



DumpKV: Learning based lifetime aware garbage collection for key value separation in LSM-tree

Zhutao Zhuang
Microsoft
Sun Yat-sen University
zhuangzhutao@gmail.com
zhuangzht@mail2.sysu.edu.cn

Xinqi Zeng
Sun Yat-sen University
xinqizeng2002@outlook.com

Zhiguang Chen*
Sun Yat-sen University
chenzhg29@mail.sysu.edu.cn
zhiguang.chen@nscg-gz.cn

ABSTRACT

Key-value separation is used in LSM-tree to store large values in separate log files to reduce write amplification but requires garbage collection to recycle invalid values. Existing LSM-tree typically adopts a static policy to recycle obsolete values, struggling to achieve low write amplification as it is challenging to predefine the static parameters for garbage collection. In this work we propose DumpKV, a learning-based lifetime-aware garbage collection mechanism which achieves lower write amplification. DumpKV trains a machine learning model based on the access history of keys and accordingly uses the lightweight model to predict the lifetime of each key, where the predicted lifetime can be used to guide the garbage collection. To reduce the interference to write throughput introduced by garbage collection, DumpKV conducts feature collection during L0-L1 compaction, leveraging the fact that LSM-tree is small under KV separation. Experimental results show that DumpKV reduces GC write size by 25.7%-53.3% in real-world workloads and 19%-65% in synthetic workloads compared to baseline key-value separation LSM-tree KV stores with small feature storage overhead.

PVLDB Reference Format:

Zhutao Zhuang, Xinqi Zeng, and Zhiguang Chen. DumpKV: Learning based lifetime aware garbage collection for key value separation in LSM-tree. PVLDB, 18(4): 1223-1236, 2024.
doi:10.14778/3717755.3717778

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/BilyZ98/DumpKV>.

1 INTRODUCTION

Log-structured merge (LSM)-tree [41] based storage engine is widely adopted in modern cloud storage and database storage due to its write-friendly peculiarity and simple concurrency control to support write-intensive scenarios including enterprise storage servers and online transactions. Typical key-value storage engines, including RocksDB [12], Titan [43], XStore [54], LevelDB [14] and TerakDB [5], etc., are all based on LSM-tree. The idea of LSM-tree is to

first buffer key-value (KV) writes in memory buffer and then flush them to disk files. Key-value pairs in a given disk file are sorted, and all these disk files are organized into multiple levels where lower-level files contain the most recent writes. Key-value pairs in lower levels are gradually moved towards higher levels via the compaction process.

The standard LSM-tree based storage engine that stores both keys and values in LSM-tree suffers from write amplification because of repeated writes of keys and values during compaction. Such write amplification can reach a factor of at least 50x [35]. Storing both keys and values in LSM-tree also leads to read amplification since a query needs more disk accesses to locate the target key across multiple levels of disk files as the LSM-tree grows in size.

Key-value separation is introduced by Wiskey [35] to reduce write amplification from recurring large value writes during compaction. The main idea of KV separation is to store large values in separate value files and store keys and pointers to values in LSM-tree. As the values have been updated gradually, the storage capacity occupied by invalid values increases, and accordingly the garbage collection (GC) is initiated to reclaim storage space occupied by obsolete values, move still valid values to new value files, and update the key-pointer pairs in LSM-tree if necessary. This design helps to alleviate write amplification because it avoids repeated write of large values during compaction process, and it is useful for workloads whose average value size of KV pairs is large. However, this design comes with the cost of sacrificing scan performance for the reason that values are stored in value files out of order.

Current design and implementation of garbage collection for KV separation struggles to achieve low write amplification because they are mostly based on static policies. Typically, the garbage collection process is triggered when the estimated or measured garbage ratio in value files exceeds some predefined threshold or when value files exceed a defined time-to-live threshold. If the garbage ratio for triggering garbage collection is low, then we can get low space amplification with high write amplification. Otherwise, a high garbage ratio for garbage collection triggering can cause low write amplification but with high space amplification, which means it wastes storage space to store obsolete values.

We argue that the root cause for why current GC solution for KV separation struggles to balance low write amplification and low space amplification at the same time is that key lifetime is not known in advance. Key lifetime is defined as the duration from the time when the key is first initially written to the time it is rewritten, which means the previous write of the key is invalidated. Despite that the idea of leveraging the lifetime of blocks or objects

*Corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 18, No. 4 ISSN 2150-8097.
doi:10.14778/3717755.3717778

to improve garbage collection efficiency has been studied in SSD [21] and file systems [45] it has not been studied how this idea can be applied to garbage collection in KV separation in LSM-tree.

This paper introduces DumpKV, an intelligent LSM-tree KV separation store designed for update-intensive workloads. DumpKV considers the lifetime of keys, using a machine learning model to predict key lifetimes and place values into appropriate files based on these predictions. This method enhances garbage collection efficiency by grouping values with similar lifetimes, reducing write and space amplification.

DumpKV categorizes value files into default, short, and long lifetimes, adjusting dynamically based on workload monitoring. Initially, values are written with a default lifetime. During garbage collection, a binary classification model predicts the remaining lifetime of valid KV pairs, and values are reassigned to the appropriate lifetime category. Garbage collection occurs when the estimated TTL of a value file is reached.

Incorporating a model into DumpKV’s LSM-tree KV separation store to achieve efficient garbage collection without degrading read/write performance presents several challenges. First, effective feature engineering is needed for accurate lifetime prediction. DumpKV uses past write information and generates exponentially decayed write counts to capture short- and long-term access patterns for per-key lifetime prediction. Second, determining short and long lifetime thresholds is crucial. DumpKV continuously collects key lifetimes and periodically determines optimal thresholds based on the cumulative distribution function. Third, minimizing the overhead of feature collection and model prediction is essential. DumpKV generates features during the $L_0 - L_1$ compaction phase and performs model prediction during background garbage collection. DumpKV leverages the small size of the LSM-tree in the KV separation architecture to keep read overhead low. Lastly, DumpKV must adapt to dynamically updated feature values for accurate key lifetime prediction. Unlike previous models trained offline, DumpKV periodically retrains the model and adjusts the training dataset adaptively.

To the best of our knowledge, this is the first work that introduces a learning-based method to help with garbage collection in KV separation in LSM-tree. Contributions of this paper are summarized as follows:

- We propose DumpKV, an intelligent KV separation store architecture that learns lifetime distribution and gives per-key level lifetime prediction during garbage collection to achieve low write amplification and space amplification at the same time.
- We propose effective features engineering techniques solely based on past write access information and feature persistence to help the model do training and prediction.
- We propose a lifetime-aware value file storage structure and garbage collection process to effectively relocate KV pairs of value files that are still valid.
- We implement DumpKV prototype atop of RocksDB, an open-source KV store that is popular in the community and is widely adopted. Experimental results show that DumpKV achieves the lowest GC write size compared to baseline KV stores with a slight extra storage space for feature data.

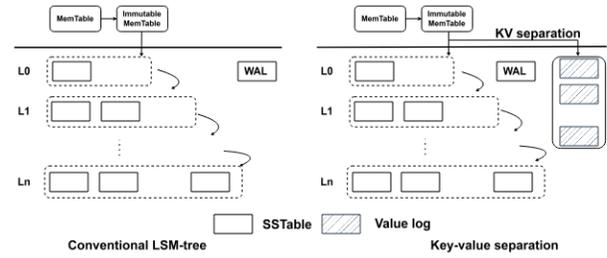


Figure 1: LSM-tree and KV separation

2 BACKGROUND

In this section we will first give introduction to LSM-tree store architecture. We then give introduction to key-value separation technique that aims to reduce write amplification for large value storage. And then we discuss about current garbage collection design and implementations in key value separation in LSM-tree storage engine and gives strengths and weaknesses of different garbage collection approach. Finally, we talk about current learned based research approach in storage system that predicts lifespan of storage object.

2.1 LSM-tree key value store

LSM-tree key value store is a write-friendly storage architecture. Fig 1 depicts a simplified storage architecture of conventional LSM-tree key value stores (e.g., LevelDB and RocksDB). LSM-tree key-value store is an append-only storage structure. KV pairs written by users are first buffered in a memory part called MemTable. When MemTable reaches size threshold, it’s converted into Immutable MemTable, which means there will be no extra writes to this Immutable MemTable, and a new MemTable is created in memory to accept new KV pairs. Then Immutable MemTable is flushed to a disk file called SSTable. LSM-tree key-value store organizes SSTables in $n+1$ levels, denoted by L_0, L_1, \dots, L_n (from lowest to highest level) in disk. Capacity of L_i is configured as a multiple (typically 10x) of that in L_{i-1} (where $1 \leq i \leq n$).

Each level except L_0 is fully sorted which means there is no overlap between SSTables. When L_i reaches size limit a compaction process is started to merge key value pairs from L_i to L_{i+1} ($0 \leq i \leq n-1$). Compaction process first picks candidate SSTables from L_i and then finds the overlapping SSTables in L_{i+1} to be candidates in compaction. Then it sorts all key value pairs from candidate SSTables and writes valid key values pairs to newly created SSTables in L_{i+1} . Finally, input candidate SSTables are deleted, and the compaction process is done. This compaction process incurs extra read and write which increases write amplification. Prior studies show that write amplification of conventional LSM-tree key value stores can reach up to 50x write amplification [35].

