory from BucketCache. To do this, we select a set of victim buckets from the eviction list and deallocate them, freeing up space to allocate to KVCache. If there are no available victim buckets in the eviction list, we remove data blocks with the smallest number of accesses from the multi-access area of BucketCache. This frees up space that can be used to store more frequently accessed data blocks, and allows KVCache to continue growing.

4.2 KVCache Replacement Policy

The traditional LRU replacement policy falls short of meeting the needs of KVCache in several ways. First, KVCache stores key-value pairs of varying sizes (*DataSize*), making it challenging to optimize replacement decisions. Second, due to the promotion policy, a key-value's last access time (*LastAccessTime*) alone cannot accurately indicate its hotness. That is, a key-value may have been accessed multiple times before being promoted to KVCache, and the promotion decision is also influenced by the access frequency of neighboring key-values in the same data block. Therefore, to make effective replacement decisions, KVCache must consider three parameters: the key-value's last access time (*LastAccessTime*), its access frequency (*AccessFrequency*), and its data size (*DataSize*).

Numerous replacement policies can be designed using these three parameters. Interestingly, studies from over two decades ago on Web proxy cache replacement have explored similar challenges [4, 6, 27, 29]. Unlike traditional page caches in operating systems, Web proxy servers retrieve and cache Web documents of varying sizes from remote servers with different latencies. KVCache's replacement policy faces similar challenges, and can benefit from the insights gained from these earlier studies.

The formula used for KVCache in this study is as follows, and this formula is a variation of the *MIX* policy [27].

$$W = \frac{NormalizedAccessFrequency^{f}}{DataSize^{d} \cdot (CurrentTime - LastAccessTime)}$$

The exponents f and d are tuning parameters that determine the relative importance of *DataSize* and *AccessFrequency*, and they need to be tuned according to the workload characteristics. However, in our study, we observed the cache hit ratio is not too sensitive to the two parameters, so both parameters are simply set to 1.

Based on the formula given above, KVCache evicts the key-value with the lowest W value from the cache. Key-value data with high *AccessFrequency* is considered hot and is given a higher weight in the numerator of the formula. Therefore, key-values with relatively low access frequency are evicted first. Similarly, since the goal of KVCache is to promote small, hot key-values from BucketCache to mitigate the low cache utilization problem of BucketCache, key-values with larger *DataSize* are given lower weight and are evicted faster. The *LastAccessTime* is also factored into the weight, with larger values of *CurrentTime* - *LastAccessTime* resulting in lower weights and faster eviction. That is, if *LastAccessTime* differs significantly from the current time, it indicates that it has not been accessed for a long time.

We note that the W value is calculated based on the current *TimeStamp*, Since computing the weight of all cached key-values on every insertion or eviction is inefficient, we use a Min Heap to store the keys and their corresponding weights. We periodically update the heap to reflect changes in the weights of the cached key-values. Additionally, we use low and high watermarks to trigger the cache space reclamation process, further reducing the overhead of cache management.

5 EVALUATION

In this section, we evaluate the performance impact of the dynamic data block size control scheme and KVcache that we implemented in HBase 2.2.7, in various aspects, including read amplification, latency, and throughput.

5.1 Experimental Setup

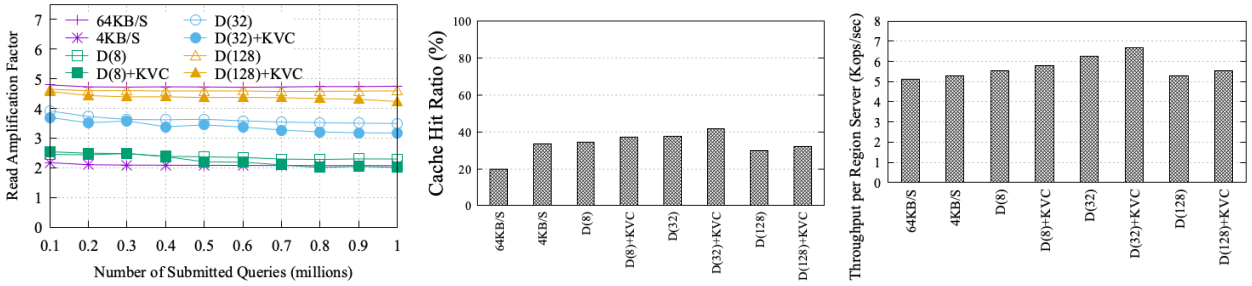
We conducted experiments on two testbed clusters. One is Naver's HBase cluster, as described in Section 2.3. The other is a 12-node cluster, with each node having dual Intel Xeon Gold 5115 processors (2.40GHz, 20 vCPUs with hyper-threading enabled), 64 GB DDR4 ECC memory, and two 7200rpm 1 TB HDDs, one for the OS (Ubuntu 18.04) and the other for HDFS (ver. 3.1.4). The nodes are connected through a one-gigabit Ethernet switch. Four nodes are used as data nodes, while the other two are used for primary and secondary NameNodes for HDFS.

