Prophet: Optimizing LSM-Based Key-Value Store on ZNS SSDs with File Lifetime Prediction and Compaction Compensation

Gaoji Liu[†§], Chongzhuo Yang[†‡], Qiaolin Yu[†δ], Chang Guo[†], Wen Xia[§], Zhichao Cao[†]
[†]Arizona State University, [§]Harbin Institute of Technology,
[‡]University of Science and Technology of China, [δ]Cornell University

***Abstract*—In LSM-tree-based key-value stores (LSM-KV stores), the high write amplification (WA) in both LSM-KV store level and device level (device internal GC) caused by compaction seriously impacts the SSD lifespan. We believe Zoned Namespace (ZNS) SSD provides a good opportunity to address the lifespan issues since it can shift the data allocation and Garbage Collection (GC) capabilities to LSM-KV stores by using the *zoned block interface* management. Thus, LSM-KV stores can potentially reduce the device WA by minimizing the amount of data migration during zone cleaning. However, most of the existing studies fail to provide a comprehensive solution for LSM-KV stores on ZNS SSD to reduce device WA in the whole data lifecycles including file allocation, file GC, and compaction.**

In this paper, we propose Prophet, a WA-optimized LSM-KV store for ZNS SSDs with techniques of SST (Sorted Static Table) file lifetime prediction, zone-aware allocation, and WA-optimized zone cleaning. More specifically, 1) We first propose a novel clock to precisely measure the SST file lifetime via the sum of flush and compaction events; 2) We design a quantitative SST file lifetime prediction algorithm and the SST file allocation scheme to improve the invalid data ratio of cleaning zones such that the amount of data being migrated can be minimized; 3) We explore the possibility of integrating compaction and GC during zone cleaning and achieve a better trade-off between performance and WA. We implemented Prophet based on RocksDB and ZenFS, and evaluated the prototype on real ZNS SSDs. Our evaluations show that Prophet achieves 79% of SST file lifetime prediction accuracy and it successfully reduces up to 31% of WA compared to state-of-the-art LIZA in ZenFS.

***Index Terms*—ZNS SSDs, LSM-KV stores, write amplification**

I. Introduction

The log-structured merge tree (LSM-tree) [1]–[8] is a widely used data structure to organize data in KV-store by creating sequential writes in the Write Ahead Log (WAL) and immutable SST files. LSM-KV stores batch key-value pairs (KV-pairs) in memory and writes them out to the storage via Flush. KV-pairs are stored as immutable files, called *SST files*. SST files are organized in different levels (i.e., from Level-0, Level-1, to the bottom-most level) and the key-ranges of SST files in the same level do not have overlaps. The Compaction mechanism merges the one SST file from $Level - i$ with SST files in $Level - i + 1$ that have key-range overlaps into new SST files. After compaction, the newly generated SST files are allocated at $Level - i + 1$, and all the old SST files are deleted. Compaction effectively improves the space efficiency by eliminating the stale KV-pairs and improves the read performance.

However, compaction causes a high write amplification (WA) at LSM-KV store level since the KV-pairs are read and written multiple times by different compaction jobs until it is finally compacted to the bottom-most level. The high WA can explicitly reduce the SSD lifespan [9]–[13]. Moreover, the frequent SST file deletion causes a large amount of cleaning pressure on the SSD device's internal Garbage Collection (GC), which further increases the device's internal WA. Since LSM-KV stores have no control over the device level garbage collection applied by FTL in regular SSDs, optimizing the WA at the device level (i.e., reducing the data migration during GC) is difficult.

We believe Zoned Namespace SSD (ZNS SSD) provides a good opportunity to address the lifespan issues of LSM-KV store since it can shift the data allocation and cleaning capabilities to the application level by using the zoned block interface management. ZNS SSD divides the storage space into fixed-size zones (e.g., 512 MB or 1077 MB per zone). Similar to existing zoned block devices such as SMR [14]–[18] and IMR [19], data can only be appended sequentially on the zone write pointer, and in-place updates are not allowed. After resetting the write pointer to the beginning of the zone, the whole zone is cleaned and ready to accept new writes. Zone cleaning must be applied by the applications to identify and migrate the valid data from the cleaning zone such that we can ensure data correctness during the zone cleaning. In an ideal case, if one zone does not have any valid data before zone cleaning, there is no device-level WA. More importantly, fewer valid data can minimize the data migration efforts during cleaning and achieve higher performance, which is validated in the related work [20], [21].