To perform read process to do key lookup, a conventional LSM-tree KV store first search query key in MemTable, and if it does not find the query key then it performs another search in Immutable MemTable if there is any. If the query key does not exist in MemTable and Immutable MemTable, LSM-tree KV store then searches in each level of LSM-tree starting from L_0 to higher levels. In L_0 , LSM-tree KV store searches all SSTables. In levels between L_1

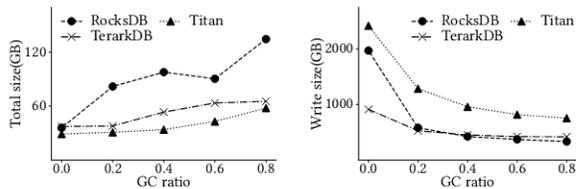


Figure 2: Total storage size and write size for different GC trigger ratios

and L_n LSM-tree KV store does binary search to locate the candidate SSTable whose key range overlaps with lookup key.

2.2 Key-value separation and garbage collection

Large size values are commonly found and contribute a considerable amount of traffic in real-world KV workloads. For example, TiDB [16, 32] is a transactional database built on top of the KV storage engine and maps each row in the table into a KV pair whose value size can reach hundreds of kilobytes. Atlas [24] manages key-value pairs in cloud storage with values exceeding 128 KB. Large-size values significantly contribute to traffic in practical KV workloads, according to field studies [1, 37, 60].

To reduce write amplification caused by repeated large value writes during compaction, Wiskey [35] proposes KV separation architecture. Fig. 1 shows the storage structure introduced by Wiskey. Wiskey’s KV separation architecture reduces write amplification during compaction by storing values in a separate file and using a smaller index pointer in the LSM-tree. This reduces compaction overhead and LSM-tree size, improving read amplification and point query performance. Key lookups involve locating the KV pair in the LSM-tree and then reading the value from the value file using the index pointer. Garbage collection reclaims space by writing valid values to new files.

Several other KV separation designs and implementations are introduced to trade off between read and write performance brought by garbage collection [5, 12, 32, 43]. They mainly differ in value file storage structure, trigger condition of garbage collection process, and victim value file selection strategy. Titan [43] adopts a similar garbage collection policy as Wiskey but with a different garbage estimation implementation. It triggers garbage collection when the estimated garbage ratio of value files reaches a threshold. RocksDB [12] develops its own KV separation design and implementation and integrates garbage collection as part of the compaction process, which is different from Wiskey. TerakDB [5] removes new value pointer write-back to reduce interference of GC by storing values in a separate value file called v-SSTable during GC.

2.3 Learning-based storage system

A fair amount of work has been done to apply machine learning model do prediction for various parts in various aspects of storage system stack to benefit read/write rate. Typical use cases are increasing cache hit rate or improving garbage collection efficiency. Leaper [61] uses model to predict which new blocks are hot after compaction and prefetches those new blocks into cache during compaction. LRB [48] uses machine learning to approximate the

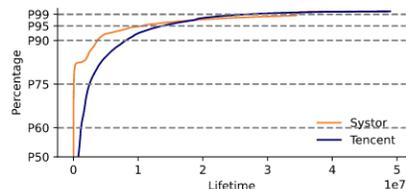


Figure 3: CDF of lifetime for Systor and Tencent trace

Belady MIN algorithm to predict objects with furthest request and evict those objects from cache to increase cache hit rate.

One main challenge in the integration of machine learning into storage systems is the necessity for the model to impose minimal training and inference overhead on critical system performance metrics, such as read and write throughput. Current solutions can be broadly classified into two categories. The first approach involves constraining the model size and simplifying input features to reduce both the training and inference time [15, 55, 61]. For example, LinnOS [15] applies a light-weight neural network with weight quantization and reformatting feature integers into decimal digits to do binary classification and do SSD performance inferring at a per I/O granularity to reduce I/O latency. QB5000 [36] develops a robust forecasting framework to predict future query arrival rates using logical composition of queries with a clustering-based technique. The second one is reducing model calling frequency or putting model inference in a non-critical path [38]. For example, LLAMA [38] uses a hashing-based mechanism to identify contexts previously seen and only execute model inference if the lookup fails to amortize model executions overhead over the lifetime of a long-running server.

3 MOTIVATION

Despite that lifetime information is used for garbage collection in storage hardware such as SSDs [21, 40, 53, 53, 59], file systems [27, 45] and memory allocation [38], it has not been studied how lifetime information can be leveraged to improve garbage collection efficiency and reduce write amplification in KV separation for LSM-tree.

Besides, current garbage collection designs for KV separation suffer from balancing low write amplification and space amplification because they only take static parameters to trigger garbage collection. Thus, current garbage collection implementation in key value separation in LSM-tree either trades off higher write amplification to have less space amplification or wastes extra storage space to maintain low write amplification.

Several works [6, 32] use recent write count to differentiate hot and cold keys, simplifying lifespan prediction into a binary classification task. However, this method relies on threshold-based garbage collection triggers and fails to capture lifetime patterns, making it ineffective for reducing write amplification in less-skewed workloads. Therefore, incorporating more features in lifetime classification is necessary to achieve lower write amplification.

Fig. 2 shows the total write size and total size of three typical KV separation implementations, RocksDB, TerakDB, and Tian, under different garbage collection trigger ratio parameter settings. All 3

KV separation LSM-tree based stores show a similar trend, and a lower garbage collection trigger ratio leads to a lower total size but with higher write amplification.

Knowing each key’s lifetime in advance allows us to group keys with similar lifetimes, enhancing garbage collection efficiency and reducing both write amplification. This motivates the use of a machine learning model to predict key lifetimes.

Fig. 3 illustrates the cumulative distribution function (CDF) of lifetime distributions derived from real-world block storage workload traces [25, 26, 65]. The data reveals a long-tail distribution, indicating that the majority of writes have short lifetimes, while a smaller proportion exhibit significantly longer lifetimes. This observation motivates the simplification of the lifetime prediction task into a binary classification problem.

To achieve an effective and efficient learned LSM-tree KV separation store, several challenges must be addressed:

Feature generation. Effective key feature generation for training and inference is needed without full stack application information. Unlike PCStream[21] and LLAMA[38], which use rich call stack traces, standalone key-value stores lack such data. Efficient feature engineering based on past write access information is required to capture lifetime patterns.

Performance overhead. Model training and inference should minimally impact system read and write performance. Unlike user-facing ML applications, LSM-tree key-value stores are sensitive to performance. Balancing write amplification, space amplification, and write performance is critical.

Persistent feature storage. Key features must be stored persistently and efficiently for model training and inference, avoiding data loss and enabling quick retrieval for lifetime prediction.

Online model updates. The model needs to be updated online to adapt to dynamically changing feature values.

4 DUMPKV DESIGN

In this section, we present the design of DumpKV and illustrate how we address the problems mentioned in the last section. First, we give DumpKV system overview. Then we describe how features are generated for use in model prediction and how features are stored efficiently and effectively. Next, we introduce how model training and inference work in DumpKV. And then we talk about how garbage collection works in DumpKV. Finally, we discuss the crash consistency mechanism and implementation details.

4.1 System overview

Figure 4 illustrates the architecture of DumpKV, a key-value separation design of LSM-tree. DumpKV uses a machine learning model to predict key lifetimes and stores values in files with lifetime limits. Large values are initially written to default lifetime files with dynamically adjusted thresholds. Feature and lifetime data collection occur during $L_0 - L_1$ compaction and garbage collection. Model training is triggered periodically once sufficient training samples are collected. Garbage collection is initiated when the value files reach their expected lifetime, with the model predicting the remaining lifetime of valid KV pairs.

DumpKV leverages past update intervals and exponentially decayed write counters as features, calculated efficiently. It uses a

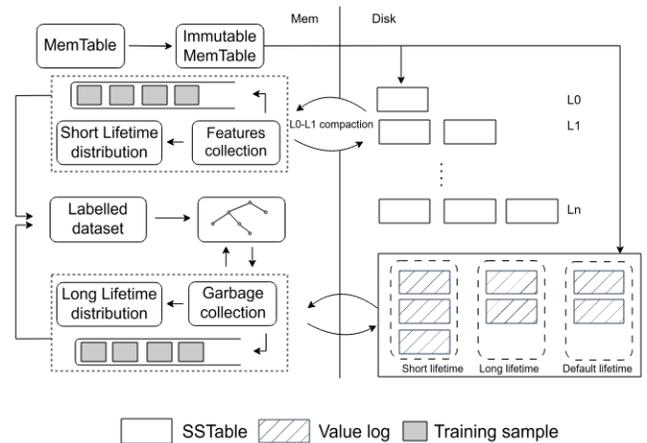


Figure 4: DumpKV architecture

lightweight gradient boosting tree model for periodic training and prediction during background garbage collection. Instead of predicting the remaining lifetime during the Flush process, DumpKV assigns adaptive default lifetimes to KV pairs to balance write performance and write amplification, avoiding costly feature data generation and model inference during Flush.

To store features effectively and prevent data loss, DumpKV integrates features into the LSM-tree. Training and inference features are queried from the LSM-tree, incurring minimal overhead due to the small LSM-tree size under the KV separation architecture. For online and continuous learning, DumpKV periodically generates feature data and trains the model to adapt to dynamically updated feature values.