For experiments on Naver's HBase cluster, we used real production datasets. However, we did not use real user queries for our experiments. Instead, we used synthetic workloads that model users' queries using Customer Model Behavior Graph (CBMG) [24]. CBMG models the spatial-temporal locality of queries that each user continuously sends to the search engine as a graph. Based on this probabilistic graph, it determines whether the next query will be a query with a similar URL, a new URL, or a previously read URL (i.e., hot URL). In CBMG, the ratio of hot URLs follows a Zipfian distribution. For the workload, we selected 1.5 million hot URLs and generated 15 million queries.

5.2 Performance Evaluation with Real Datasets and CBMG Workloads

Figure 8(a) shows the read amplification factor for the CBMG workload in the Naver's HBase cluster. The performance of the vanilla HBase with the default configuration is denoted by `64KB/S`, where the data block size is fixed to 64KB. With this default setting, the read amplification factor - the ratio of the number of pages read from the disk to the number of pages containing the data being read - is more than 4.7. However, if the data block size is statically set to 4KB, i.e., `4KB/S`, the read amplification factor drops to 2. It's worth noting that the read amplification factor is not reduced to 1. This is because data blocks include key-value pairs of variable length and are not perfectly aligned to 4KB disk pages. Aligning data blocks to disk pages would result in padding bytes that waste disk space and hurt disk utilization.

Enabling dynamic block size control (`D(n)`, `n` is corresponding to `MaxNumData`) ensures that a data block is closed if its size exceeds



(a) Read Amplification Factor (b) Cache Hit Ratio (c) Throughput

Figure 8: Performance Effect of Dynamic Block Size Control and Key-Value Cache (Real Datasets); *s*KB/S: Static data block size fixed at *s* KB, D(*n*): With dynamic data block size control enabled (`MinDataBlockSize`=4 KB, `MaxDataBlockSize`=64 KB, and `MaxNumData`=*n*), +KVC: With KVCache enabled

64KB or if it surpasses a minimum size of 4KB and contains more than a certain number of small key-values. As a result, the data block size falls between 4KB and 64KB, depending on `MaxNumData`. Consequently, increasing `MaxNumData` leads to a rise in the read amplification factor. In the workload, setting `MaxNumData` to 8 yields read amplification factors similar to `4KB/S`, whereas setting `MaxNumData` to 128 produces read amplification factors comparable to `64KB/S`. If KVCache is enabled, the read amplification factors decrease slightly due to more frequent cache hits.

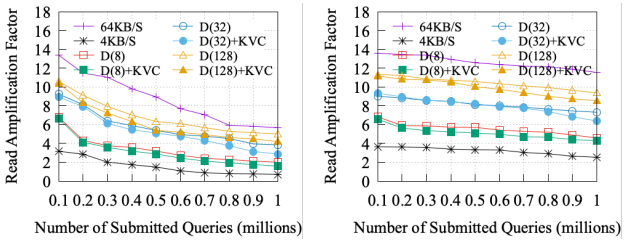
Figure 8(b) shows the cache hit ratio for the same experiments shown in Figure 8(a). `64KB/S` shows the lowest cache hit ratio because it suffers from cache pollution caused by large data block size. In contrast, `4KB/S` shows a 68% higher cache hit ratio than `64KB/S`, i.e., 33.46% vs. 19.87% because it avoids caching unsolicited disk pages. `D(8)` has a similar cache hit ratio with `4KB/S` because their block sizes are similar. Interestingly, the cache hit ratio of `D(32)` is 12.6% higher than that of `4KB/S` although the data block size of `D(32)` is larger than 4KB. This is because the CBMG workload has spatial-temporal locality, and caching adjacent small key-values together can get the advantage of locality. However, if the `MaxNumData` is set too large, i.e., 128, it suffers from the cache pollution and the cache hit ratio decreases. We note that enabling the KVCache helps improve the cache hit ratio as expected.