Therefore, the main challenges of designing and optimizing LSM-KV store for ZNS SSD are: 1) to design an appropriate data allocation strategy such that most (or all) of the data in the same zone become invalid at about the same time and we can minimize the WA, and 2) to design a sophisticated zone cleaning scheme such that the zone selected for cleaning can achieve a better tradeoff between valid data migration overhead and space reclaiming efficiency. There are several existing studies that are optimizing LSM-KV stores on ZNS SSDs, which mainly focused on reducing the zone cleaning

overhead and device internal write WA [21]–[29]. In particular, existing studies like LIZA in ZenFS [21], [26], CAZA [27], LL Compaction [28], [29], ZNSKV [30], lifetimeKV [31], are focusing on adapting RocksDB (a widely used LSM-KV store) on ZNS SSDs and optimizing device level WA. However, there are several critical issues that are not fully explored and resolved, which hinder the application of existing studies.

First, precisely predicting the SST file deletion time is difficult. The previous studies [21], [26]–[28] have reached a consensus that we need to allocate SST files with a similar deletion time to the same zone. Consequently, when a zone is selected to be cleaned, most of the SST files are already deleted, which can minimize the cleaning overhead and WA. If we perfectly predict the deletion time of all SST files and allocate them to the zones with minimal deletion time variance, the cleaning zone can be reset as early as possible to minimize the data migration overhead. Since the creation time of each SST file is known, the key to the problem is to predict the SST file lifetime (i.e., the time from creation to deletion). But precisely predicting the lifetime time of a newly generated SST file is fundamentally challenging. The deletion of an SST is triggered when this file is compacted. However, compaction is determined by several factors: 1) Compaction policy (it determines the SST files to be compacted); 2) The LSM-KV store status, including total data size, levels, and level fan out; and 3) workload characteristics including key-range hotness, access patterns, and queries.

Second, how to effectively allocate SST files with a similar deletion time to a group of open zones is a complex issue. If we know the deletion time of all the SST files, grouping them into several zones to minimize the deletion time variation is feasible (i.e., a typical clustering algorithm can achieve optimal results). However, during the LSM-KV store running time, when one SST file is generated, we are not able to know the actual deletion time of allocated SST files in different zones and the deletion time of SST files being created in the near future. Therefore, there is only limited information available for designing the SST file allocation algorithm. Also, the number of open zones is also changing dynamically, which makes the problem even more complex.

Third, the tradeoffs between GC and compaction are not fully explored. There are two possible solutions for LSM-KV stores to clean one zone: 1) aggressively compact all the valid SST files in one zone such that all the SST files in this zone become invalid and we can directly reset the write pointer, and 2) apply traditional GC policy to migrate the valid SST files to other zones and then reset the write pointer. Some existing studies ([21], [26], [27], [32]–[35]) use GC to clean a zone while other studies ([28], [36]–[38]) redesigned the compaction policy to actively compact the valid SST files during the zone cleaning to achieve GC-free. Using compaction to delete all the SST files ahead of the zone reset can fully avoid the device level WA (i.e., no data migration is needed). However, it will increase the LSM-KV store level WA and may cause performance regression due to the extra unnecessary compaction jobs that are not triggered by the compaction policy itself.