Given the long-tail distribution of lifetime data, a binary classification model is suitable for simplifying model training and inference. A regression model is unsuitable due to the vast range of key lifetimes, from single digits to millions, which complicates effective training. Additionally, precise lifetime prediction is not crucial since garbage collection is infrequent. With a multi-class classification model, misclassification is likely for adjacent lifetime classes due to their increasing numerical values. For instance, if a model predicts a 1M remaining lifetime but the actual is 2M, it results in extra CPU time and write amplification. Conversely, predicting 2M when the actual is 1M only wastes a small amount of storage for a short period. Therefore, it is preferable for the model to overestimate lifetimes, trading off storage space for reduced CPU time and write amplification. Larger gaps between lifetime classes simplify model training by creating broader classification boundaries.

4.2 Features and storage

4.2.1 Features. DumpKV uses dynamic and static features together to train models.

Deltas. *Delta* is defined as an interval between consecutive writes of the same key. Δ_0 is the elapsing time since the latest previous writes. Δ_1 indicates the time interval between the previous two writes. We choose to use key sequence as a time unit instead of real time, such as a microsecond or second, for the following two reasons. First, time movement is driven by new

Table 1: List of notations used in section below

Notation	Description
$H_s(x), H_l(x)$	Lifetime distribution histogram for short and long lifetime value files
l_s, l_l, l_d	TTL for short, long and default lifetime value files
s_idx, l_idx	Lower bound index for l_s and l_l in $EDWC_i$ Windows
$gc_ratio_d,$ $gc_ratio_s,$ tc_ratio_l	GC invalid ratio for default, short and long lifetime value files
sp_0, lp_0, sp_1, lp_1	Base and extra percentage value for dynamic short and long lifetime of value files
$ini_sp_0, ini_lp_0,$ $ini_sp_1, ini_lp_1,$ ini_dp_1, ini_dp_0	Initial percentage value for percentage value of sp_0 and lp_0, sp_1, lp_1, dp_1
$\alpha_s, \alpha_l, \alpha_d$	Steepness factor for transformation function
$\beta_s0, \beta_s1, \beta_l0,$ β_l1, β_d	Midpoint shift of transformation function for short, long and default lifetime adjustment

incoming writes rather than clock time, which is a more logical trigger for initiating a garbage collection job. Second, key sequence is used as a unique timestamp in the current LSM-tree store engine, so it is directly available. Note that key lifetime and value file lifetime share the same time unit as Δ which uses key sequence as time unit for consistency.

DumpKV uses up to 32 Δ for a key, and Δ_i where $i \geq 32$ is discarded. To maintain numerical stability of training data, DumpKV maps Δ value to index of exponentially increasing intervals, which are 1M, 2M, 4M, 8M, etc. For example, a 3M Δ value is mapped to 2 in training data. We use past update interval as a feature because it captures the past write access information of a key.

Δ values remain constant over time and are unaffected by the arrival of new writes for the same key. This characteristic simplifies storage and minimizes the need for frequent updates of feature values. It should be noted, however, that the storage space required for the Δ varies for each key. For example, key A with 10 past writes has 9 Δ , while key B with 2 past writes only has 1 Δ that occupies storage space. This helps save space for feature storage with less frequently updated keys.

Exponentially decayed write counters(EDWCs). DumpKV applies $EDWCs$ to capture update frequency information. $EDWCs$ track the write access count in a specific time window. Each $EDWC_i$ is initialized as 0. When a new write request arrives, $EDWC_i$ where $i \geq 0$ is updated according to the following expression:

$$EDWC_i = 1 + EDWC_i \times 2^{-\Delta/\Delta_0/2^{20+i}} \quad (1)$$

Each $EDWC_i$ is not updated until a new write request of key arrives, which is different than that Δ_0 is updated each time it's used for model prediction. $EDWC_i$ with a larger value i cover a longer time window. For example, a key that was updated frequently

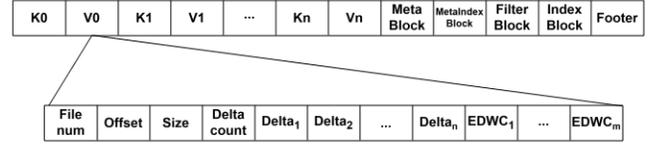


Figure 5: Feature storage format in LSM-tree

10M timestamp ago but is rarely updated now would have a low $EDWC_1$ but still have a large $EDWC_4$. So keys with few write accesses recently could still have a high write access count in the long term.

$EDWC$ is motivated by prior work in block storage caching [28], cache prediction [48] and video popularity prediction [50]. The idea is to use $EDWCs$ to track a long-term trend of popularity by approximating the decay rate of object popularities. The write access pattern of keys is captured with multiple $EDWCs$ with different decayed constants that represent different time windows.

Additional optional features. Besides dynamic features that are updated when each new write arrives, other features can be used to help model determine the lifetime of keys. For example, the size of the value and the type of key that is related to the application can be utilized. Additional features from higher-level applications or databases can be leveraged, such as logical features like tables/columns or the write query's syntax tree. These logical features, along with arrival rate patterns as introduced in QB5000 [36], can help with key lifetime prediction.

4.2.2 Feature storage . Fig.5 shows feature storage format. The value part in SSTable in LSM-tree first stores the value file number, value offset, and value size in the corresponding value file. After these three fields, DumpKV stores feature data to be used for future model prediction. Δ Count is used to indicate how many Δ are stored in the value part, and it only takes one byte of storage space. Following that are the actual values for Δ . Following that is $EDWC_i$ data for the current key. DumpKV stores up to 32 past distances Δ and 10 $EDWC$ for each key that has at least one previous write. Table 2 shows feature data storage size for keys with different accumulated write counts. Only 1 byte is required for keys that are first written to the storage engine and have no past writes because no past distances and $EDWC$ features are stored in LSM-tree. Maximum storage space for feature data is 297 bytes, which is affordable given that value size is usually large in KV separation architecture, and it's rare to have keys with more than 32 past writes that take up the majority of overall write requests. DumpKV uses varint encoding to save more storage space.

4.3 Training data collection

DumpKV maintains two training sample collection queues to collect training samples from $L0 - L1$ compaction and garbage collection separately. A separate consumer thread is then responsible for gathering samples from these two queues, doing data labeling, and building the training dataset.

The rationale behind maintaining two separate queues for collecting training samples is as follows: A true lifetime label for past writes of the same key can be obtained during $L0 - L1$ compaction. This queue helps the model learn lifetime distribution for keys that

Table 2: Feature storage overhead for keys with different numbers of past write requests.

# of past writes	0	1	2	3	4-12	13-32	> 32
Feature size	1	49	57	65	<127	<297	297

have at least two writes. To help model pay attention to one-time write keys and keys that expect to have long lifetime DumpKV maintains another queue to collect training data from garbage collection process.

In the process of $L0-L1$ compaction, for each key that is iterated, DumpKV performs a search operation in levels Li where $1 \leq i$ to identify any previous write of the same key. The lifetime of the previous write is calculated by subtracting the sequence number of the current write from the sequence number of the previous write. Subsequently, the feature data of the previous write is extracted from the value part in the LSM-tree and is appended to the $L0 - L1$ compaction queue, along with its lifetime. Then all $EDWC_i$ of previous writes of the same key are updated according to Eq.1 and they are written to the value part in LSM-tree together with the latest Δ feature with feature storage format as shown in Fig.5 for future feature collection, model training, and calling.

The rationale for collecting training data and key lifetime during $L0 - L1$ compaction is as follows: First, all keys are moved from lowest level $L0$ to higher level through $L0 - L1$ compaction, so DumpKV can accurately capture lifetime distribution by collecting lifetime of all keys during $L0 - L1$ the compaction process. If this data collection process is put off to $L1 - L2$ compaction or a higher-level compaction process, then there is a risk that some key lifetime information is lost because obsolete keys are dropped during the compaction process. Second, $L0 - L1$ compaction process includes the latest inserted keys, which helps bring the latest training data to the training dataset. Third, compaction overhead is small under key value separation architecture because LSM-tree is small and large value write to value file in Flush process dominates foreground write overhead. Since SSTables in $L0$ are unsorted and there might be multiple same keys in $L0$ during compaction, DumpKV maintains the correctness of key lifetime collection in $L0$ by checking the existence of the same key in $L0$ and calculating the lifetime of sequential writes of the same key.

The size of the LSM-tree is inherently small under KV separation architecture, which results in a minimal overhead when performing point queries within the LSM-tree, without the need to read values from the value file. This efficiency allows for the majority of the LSM-tree to be stored in the block cache.

During the garbage collection process, DumpKV calculates the elapsed lifetime for each valid key. This is achieved by subtracting the sequence number of the key from the current timestamp sequence number. Subsequently, the feature data of this key is extracted from the value part. This data is then appended to the garbage collection training sample collection queue, along with the elapsed lifetime, which is included as a speculated lifetime label. The probability of this action is equivalent to the current garbage collection valid ratio.

4.4 Data labelling

Data labeling is required to assign a lifetime label for each training sample to train the model. DumpKV does a binary classification task and gives binary labels to indicate that whether a key will have a short remaining lifetime or a long remaining lifetime.