Figure 8(c) shows the query processing throughput per HBase RegionServer. Interestingly, `4KB/S` does not exhibit higher throughput than `64KB/S`, despite its lower read amplification factor and higher cache hit ratio. This can be attributed to the fact that 4KB data blocks suffer from random reads, resulting in disk seek overhead, whereas `64KB/S` takes the benefits of sequential reads. However, the dynamic block size control strikes a balance between cache pollution and read amplification, thereby exhibiting higher throughput than vanilla HBase. In particular, `D(32)` and `D(32+KVC)` demonstrate 22% and 30% higher throughput, respectively, compared to the default `64KB/S`.

5.3 Performance Evaluation with YCSB

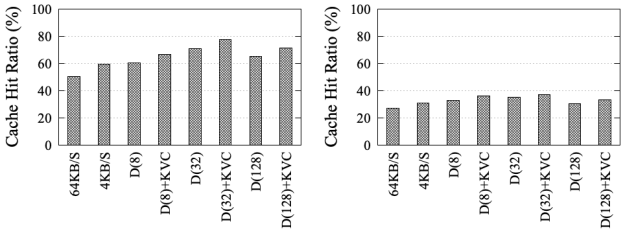
In this section, we evaluate the performance effect of the proposed schemes using the standard YCSB benchmark on the second testbed, i.e., 12 node HBase cluster.

First, in the experiments shown in Figure 9, we measure the read amplification factor using YCSB Workload B in Zipfian and uniform distributions. Workload B is a read-intensive, with 95% of



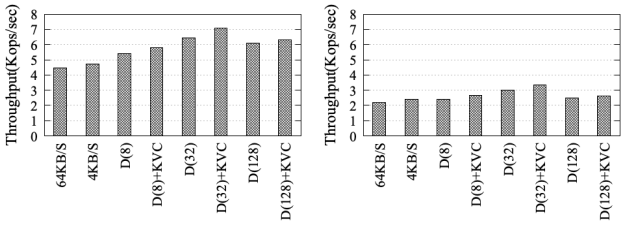
(a) YCSB B (Zipfian) (b) YCSB B (Uniform)

Figure 9: Read Amplification Factor



(a) YCSB B (Zipfian) (b) YCSB B (Uniform)

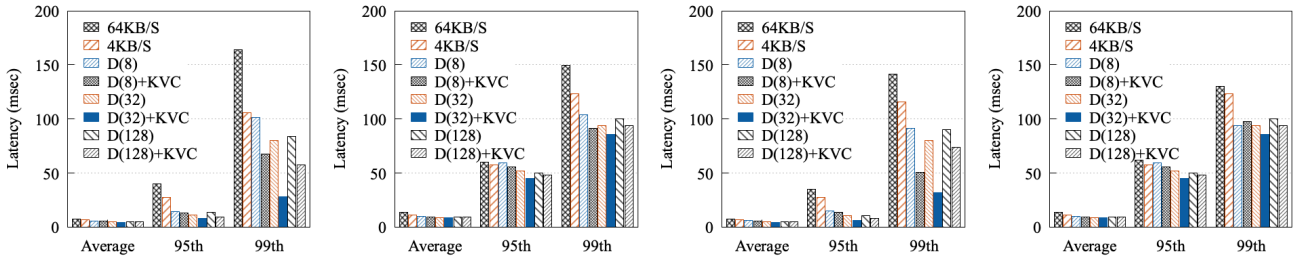
Figure 10: Cache Hit Ratio



(a) YCSB B (Zipfian) (b) YCSB B (Uniform)

Figure 11: Throughput

queries being reads and 5% writes. Before running each workload, we pre-load 500 GB of key-values in the database. The size of key-values follows a Zipfian distribution, similar to the real datasets in Naver's HBase cluster, where most key-values are 256 bytes or



(a) YCSB B (Zipfian) (b) YCSB B (Uniform) (c) YCSB C (Zipfian) (d) YCSB C (Uniform)

Figure 12: Latency Comparison

smaller than 64KB. Note that the proposed schemes do not affect the write performance, and therefore the write-intensive workload results are not presented. We also conducted experiments using the YCSB C (100% reads) workload as well, but the results did not differ significantly from those of workload B. Due to constraints on the number of pages, YCSB C results are omitted.