To address the aforementioned critical issues and challenges, we propose ***Prophet***, a comprehensive optimization solution for LSM-KV stores on ZNS SSDs to achieve better tradeoffs between the SSD lifespan and overall performance. Prophet is a framework that includes the precise SST file lifetime prediction, deletion-time aware allocation, and GC with compaction compensation during zone cleaning. We first analyze the lifetime of SST files in different compaction scenarios and propose a complete lifetime prediction algorithm for each case, which is independent of the compaction policies. We innovatively propose to use the number of flush and compaction as logical clocks (FC-ticks) instead of natural time to achieve a more precise lifetime measurement. Second, to leverage the predicted lifetime of each newly generated SST file in the allocation, we propose the dynamic deletion-time range assignment scheme for each zone. We propose a way of estimating the deletion time range of each open zone and allocating the SST files to a zone that matches its predicted deletion time. We have considered the situation when there is no suitable zone to allocate the SST file and given a solution to make sure the deletion time distribution is similar in each zone. Finally, we analyze and compare the advantages and limitations of using GC and compaction during zone cleaning. We found the observations and insights that compaction can be a good compensation for GC when the valid SST files are expected to be compacted in the near future. Thus, we propose to dynamically apply GC and compaction for different SST files during the zone cleaning based on the compaction estimations. In this way, we can achieve the lowest device writes with minimal performance overhead compared with the full GC or full compaction schemes.

We implement Prophet based on RocksDB v7.6 and ZenFS v2.1.0, and open-sourced it at **Github**[1] for further investigations and research. We evaluate Prophet via db_bench in RocksDB on the real ZNS SSD prototype (WD ZN540 with 1TB capacity [39]). Our evaluation shows that Prophet can reduce up to 31% of WA compared to LIZA in ZenFS. Specifically, Prophet can achieve about 79% SST file lifetime prediction accuracy (the prediction error is smaller than 20 FC-ticks). Also, the GC with compaction compensation can further reduce about 7% to 10% of the device writes compared to the full GC or full compaction schemes. We summarize our contribution as follows:

- We propose to use the number of flush and compaction as the clock "Tick-Tock" to measure the SST file lifetime and design a detailed SST file lifetime analysis and prediction algorithms.
- We address the SST file allocation issue by dynamically assigning the deletion time range for each open zone. It successfully ensures a similar SST file deletion time distribution for each zone.
- We explore the tradeoffs between GC and compaction for zone cleaning, and propose compaction as compensation

[1]https://github.com/asu-idi/prophet-rocksdb

for GC. It avoids the potential extra I/Os caused by unnecessary compaction and further reduces device writes by about 7% to 10%.
- Prophet includes a complete solution to optimize LSM-KV stores on ZNS SSDs, which achieves up to 31% of WA reduction with no performance regression compared to LIZA in ZenFS.

II. Background and Related Work

A. LSM-based Key-Value Store

The log-structured merge tree (LSM-tree) [1] is a widely used data structure to organize data in KV-store suck as HBase [40], LevelDB [41], and RocksDB [42]. A noticeable feature is that LSM-tree writes data to the storage system sequentially, which is a preferred feature for ZNS SSDs. LSM-tree-based key-value stores (LSM-KV stores) caches key-value pairs in memory as Memtables and persist the Memtable in Sorted State Table Files (SST files) as tiered levels fashion on the storage system [43]. The data in Memtable will finally be written to Level 0 by flush operation.

Each level (except Level 0) contains a number of SST files that partition the key range of the level. The size limit of the level increases exponentially in a specific constant number (i.e., level fan-out). If the size of the level reaches the limit, compaction will be triggered to merge some of the SST files to the adjacent higher level (assume $level_{i+1}$ is the higher level of $level_i$ and $level_{i-1}$ is the lower level of $level_i$ in LSM-KV stores).

During one compaction job, a victim SST file in $level_i$ is selected based on the compaction policies. Then, all the SST files that have key-range overlaps with the selected file are selected at $level_{i+1}$. Finally, all those selected SST files are merged into new SST files and registered at $level_{i+1}$. Different LSM-KV stores may have different compaction policies, for example, LevelDB [41] supports Round_Robin compaction policy, while RocksDB [42] supports 5 different compaction policies (By_Compensated_Size, Oldest_Largest_Seq_First, OldestSmallestSeqFirst, Min_Overlapping_Ratio, Round_Robin).