DumpKV applies different data labeling strategies for samples coming from $L0 - L1$ compaction training sample queue and those coming from the garbage collection training sample queue. For training samples that come from $L0 - L1$ compaction queue, DumpKV excludes samples whose ground truth lifetime is shorter than the default lifetime l_d because those keys with a lifetime that is shorter than l_d are dropped during the first garbage collection anyway, so there is no need to pay attention to these keys. Subsequently, a sample c is labeled with the following expression:

$$label_c = \begin{cases} 1 & \text{if } \Delta a_0 - l_d > l_s \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

Equation (2) shows that a sample is labeled as 1, i.e., a long remaining lifetime if the duration between the last two adjacent write requests is greater than the sum of the short lifetime threshold and the default lifetime threshold for value files.

The process of labeling samples from the garbage collection sample queue presents a challenge due to the unknown ground truth lifetime. Despite this, it is imperative to include keys in the training dataset that have either a long lifespan or are written only once. This inclusion allows the model to predict a longer remaining lifetime for these keys during the garbage collection process. DumpKV introduces a simple but effective heuristic rule-based labeling method. It assigns a label of 1, indicating a long remaining lifetime, to samples that are written only once. The following expression 3 is used to assign label for sample g with at least two past writes from garbage collection training sample queue

$$label_g = \begin{cases} 1 & \text{if } EDWC_{s_idx} > 1.0 \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

The rationale is that the model is advised to deem this key as inactive if keys have fewer than one write operation in the l_s window, implying a long remaining lifetime.

4.5 Dynamic lifetime adjustment

During the training data collection process, lifetime distribution monitoring is conducted to identify optimal short and long lifetime points in a dynamic way. These two points are subsequently utilized for data labeling in collected training samples.

DumpKV employs two histograms, short lifetime and long lifetime, to capture different lifetime points. This approach is based on the observation that lifetime data from $L0 - L1$ compaction accurately reflects the true lifetime for keys with at least two writes, unlike data from garbage collection, which is less accurate for longlived keys written only once. The short lifetime histogram is updated with ground truth data from $L0 - L1$ compaction, while the long lifetime histogram is updated with speculated data sample from garbage collection. Histograms use rangebased, lockfree method to ensure minimal overhead for lifetime distribution monitoring.

And then short and long lifetime points l_s and l_l is calculated periodically based on current garbage collection ratio with following expression

$$\begin{cases} sp_0 = ini_sp_0 * \sigma(\alpha_s * (1.0 - gc_ratio_s - \beta_{s0})) \\ sp_1 = ini_sp_1 * \sigma(\alpha_s * (1.0 - gc_ratio_s - \beta_{s1})) \\ l_s = H_s(sp_0 + sp_1) \end{cases} \quad (4)$$

$$\begin{cases} lp_0 = ini_lp_0 * \sigma(\alpha_l * (1.0 - gc_ratio_l - \beta_{l0})) \\ lp_1 = ini_lp_1 * \sigma(\alpha_l * (1.0 - gc_ratio_l - \beta_{l1})) \\ l_l = H_l(lp_0 + lp_1) \end{cases} \quad (5)$$

where

$$\sigma(x) = 1/(1 + e^{-x}) \quad (6)$$

DumpKV calculates sp_0 and lp_0 to get a base percentage value for short and long lifetime value files which ensures that lower bound of lifetime is close to 0 if gc_ratio is close to 1. Variables sp_1 and lp_1 is more sensitive to low gc_ratio because β_1 is higher than β_0 and are used to be added with sp_0 and sp_1 to get final percentage value to get a proper lifetime value from lifetime histograms H_s and H_l . Variables α_s and α_l control magnitude change for lifetime point. Variables β_0 and β_1 control steepness for lifetime value change. H_s and H_l are histogram functions which return value given specified lifetime distribution percentage. The rationale for these two expression is to assign a higher lifetime value point when the gc_ratio is low, thereby increasing garbage collection efficiency, and vice versa. Adaptive default lifetime for value files generated in flush process is then generated by taking current gc_ratio into account with following expression

$$\begin{cases} dp_1 = ini_dp_1 * \sigma(\alpha_d * (1.0 - gc_ratio_d - \beta_d)) \\ l_d = H_s(ini_dp_0 + dp_1) \end{cases} \quad (7)$$

The idea is that default TTL for value files generated in Flush moves towards l_l when gc_ratio is low and vice versa. To mitigate the issue of data imbalance in the training dataset, DumpKV establishes an equal sample count threshold for each category of samples coming from L_0-L_1 compaction sample queue and garbage collection sample queue.

4.6 Model training

Model. DumpKV uses Gradient Boosting Machine(GBM)[13, 19] as model to help train and do prediction of lifespan of keys. GBM is a machine learning model that is lightweight and is highly efficient on CPUs and widely used in many online machine learning application with tabular data[9, 34, 46, 51], which is suitable for performance-sensitive storage applications. Other models, such as linear regression, support vector machine, logistic regression, and two layer neural network struggle to outperform GBM on similar task [48]. Besides, GBM can handle missing values in an efficient way and does not require feature normalization[19, 48].

Prediction target. The model's prediction target is to determine whether the remaining lifetime of keys, which are still valid during garbage collection, is short or long. This transforms the training task into a binary classification task. Specifically, samples with a predicted remaining lifetime close to the short lifetime point are

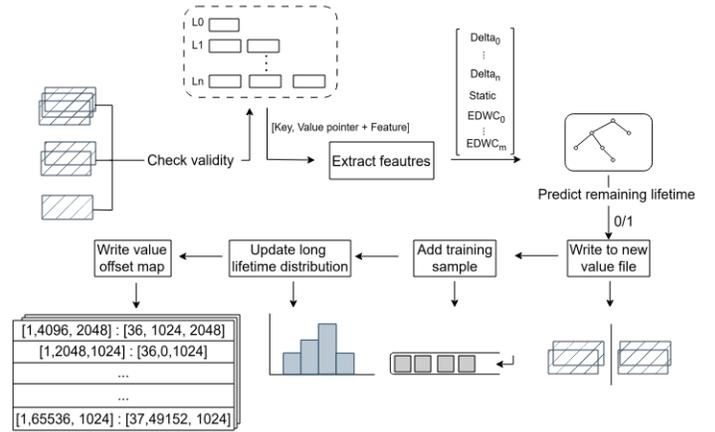


Figure 6: DumpKV garbage collection process

expected to yield a model output close to 0. Conversely, samples with a predicted remaining lifetime exceeding the short lifetime point are expected to yield a model output close to 1.

Loss function. DumpKV uses binary logloss as loss function to train GBM model. Binary logloss is a typical loss function used to train binary classification model.

Model training. Model training is started once training samples reaches threshold. Specifically, DumpKV creates a new GBM model and trains this new GBM model with new training dataset with specified loss function. After training is finished, the newly trained GBM model replaces the previous model and memory of the previous model is deallocated. Because this training process is done in separate thread and does not block flush operation or compaction process it brings small overhead to performance of storage engine.

4.7 Garbage collection

Garbage collection job is triggered once the lifetime TTL of value files is reached. Fig. 6 shows the garbage collection process in DumpKV. During the garbage collection job DumpKV checks the validity of each value by doing a search in LSM-tree. This search process can be fast by reading blocks from block cache because LSM-tree is small so that the majority of LSM-tree can be stored in block cache. If the value is invalidated and obsolete, then this value is discarded and no more extra steps are taken.

If the value is valid, DumpKV reads its feature data from the LSM-tree during validity checking, prepares Δ_{i0} by subtracting the KV pair's sequence number from the current timestamp, and updates each $EDWC_i$ with the expression mentioned in 4.2.1. DumpKV then uses the GBM model to predict the remaining lifetime and assigns the value to the appropriate lifetime category. It creates a training sample with the current feature data and adds it to the garbage collection training sample queue for future model training. Finally, DumpKV updates the long lifetime distribution histogram with Δ_{i0} .

After garbage collection job is finished, DumpKV recalculates default, short and long lifetime points l_d , l_s and l_l according to expressions \$4 and \$5 based on latest gc_ratio . Note that value files

with different lifetime thresholds can be put into the same garbage collection job.

During garbage collection, a value index map is created for each new value file. For example, a value from value file:1 at offset 4096 and size 2048 is rewritten to value file:36 at offset 1024. This map is necessary because the value index pointer in the LSM-tree remains unchanged during garbage collection. Future reads use this map to find the latest value file. The map is immutable and stored to prevent data loss. Keys in the map are in increasing order, allowing binary or hash searches if loaded in memory. The value pointer index in the LSM-tree is updated lazily during compaction, and the map is deleted when no older value files depend on it.

4.8 Crash consistency

DumpKV ensures crash consistency by storing key features in the value pointer index of the LSM-tree store engine. Model crash consistency is maintained by saving GBM model parameters to a file after each training session, allowing restoration upon engine restart. Training data is not persisted, as it is dynamically generated during engine operation. Garbage collection handles crash consistency similarly to compaction, without writing new value pointer indexes back to the LSM-tree, avoiding duplicate writes. If a crash occurs during garbage collection, incomplete value and index map files are discarded upon restart.

4.9 Implementation details

We implement DumpKV on top of RocksDB v8.00. DumpKV checks whether each value file reaches its lifetime limit after finishing each flush, compaction or garbage collection job. DumpKV implements a separate garbage collection module along with the existing compaction and flush module. Training dataset size is set to 256k by default. Compressed sparse row (CSR) format is used for model training and inference because Δ values for keys can be missing.