Figure 9(a) shows that using the default 64KB data block reads about 14× the number of actual pages requested by clients for the first 100,000 queries. As more queries are submitted, the read amplification factor decreases due to cache hits. Reducing the data block size to 4KB reduces the read amplification, and each query reads about 3× the number of actual pages requested for the first 100,000 queries. As more queries are submitted, the read amplification factor drops below 1 due to cache hits. By enabling the dynamic block size control, the data block sizes range from 4KB to 64KB, and the read amplification factor is also in the range between the read amplification factor of `4KB/S` and that of `64KB/S`. It strikes a balance between cache pollution and read amplification, resulting in higher throughput compared to the vanilla HBase.

Figure 9(b) shows the results for the YCSB workloads in uniform distribution. In contrast to the Zipfian distribution, uniformly distributed keys have low spatial-temporal locality, resulting in low cache hit ratios for all configurations. As a result, the read amplification factors are higher compared to the Zipfian distribution.

Figure 10 shows the cache hit ratio results for the same experiments shown in Figure 9. Similar to the CBMG results, `4KB/S` yields higher cache hit ratios than `64KB/S`, as smaller data blocks help alleviate the cache pollution problem. However, as `MaxNumData` increases, the cache hit ratio also increases since adjacent key-values benefit from spatial-temporal locality and are preloaded. When KVCache is enabled, the cache hit ratio further improves, leading to a 6.4~10.04% boost. Nevertheless, the outcomes demonstrate that the effect of KVCache is not significant when search keys are uniformly distributed and spatial-temporal locality is low.

Figure 11 demonstrates the effectiveness of the dynamic block size control method and KVCache in improving throughput by reducing read amplification and increasing cache hit ratio. The results indicate that this approach can achieve up to 61.1% higher throughput than vanilla HBase with 64KB block size.

Finally, Figure 12 shows the average, 95th percentile, and 99th percentile tail latency. The use of large 64KB data blocks results in high tail latency due to large disk I/Os. On the other hand, the tail latency with 4KB DataBlocks is not optimal due to frequent disk seeks. However, dynamic block size control balances trade-offs between the cache pollution and read amplification, resulting in lower tail latency than `4KB/S`. KVCache further reduces tail latency by yielding higher cache hit ratios and eliminating the overhead of parsing and retrieving key-values from encoded data blocks.

6 RELATED WORK

In this section, we highlight some of the most relevant studies. Several techniques have been recently proposed to improve read performance by reducing search ranges in the key space [15, 18, 22, 35]. In particular, PinK [15] proposed to use fractional cascading shortcuts to reduce search ranges of overlapping sorted arrays. MatrixKV [35] boosts read performance by minimizing the overlap of Level 0 to prevent reading too many Level 0 indexes. Wiskey [23] improves read performance by separating keys and values, allowing more keys to be cached in MemTables. Spatially Fragmented LSM-tree (SFM) [18] proposed to partition key spaces into read-intensive and non-read-intensive ones, and delay compactions for the non-read-intensive key spaces. TridentKV [22] is a read-optimized RocksDB that partitions the data based on access frequency, with the most frequently accessed data stored in the most easily accessible partition. It also employs an adaptive indexing technique to adjust the index structure to reduce the number of disk seeks.

Efforts have also been made to improve read performance by efficiently utilizing memory. Accordian [5] propose to use multiple layers of in-memory sorted arrays, such that it can reduce read latency by holding more keys in memory. AC-Key [34] is similar to this study in that it proposes a hierarchical cache structure consisting of three caching components: block cache, key-value cache, and key-pointer cache. Our approach differs from AC-Key as our main emphasis is on managing I/O granularity and caching units.

There also exist some works that tried to improve read performance by optimizing Bloom filters. In particular, ElasticBF [20] improves read performance even with skewed workload by dynamically adjusting the false positive rate based on data hotness and access frequency. For range queries, RocksDB [11] uses a prefix Bloom filter to reduce read amplification. SuRF [36] supports filtering for both point and range queries to reduce read I/O.

There have been various other methods proposed that are orthogonal yet complementary to our research. Bourbon [9] and TridentKV [22] improved read performance of LSM trees by implementing the *learned index* and avoiding index block reads. FlashKV [37] proposed to use a file system at the user level with SPDK, instead of the Linux file system, to manage raw flash devices at the application layer. Endure [14] argues that existing methods optimize for a

single performance metric, which may lead to poor performance under different workloads. Endure introduces a novel metric called *Endurance Score* to balance read and write performance.