To serve read requests, DumpKV first does a query in LSM-tree to fetch the KV pair, which has a potential invalid value pointer. To get the latest value pointer for query key DumpKV searches value offset maps with the initial value pointer in LSM-tree in an iterative way. The latest value pointer for the query key is then obtained in the last value offset map. Then DumpKV reads value from value file with latest value pointer and returns value back to user.

5 EVALUATION

In this section, we evaluate and compare DumpKV with three state-of-the-art KV stores with KV separation design and implementation, which are RocksDB, TerakDB, and Titan. RocksDB has standard LSM-tree KV store implementation, which is referred to as RocksDB-std, and KV separation store implementation, which is referred to as RocksDB in the following evaluation part.

5.1 Experiment Setup

Testbed. We run experiments on multiple machines equipped with 40-core Intel Xeon Gold 6230N CPU, 196GB memory, and Intel PEDME016T4 1.5TB SSD, Ext4 filesystem, and Ubuntu 22.04 LTS.

Workloads. We use two sampled real-world workload traces called Systor [25, 26] and Tencent [65] and synthetic workloads generated by YCSB [8, 63] to test the performance of DumpKV in comparison

with heuristic baseline methods and other typical KV separation LSM-tree implementations. According to the actual experimental requirements, we generated zipfian-distributed workloads with a variety of skewness parameters, including 0.2, 0.5, 0.9, etc.

System configuration. For all KV stores, MemTable size is set to 100 MB, and value file size is set to 256MB. SSTable file size is set to $32 * 10 * \text{MemTable_size} / \text{value_size}$. For Titan, TerakDB, RocksDB, and DumpKV, the number of background garbage collection threads is set to 1. The maximum number of background compaction jobs is set to 16. Compression algorithms are disabled for all KV stores. Bloom filter is enabled for all KV stores, and the number of bloom filter bits is set to 10 according to the official tuning guideline. Block cache size is set to 16 GB and is used to cache blocks of SSTable only, and cache for blocks of value files is disabled. For Titan and TerakDB, the garbage collection triggering ratio is set to 0.2. For RocksDB, we set $\text{blob_garbage_collection_age_cutoff}$ to 0.8 and $\text{blob_garbage_collection_force_threshold}$ to 0.2. For DumpKV, α_s and α_l are set to 10. β_{s0} and β_{l0} set to 0.25, β_{s1} and β_{l1} are set to 0.75. For short lifetime point calculation, ini_sp_0 is set to 60 and ini_sp_1 is set to 40. The default lifetime point adjustment ini_dp_0 is set to 50 and ini_dp_1 is set to 20. For long lifetime point calculation, ini_lp_0 is set to 80 and ini_lp_1 is set to 20.

5.2 Real-world workload evaluation

In this section we conduct experiments with real-world datasets. We use workload trace from Systor and Tencent block storage services, which contain relatively large amounts of keys.

Baseline heuristic methods. We compare the model with 3 heuristic methods and see prediction performance between them. The first heuristic method is to use Δ_0 which is last lifetime for the same key as remaining lifetime indicator. A key is determined to have a short remaining lifetime if its last lifetime Δ_0 is shorter than short_lifetime . The second heuristic method uses the average value of all Δ_i as a remaining lifetime indicator. Similar to Δ_0 , if the average value of Δ_i of the same key is shorter than short_lifetime then the remaining lifetime of the key is determined to be short. The third heuristic method uses past write access count as the remaining lifetime indicator. A key is determined to have a short remaining lifetime if it has more than two writes before.

5.2.1 Offline evaluation. In this section, we compare performance results between heuristic methods and GBM model to see the prediction performance impact of feature engineering.

Dataset. The dataset we use for offline evaluation is generated from sampled Systor and Tencent trace datasets. Feature values of Δ and EDWC_s are generated offline. To simulate the online running process of the model, we filter records whose true lifetime is less than the short lifetime l_d and only apply the model to predict the remaining lifetime for writes of keys whose true lifetime is longer than l_d . For both workloads, the default lifetime l_d is set to the 60th percentile, short lifetime l_s to the 70th percentile, and long lifetime l_l to the 85th percentile of the lifetime CDF.

Metrics. Metrics we use for offline evaluation are precision, recall, and F1 score. Precision of positive samples affects total size, and

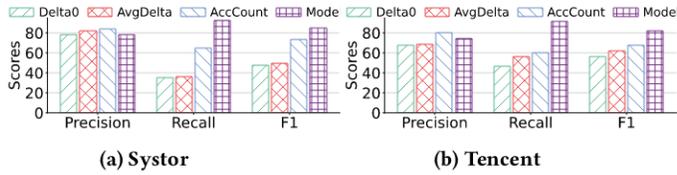


Figure 7: Offline evaluation metrics result among baseline heuristic methods and model for Systor and Tencent workload

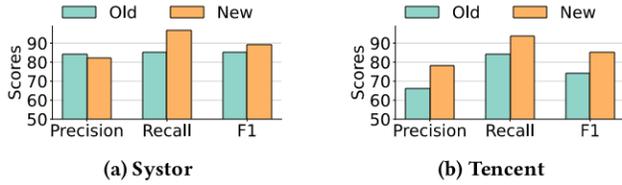


Figure 8: Offline evaluation metrics result from model trained from old and new data

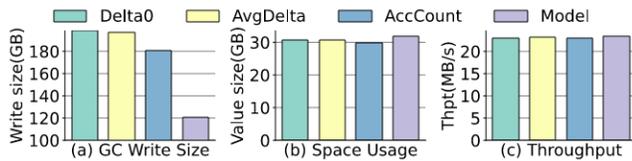


Figure 9: Metrics result comparison between heuristic methods and model for Systor workload

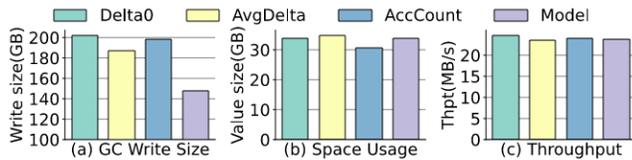


Figure 10: Metrics result comparison between heuristic methods and model for Tencent workload

recall of positive samples affects GC write size. The F1 score provides a single metric that balances both the concern of precision and recall.

Fig.7(a) and Fig.7(b) show offline metrics results for positive samples from first-time prediction for Systor and Tencent workload data. Precision scores for heuristic methods and the model are similar, which indicates that both heuristic methods and models have few false positive predictions. Recall scores of models for both Systor and Tencent are the highest and outperform baseline heuristic methods by more than 20 points. The model outperforms baseline heuristic methods, achieving an improvement of 12 to 37 points for Systor workload and 9 to 28 points for Tencent workload in F1 score. This indicates that the model has fewer false negative predictions, which means that it can relatively correctly predict long lifetime keys with dynamically adjusted EDWC and Delta features.

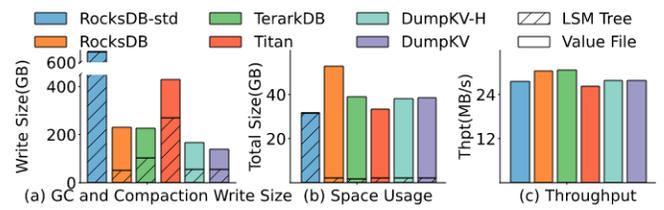


Figure 11: Online evaluation results between KV stores for Systor workload

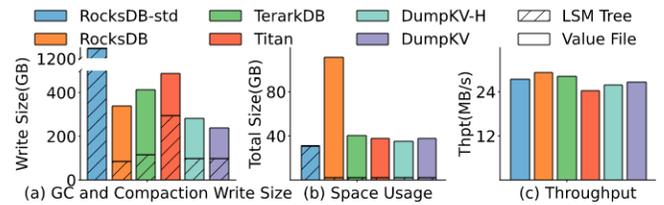


Figure 12: Online evaluation results between KV stores for Tencent workload

5.2.2 *Retraining effectiveness.* Fig.8 shows metrics results for models trained from the first 0-10% of dataset and 40%-50% of dataset to predict 50%-60% of dataset. A model trained with the latest data outperformed the recall scores by 12 and 10 points, and the F1 scores by 4 and 11 points for Systor and Tencent, respectively. This indicates that the write size can be reduced when the model is trained with newer data. The primary reason for this improvement is the cumulative updating of EDWC and Delta, which justifies the rationale for periodically retraining the model with the latest data.

5.2.3 *Model vs. Heuristic methods.* We next evaluate the prediction performance of heuristic methods and the model to see how each one affects the write size of KV stores. The lifetime threshold for value files is fixed to avoid impact from dynamic lifetime adjustment and get a fair comparison between the model and heuristic methods. Fig.9 and Fig.10 show GC write size, total value file size, and write throughput performance metrics for heuristic methods and the model. Compared to other 3 heuristic methods, the model reduces GC write size by 49%-64% and 26%-36% for Systor and Tencent workloads, respectively. The space usage of the model is similar to or slightly higher than that of heuristic methods because the model tends to bias towards providing long lifetime predictions for hard-to-predict samples to reduce write size. The throughput performance of both heuristic methods and the model is comparable, indicating that the introduction of model prediction has minimal impact on foreground write performance.

5.2.4 *Online evaluation.* We next evaluate how the model and heuristic methods perform with dynamic lifetime adjustment enabled compared to other baseline KV stores. We modify DumpKV with a heuristic method called DumpKV-H, which uses past write access count as a remaining lifetime indicator since it has the lowest GC write size among all 3 heuristic methods. Fig.11 and Fig.12 show GC and compaction write size, space usage of value files,

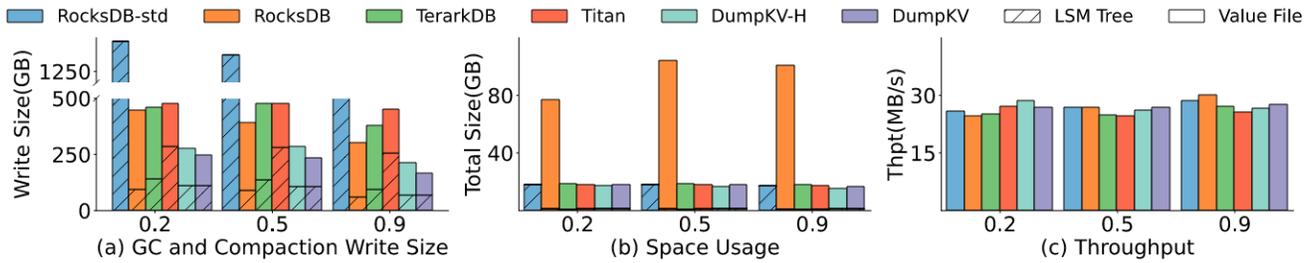


Figure 13: Performance comparison results for different KV stores in terms of total write size, total storage space size and throughput.

and throughput for all KV stores. DumpKV reduces GC write size by 31.0%, 46.4%, 52.1%, and 23.7% compared to TerakDB, Titan, RocksDB, and DumpKV-H for Systor workload. For Tencent workload, DumpKV reduces GC write size by 53.3%, 28.4%, 45%, and 25.7% compared to TerakDB, Titan, RocksDB, and DumpKV-H. The GC write size of both DumpKV-H and DumpKV is lower than that of other baseline KV stores, indicating that the proposed lifetime-aware garbage collection mechanisms effectively reduce GC write size. The gap in GC write size reduction between DumpKV-H and DumpKV decreases when dynamic lifetime adjustment is enabled for heuristic methods under high-skewed lifetime distribution workloads, such as Systor. This suggests that dynamic lifetime adjustment can help select appropriate lifetime values for value files based on the garbage collection ratio. DumpKV achieves a notable lower GC write size compared to DumpKV-H, demonstrating that the model has better prediction performance than naive heuristic methods. The total size of value files and throughput across all methods are similar.

5.3 Performance comparison

We evaluate and compare the performance of different KV stores under write-intensive workloads. Specifically, we generated three workload datasets using YCSB with different skewness parameters, which are 0.2, 0.5, and 0.9, and the lifetime distribution shows a similar long-tail distribution as that in Fig.3. Low skewness means that cold keys have more writes. Key size is set to 256 bytes, and value size is set to 4 KB. Each KV store is assumed to be empty initially. We load 200 GB of data into each KV store separately. Requests are issued to each KV store as fast as possible to do a stress test.

GC write size. Fig.13(a) shows GC and compaction write size from all KV stores. DumpKV shows the lowest total write size among all KV stores, which demonstrates that lifetime-aware garbage collection can significantly reduce write amplification by grouping large values with similar lifetime, thus achieving efficient garbage collection. Compared to RocksDB, DumpKV reduces GC write size by 57%-62%. Compared to TerakDB, DumpKV reduces GC write size by 58%-65%. Compared to Titan, DumpKV reduces GC write size by 30%-50%. Titan shows the second highest GC and compaction write size because it repeatedly writes KV pairs with the latest value pointer to LSM-tree during garbage collection. Compared to

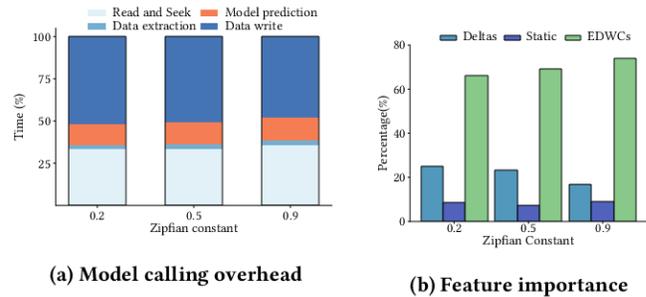


Figure 14: Model calling overhead for different skewness workload

DumpKV-H, DumpKV reduces GC write size by 19%-31%. RocksDB-std shows the highest GC and compaction write size because it repeatedly writes large values during the compaction process.

Total size. Fig.13(b) shows the total size of all KV stores after all write requests are issued. DumpKV achieves similar total size compared to RocksDB-std, TerakDB, and Titan. This shows that DumpKV can accurately build short and long lifetime thresholds based on workload and give relatively accurate predictions of the lifetime of keys. Noted that RocksDB struggles to achieve low total size because it does not check validity of keys by searching LSM-tree during garbage collection. And its garbage definition does not reflect the real number of invalidated keys.

Throughput. Fig.13(c) shows the throughput of all KV stores. Dumpkv achieves 25%-40% higher write rate compared to RocksDB because it does not need to write large values during compaction, which also includes garbage collection. RocksDB-std shows the highest write rate because it does not need to write KV pairs to SSTables and value files separately during the flush process.

Feature storage overhead. Fig.13(b) also shows LSM-tree size of all KV stores. Compared to RocksDB, DumpKV has 14%-18% LSM-tree size increase. This LSM-tree size increase only accounts for 0.05% total size increase, which is small overhead because large values take up the majority of storage space. Noted that LSM-tree size is small so that the whole LSM-tree can fit into the block cache entirely.

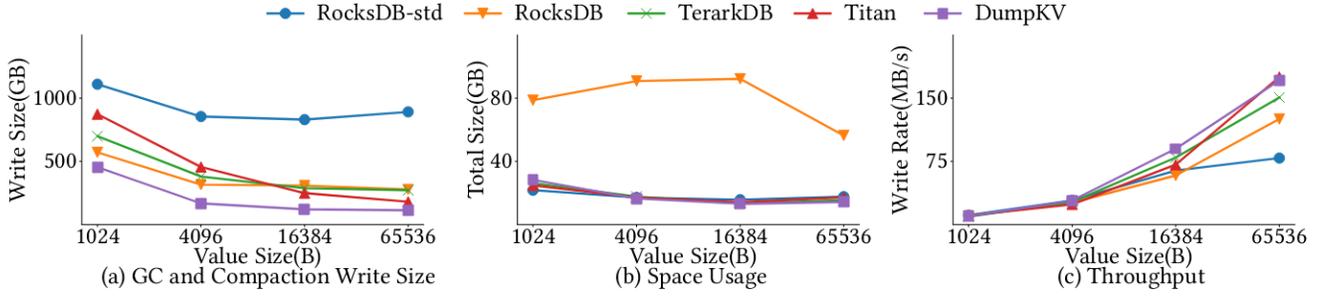


Figure 15: Performance of KV stores under different KV pair sizes

Table 3: Training dataset size and model calling overhead for workloads with different skewness

Skewness	Trn count	Dataset size	Model size	Avg call time	Avg Trn time
0.2	64	44 MB	115 KB	6.0 μ s	2 sec
0.5	60	82 MB	115 KB	6.0 μ s	2 sec
0.9	31	69 MB	115 KB	5.6 μ s	2 sec

5.4 Model overhead and feature importance

We analyze model training and inference overhead of DumpKV. Fig.14 shows model calling time during the garbage collection process of different workloads. It takes about 10% of total garbage collection time to do model calling to give remaining lifetime prediction to valid keys, which is a small overhead. DumpKV only invokes the model to give the remaining lifetime prediction if KV pairs in value files are still valid, which leads to less computation overhead.

Table 3 presents the model training count, average training dataset size, model size, average model calling time, and average training time for workloads with varying skewness. Less skewed workloads have higher training counts due to more balanced labeled datasets. The average dataset size ranges from 44 MB to 82 MB, representing a small memory overhead. Workloads with 0.2 skewness require the most memory due to greater average deltas. The GBM model’s memory footprint is only 115KB, a negligible overhead. Each round of model training takes around 2 seconds.

Fig.14(b) shows feature importance contribution to the overall model training process. Feature *EDWC* contributes 60%-70% to overall model performance gain, which is highest among all features. This shows that write access frequency information is most useful in capturing lifetime patterns of keys. The second highest contribution to model performance gain comes from *Deltas* which it takes up 16%-25%. This shows that past update distance information can be a useful feature for lifetime prediction as well. Static features contribute 7%-9% of overall model performance gain.