7 CONCLUSION

This study proposes two effective methods to tackle the read amplification problem in HBase. The conventional approach of using fixed-size large data blocks leads to reading a large amount of unnecessary data from the disk, thereby causing cache pollution issues. On the other hand, small data blocks mitigate the read amplification problem but may result in frequent disk seeks, causing degraded performance due to random reads. To address these challenges, we introduce a dynamic data block size control mechanism based on the size of key-values stored in data blocks. This approach stores large key-values in large data blocks and small key-values in small data blocks, effectively reducing unnecessary disk IO and balancing the trade-offs between cache pollution and read amplification.

Furthermore, we propose KVCache, a fine-grained cache component implemented on top of HBase's conventional block cache to optimize performance. KVCache improves cache efficiency by evicting low-utilized data blocks from the block cache and moving frequently accessed key-values to KVCache. By utilizing the dynamic block size control method and KVCache, we significantly enhance the read throughput of HBase, achieving up to 61.1% higher throughput than vanilla HBase with 64 KB block size.

ACKNOWLEDGEMENT

This work was supported by Naver Corp, and also in part by NRF (grant No. NRF2022R1A2C2091680) and IITP (grant No. 2021-0-01817). The corresponding author is Beomseok Nam.

REFERENCES

[1] Naver Portal. https://www.naver.com/.
[2] Powered By Apache HBase. https://hbase.apache.org/poweredbyhbase.html.
[3] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache storage for the next billion. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.
[4] Hyokyung Bahn, Sam H. Noh, Sang Lyul Min, and Kern Koh. Using full reference history for efficient document replacement in web caches. In *Proceedings of the 2nd Conference on USENIX Symposium on Internet Technologies and Systems*, 1999.
[5] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. Accordion: Better memory organization for LSM key-value stores. *Proceedings of the VLDB Endowment*, 11(12):1863–1875, 2018.
[6] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 1999.
[7] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.
[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. BigTable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
[9] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.
[11] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing Space Amplification in RocksDB. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2017.
[12] HBase. https://hbase.apache.org/.
[13] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, 2017.
[14] Andy Huynh, Harshal A. Chaudhari, Evimaria Terzi, and Manos Athanassoulis. Endure: A Robust Tuning Paradigm for LSM Trees under Workload Uncertainty. *Proceedings of the VLDB Endowment*, 15(8):1605–1618, apr 2022.
[15] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.
[16] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, 2019.
[17] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.
[18] Hoyoung Lee, Minho Lee, and Young Ik Eom. SFM: Mitigating Read/Write Amplification Problem of LSM-Tree-Based Key-Value Stores. *IEEE Access*, 9:103153–103166, 2021.
[19] LevelDB. https://github.com/google/leveldb.
[20] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.
[21] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–13, 2011.
[22] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. TridentKV: A Read-Optimized LSM-Tree Based KV Store via Adaptive Indexing and Space-Efficient Partitioning. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1953–1966, 2022.
[23] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.
[24] Kaszo Mark and Legany Csaba. Analyzing Customer Behavior Model Graph (CBMG) using Markov Chains. In *Proceedings of the 11th International Conference on Intelligent Engineering Systems*, 2007.
[25] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, 2015.
[26] Fei Mei, Qiang Cao, Hong Jiang, and Lei Tian Tintri. LSM-tree Managed Storage for Large-scale Key-value Store. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017.
[27] Nicolas Niclausse, Zhen Liu, and Philippe Nain. A New Efficient Caching Policy for the World Wide Web. In *Proceedings of Workshop on Internet Server Performance (WISP'98)*, 1998.
[28] Oracle Berkeley DB. https://www.oracle.com/database/technologies/related/berkeleydb.html.
[29] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys*, 35(4):374–398, dec 2003.
[30] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.
[31] P.A. Riyaz and Surekha Mariam Varghese. SLSM - A Scalable Log Structured Merge Tree with Bloom Filters for Low Latency Analytics. *Procedia Technology*, 24:1491–1498, 12 2016.
[32] RocksDB. https://rocksdb.org/.
[33] Subhadeep Sarkar, Dimitris Staratzis, Ziehen Zhu, and Manos Athanassoulis. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment*, 14(11):2216–2229, jul 2021.
[34] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du. AC-Key: Adaptive Caching for LSM-Based Key-Value Stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (ATC)*, 2020.
[35] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.
[36] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*, 2018.
[37] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. FlashKV: Accelerating KV performance with open-channel SSDs. *ACM Transactions on Embedded Computing Systems*, 16(5s):1–19, 2017.
Devices.