5.5 Dynamic lifetime adjustment

We next examine how dynamic lifetime adjustment in DumpKV helps determine short and long lifetime thresholds based on garbage collection invalid ratio. Figures 16 and 17 show changes in garbage

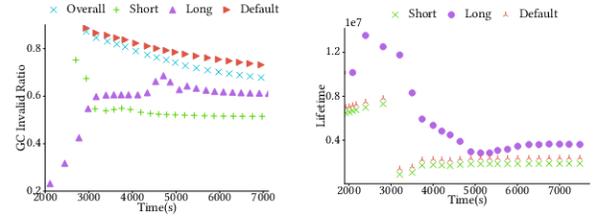


Figure 16: Dynamic lifetime adjustment and GC rate change for 0.2 skewness

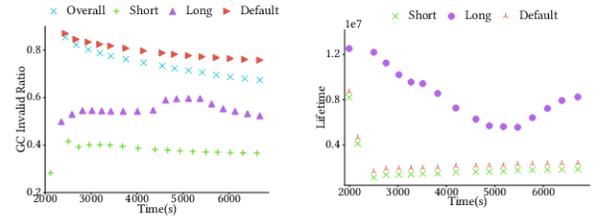


Figure 17: Dynamic lifetime adjustment and GC rate change for 0.9 skewness

collection invalid ratios and lifetime adjustments for default, short, and long lifetime value files. A higher invalid ratio indicates more invalid keys or garbage. DumpKV dynamically adjusts lifetime thresholds based on the invalid ratio. For a workload with 0.9 zipfian skewness, DumpKV sets a higher long lifetime threshold compared to a 0.2 skewness workload, demonstrating its ability to adapt to different workloads. Default lifetime value files have the highest invalid ratios (0.72 and 0.75), indicating most KV pairs are invalidated within the default lifetime window. Default and short lifetime thresholds show similar trends due to high invalid ratios.

5.6 Varying value size

We evaluate the performance of DumpKV for different value sizes. Specifically, we compare all KV stores with value sizes ranging from 1KB to 64KB. The total size of KV pairs loaded into all KV stores is 200 GB for all value sizes. Initial default lifetime for value files are set to 10% of total writes to accumulate training data for model training for DumpKV. Workload data is generated with YCSB and

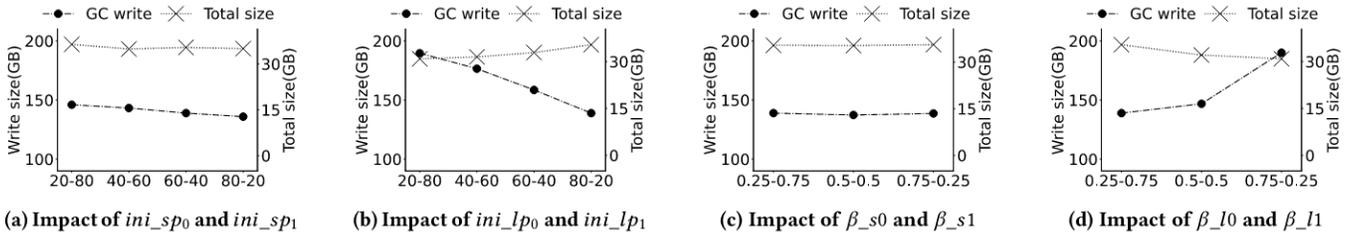


Figure 18: Empirical study on tunable parameters in DumpKV.

the zipfian constant is set to 0.99 for all value sizes. Fig.15 shows results of total write size and total storage size of all KV stores for varying sizes. DumpKV reduces GC write size by 20%-61%, 35%-58%, 37%-63% compared to RocksDB, TerakDB, and Titan.

When comparing total sizes, DumpKV exhibits a marginally larger size than both TerakDB and Titan, with an increase of 3.4% or 12% for value size 1KB, respectively. This mainly comes from LSM-tree storage size increase caused by extra feature storage overhead. RocksDB shows 64%-85% higher total size than DumpKV because it cannot reclaim storage space of invalidated KV pairs in value files in time due to the fact that it does not check the validity of KV pairs during garbage collection. DumpKV shows comparable throughput compared to other baseline KV stores for value sizes ranging from 1KB to 16KB.

5.7 Tunable Parameters

We analyze the sensitivity of parameters in DumpKV. Fig.18(a) shows the result of sensitivity for different combinations of ini_sp_0 and ini_sp_1 . The sum of ini_sp_0 and ini_sp_1 is 100 for all combinations, which means that *short_lifetime* range can be adjusted between 0-100% of the lifetime distribution histogram, which is the same for ini_lp_0 and ini_lp_1 . An increase of ini_sp_0 brings marginal GC write size reduction, which implies that lifetime distribution for short lifetime KV pairs is narrow. On the other hand, Fig.18(b) illustrates that the reduction in GC write size is more sensitive to increases in ini_lp_0 . Higher values of ini_lp_0 result in the lowest GC write sizes. This suggests that increasing the base lifetime for long lifetime value files helps to capture the long-tail distribution of long-lifetime key-value pairs. Fig.18 and Fig.18 show GC write size and total value file size with different midpoint shifts for the transformation function for short and long lifetime adjustment, respectively. Similar to ini_sp_0 and ini_lp_0 , reduction in GC write size is less sensitive to change of β_s0 and more sensitive to change of β_l0 . Lower β_l0 leads to a larger value for long lifetime value files, which leads to a lower GC write size. Note that total size of value files sees a slight increase as reduction of GC write size is increased because there is more garbage in long lifetime value files.

6 RELATED WORK

Garbage collection Lifetime information is used for garbage collection in fair amount of work in storage hardware like SSDs, HDDs, and file systems [21, 40, 53]. Several works [21, 45, 53, 59] improves IOPS and reduces garbage collection overhead in SSDs by identifying dominant I/O activities or inferring block invalidation time and

grouping blocks with similar lifetime. Other research focuses on reducing or avoiding garbage collection overhead [4, 7, 20, 29]. Some work suggests performing garbage collection during idle periods or preemptively [30, 39, 42].

LSM-tree based KV stores Numerous studies aim to enhance read/write performance in LSM-tree storage engines [2, 3, 56, 66]. NoveLSM [18] leverages non-volatile memories and techniques like byte-addressable skip lists to reduce latency. Several works [32, 62, 64] focus on improving system performance by tuning system configurations or restructuring storage structure. Other works focus on reducing write amplification [10, 44, 47].

Use of machine learning to improve system performance Machine learning models have been leveraged in storage systems [48, 50, 61]. LinnOS [15] uses a neural network to infer SSD performance per-I/O, improving I/O latency accuracy. GL-Cache [58] clusters objects for efficient learning and eviction, enhancing throughput. Machine learning is also used in query optimization [17], database tuning [31, 52], and learned indexes [11, 22, 23, 54]. Recent works use machine learning for cache prefetching or admission [33, 49, 55, 57].

7 CONCLUSION

In this paper, we introduce DumpKV, a learning-based, lifetime-aware garbage collection framework for KV separation in LSM-trees. DumpKV collects training samples from $L0 - L1$ compaction and garbage collection, leveraging the small size of LSM-trees for fast feature generation. It predicts key lifetime during garbage collection to balance throughput, write amplification, and total size, and dynamically adjusts lifetime thresholds for value files to achieve a low GC write size. Evaluation results show that DumpKV achieves the lowest GC write size compared to baseline KV stores for both real-world and synthetic workloads while maintaining comparable space usage and throughput.

ACKNOWLEDGMENTS

This work is supported by the National Key R&D Program of China under Grant NO. 2023YFB4502704, Guangdong Province Special Support Program for Cultivating High-Level Talents: 2021TQ06X160, National Natural Science Foundation of China (NSFC): 62272499 and 62332021, Pazhou Lab under Grant NO. PZL2023KF0001 and Research and Development Program of Shenzhen under Grant NO. 202407293000344.

REFERENCES

- [1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (London, England, UK) (SIGMETRICS '12). Association for Computing Machinery, New York, NY, USA, 53–64. <https://doi.org/10.1145/2254756.2254766>
- [2] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 363–375. <https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau>
- [3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. <https://www.usenix.org/conference/atc19/presentation/balmau>
- [4] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. 1995. Heuristic Cleaning Algorithms in Log-Structured File Systems. In *USENIX 1995 Technical Conference (USENIX 1995 Technical Conference)*. USENIX Association, New Orleans, LA. <https://www.usenix.org/conference/usenix-1995-technical-conference/heuristic-cleaning-algorithms-log-structured-file>
- [5] Bytedance. 2024. *TerakDB*. <https://github.com/bytedance/terakdb>
- [6] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 1007–1019. <https://www.usenix.org/conference/atc18/presentation/chan>
- [7] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. 2004. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.* 3, 4 (nov 2004), 837–863. <https://doi.org/10.1145/1027794.1027801>
- [8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) (SoCC '10). Association for Computing Machinery, New York, NY, USA, 143–154. <https://doi.org/10.1145/1807128.1807152>
- [9] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) (SOSP '17). Association for Computing Machinery, New York, NY, USA, 153–167. <https://doi.org/10.1145/3132747.3132772>
- [10] Niv Dayan and Stratos Idores. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) (SIGMOD '18). Association for Computing Machinery, New York, NY, USA, 505–520. <https://doi.org/10.1145/3183713.3196927>
- [11] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossman, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 969–984. <https://doi.org/10.1145/3318464.3389711>
- [12] Facebook. 2024. *RocksDB*. <https://github.com/facebook/rocksdb>
- [13] Jerome H. Friedman. 2001. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics* 29, 5 (2001), 1189 – 1232. <https://doi.org/10.1214/aos/1013203451>
- [14] Google. 2024. *LevelDB*. <https://github.com/google/leveldb>
- [15] Mingzhe Hao, Levent Toksoz, Nanqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. 2020. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 173–190. <https://www.usenix.org/conference/osdi20/presentation/haoh>
- [16] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liqian Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP database. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3072–3084. <https://doi.org/10.14778/3415478.3415535>
- [17] Alekh Jindal, Shi Qiao, Rathijit Sen, and Hiren Patel. 2021. Microlearner: A fine-grained Learning Optimizer for Big Data Workloads at Microsoft. In *2021 International Conference on Data Engineering*. IEEE, 2423–2434. <https://www.microsoft.com/en-us/research/publication/microlearner-a-fine-grained-learning-optimizer-for-big-data-workloads-at-microsoft/>
- [18] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 993–1005. <https://www.usenix.org/conference/atc18/presentation/kannan>
- [19] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf
- [20] Juwon Kim, Minsu Kim, Muhammad Danish Tehseen, Joontaek Oh, and Youjip Won. 2022. IPLFS: Log-Structured File System without Garbage Collection. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 739–754. <https://www.usenix.org/conference/atc22/presentation/kim-juwon>
- [21] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyool Lee, and Jihong Kim. 2019. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 295–308. <https://www.usenix.org/conference/fast19/presentation/kim-taejin>
- [22] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. arXiv:1911.13014 [cs.DB]
- [23] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System.
- [24] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jaidou Cong. 2015. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14. <https://doi.org/10.1109/MSST.2015.7208288>
- [25] Chungan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. 2016. Systor '17 Traces (SNIA IOTTA Trace Set 4928). In *SNIA IOTTA Trace Repository*, Geoff Kuenning (Ed.). Storage Networking Industry Association. <http://iota.snia.org/traces/block-io?only=4928>
- [26] Chungan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. 2017. Understanding Storage Traffic Characteristics on Enterprise Virtual Desktop Infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference* (Haifa, Israel) (SYSTOR '17). ACM, New York, NY, USA, Article 13, 11 pages.
- [27] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 273–286. <https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee>
- [28] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.* 50, 12 (2001), 1352–1361. <https://doi.org/10.1109/TC.2001.970573>
- [29] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, and Jongman Kim. 2013. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 2 (2013), 247–260. <https://doi.org/10.1109/TCAD.2012.2227479>
- [30] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. 2011. A semi-preemptive garbage collector for solid state drives. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. 12–21. <https://doi.org/10.1109/ISPASS.2011.5762711>
- [31] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: a query-aware database tuning system with deep reinforcement learning. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2118–2130. <https://doi.org/10.14778/3352063.3352129>
- [32] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 673–687. <https://www.usenix.org/conference/atc21/presentation/li-yongkun>
- [33] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. 2004. C-Miner: Mining Block Correlations in Storage Systems. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*. USENIX Association, San Francisco, CA. <https://www.usenix.org/conference/fast-04/c-miner-mining-block-correlations-storage-systems>
- [34] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, and Feng Sun. 2017. Model Ensemble for Click Prediction in Bing Search Ads. In *Proceedings of the 26th International Conference on World Wide Web Companion* (Perth, Australia) (WWW '17 Companion). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 689–698. <https://doi.org/10.1145/3041021.3054192>
- [35] Lanyue Lu, Thanumalayan Sankaranarayanan Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Wisckey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 133–148. <https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu>

- [36] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) (SIGMOD '18). Association for Computing Machinery, New York, NY, USA, 631–645. <https://doi.org/10.1145/3183713.3196908>
- [37] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 1–16. <https://www.usenix.org/conference/fast21/presentation/ma>
- [38] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) (ASPLOS '20). Association for Computing Machinery, New York, NY, USA, 541–556. <https://doi.org/10.1145/3373376.3378525>
- [39] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. 1997. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (Saint Malo, France) (SOSP '97). Association for Computing Machinery, New York, NY, USA, 238–251. <https://doi.org/10.1145/268998.266700>
- [40] Seonggyun Oh, Jeeyun Kim, Soyoung Han, Jaeho Kim, Sungjin Lee, and Sam H. Noh. 2024. MIDAS: Minimizing Write Amplification in Log-Structured Systems through Adaptive Group Number and Size Configuration. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. USENIX Association, Santa Clara, CA, 259–275. <https://www.usenix.org/conference/fast24/presentation/oh>
- [41] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.
- [42] Dongil Park, Seungyong Cheon, and Youjip Won. 2015. Suspend-aware segment cleaning in log-structured file system. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems* (Santa Clara, CA) (HotStorage'15). USENIX Association, USA, 17.
- [43] PingCAP. 2024. Titan. <https://github.com/tikv/titan>
- [44] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) (SOSP '17). Association for Computing Machinery, New York, NY, USA, 497–514. <https://doi.org/10.1145/3132747.3132765>
- [45] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2018. FStream: Managing Flash Streams in the File System. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 257–264. <https://www.usenix.org/conference/fast18/presentation/rho>
- [46] Matthew Richardson, Ewa Dominowska, and Robert Ragno. 2007. Predicting clicks: estimating the click-through rate for new ads. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) (WWW '07). Association for Computing Machinery, New York, NY, USA, 521–530. <https://doi.org/10.1145/1242572.1242643>
- [47] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building Workload-Independent Storage with VT-Trees. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX Association, San Jose, CA, 17–30. <https://www.usenix.org/conference/fast13/technical-sessions/presentation/shetty>
- [48] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. 2020. Learning Relaxed Belady for Content Distribution Network Caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 529–544. <https://www.usenix.org/conference/nsdi20/presentation/song>
- [49] Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza. 2008. Context-aware prefetching at the storage server. In *USENIX 2008 Annual Technical Conference* (Boston, Massachusetts) (ATC'08). USENIX Association, USA, 377–390.
- [50] Linpeng Tang, Qi Huang, Amit Puntambekar, Ymir Vigfusson, Wyatt Lloyd, and Kai Li. 2017. Popularity Prediction of Facebook Videos for Higher Quality Streaming. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 111–123. <https://www.usenix.org/conference/atc17/technical-sessions/presentation/tang>
- [51] Ilya Trofimov, Anna Kornetova, and Valery Topinskiy. 2012. Using boosted trees for click-through rate prediction for sponsored search. In *Proceedings of the Sixth International Workshop on Data Mining for Online Advertising and Internet Economy* (Beijing, China) (ADKDD '12). Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. <https://doi.org/10.1145/2351356.2351358>
- [52] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) (SIGMOD '17). Association for Computing Machinery, New York, NY, USA, 1009–1024. <https://doi.org/10.1145/3035918.3064029>
- [53] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. 2022. Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 429–444. <https://www.usenix.org/conference/fast22/presentation/wang>
- [54] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 117–135. <https://www.usenix.org/conference/osdi20/presentation/wei>
- [55] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2024. Baleen: ML Admission & Prefetching for Flash Caches. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. USENIX Association, Santa Clara, CA, 347–371. <https://www.usenix.org/conference/fast24/presentation/wong>
- [56] Peng Xu, Nannan Zhao, Jiguang Wan, Wei Liu, Shuning Chen, Yuanhui Zhou, Hadeel Albahar, Hanyang Liu, Liu Tang, and Zhihu Tan. 2022. Building a Fast and Efficient LSM-tree Store by Integrating Local Storage with Cloud Storage. *ACM Trans. Archit. Code Optim.* 19, 3, Article 37 (may 2022), 26 pages. <https://doi.org/10.1145/3527452>
- [57] Juncheng Yang, Reza Karimi, Trausti Saemundsson, Avani Wildani, and Ymir Vigfusson. 2017. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) (SoCC '17). Association for Computing Machinery, New York, NY, USA, 66–79. <https://doi.org/10.1145/3127479.3131210>
- [58] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. 2023. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 115–134. <https://www.usenix.org/conference/fast23/presentation/yang-juncheng>
- [59] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2017. AutoStream: automatic stream management for multi-streamed SSDs. In *Proceedings of the 10th ACM International Systems and Storage Conference* (Haifa, Israel) (SYSTOR '17). Association for Computing Machinery, New York, NY, USA, Article 3, 11 pages. <https://doi.org/10.1145/3078468.3078469>
- [60] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Trans. Storage* 17, 3, Article 17 (Aug. 2021), 35 pages. <https://doi.org/10.1145/3468521>
- [61] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: a learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proc. VLDB Endow.* 13, 12 (jul 2020), 1976–1989. <https://doi.org/10.14778/3407790.3407803>
- [62] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 17–31. <https://www.usenix.org/conference/atc20/presentation/yao>
- [63] YCSB-C. 2024. YCSB-C. <https://github.com/basicthinker/YCSB-C>
- [64] Jinghuan Yu, Sam H. Noh, Young ri Choi, and Chun Jason Xue. 2023. ADOC: Automatically Harmonizing Dataflow Between Components in Log-Structured Key-Value Stores for Improved Performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 65–80. <https://www.usenix.org/conference/fast23/presentation/you>
- [65] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. 2020. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 785–798. <https://www.usenix.org/conference/atc20/presentation/zhang-yu>
- [66] Yuanhui Zhou, Jian Zhou, Shuning Chen, Peng Xu, Peng Wu, Yanguang Wang, Xian Liu, Ling Zhan, and Jiguang Wan. 2023. Calcspar: A Contract-Aware LSM Tree for Cloud Storage with Low Latency Spikes. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 451–465. <https://www.usenix.org/conference/atc23/presentation/zhou>