Take the Round_Robin compaction policy as an example. Round_Robin compaction (RR-compaction) is a widely used compaction policy both implemented in RocksDB and Leveldb. In the RR-compaction design, each level maintains a compaction cursor that points to an SST file to be selected as the victim to start compaction, when $level_i$ triggers compaction. After that, the compaction cursor will move to the next position or the beginning (if it's at the end).

RocksDB is one of the most widely used and explored LSM-KV stores in both academia and industry. Therefore, in this paper, we are focusing on analyzing the compaction of RocksDB and optimizing RocksDB for ZNS SSDs. Considering the typicality and generality of RocksDB's design and implementation, the analysis and proposed scheme of this paper can be easily applied to other LSM-KV stores like LevelDB and HBase. RocksDB accesses on-disk data through its file system wrapper API which uses a unique identifier (e.g., a file name) that maps to a byte-addressable linear address space (e.g., a file). The identifier supports a set of operations such as random read/write, add, remove, and meta operations. ZenFS [21], [26] is an implementation of the file system wrapper to support end-to-end data placement onto zoned storage devices (e.g., ZNS SSD). ZenFS is a minimal on-disk file system, which follows the access constraints of zone-based interfaces and collaborates with device-side zone metadata on writes (e.g., write pointer) when placing data into the zone [44].

B. Zoned Namespace SSD

The NVMe Zoned Namespace (ZNS) is a new storage interface that divides the logical address space into a batch of zones. It can achieve better write performance and cost-effectiveness than conventional SSDs [21], [45]–[48]. ZNS SSDs eliminate the in-device garbage collection, over-provisioning, and some mappings by transferring the work from device FTL firmware to host software or applications.

A zone is the basic space management unit of ZNS SSDs. Compared with the conventional SSD which executes garbage collection by the internal FTL, ZNS allows the host to have control of data allocation, garbage collection, and over-provisioning. Different from the existing open-channel SSDs [49], [50], ZNS SSDs still have a simple FTL [51] to achieve the internal wear-leveling, performance isolation, and reliability guarantees, which makes ZNS SSDs easier to use and manage than the open-channel SSDs [52], [53].

However, applying the existing applications on ZNS SSDs also faces challenges. First, a non-empty zone only allows sequential writes [20] after erasing all the data in it by calling *RESET* command [54]. Thus, the host that manages the data has direct responsibilities for data allocation and zone cleaning before calling the command. Second, ZNS SSDs only accept sequential writes as they use the zones for storage and non-sequential writes violate zone operations. Most applications are not suitable for direct ZNS SSDs usage. Importantly, applications need to optimize the performance and device WA, which are missing in most of the current designs.

C. Related Work

There are several studies focusing on optimizing RocksDB on ZNS SSDs including LIZA in ZenFS [21], [26], CAZA [27], ZNSKV [30], lifetimeKV [31], and LL compaction [28]. All previous studies can be divided into two categories: the first is to modify the execution logic of the compaction to make the SST file deletion time on the same zone similar (e.g., lifetimeKV and LL compaction), the disadvantage of this approach is the need to modify compaction. However, previous studies have shown that compaction requires a trade-off among write amplification, space amplification, and read amplification. When optimizing write amplification by modifying compaction, it invariably increases space amplification. The second is to attempt to predict the lifetime of the SST file and allocate SST files with similar deletion times on the same zone (e.g., CAZA, LIZA, ZNSKV). However, their predictions are rough and have strict assumptions: LIZA assumes that

all SST files at the same level have the same lifetime, while ZNSKV simply calculates the lifetime by adding the level and the overlapping number together.

LIZA in ZenFS [21], [26] In the ZenFS implementation, it assumes that SST files at the same level have the same lifetime. SST files lifetime prediction based on four different file hotness hints (i.e., Short, Medium, Long, and Extreme) from RocksDB. Write-Ahead Log (WAL) and Manifest have the Short hint, files in level 0 or 1 have the Medium hint; files in level 2 have the Long hint, and the remaining files that reside in larger levels have the Extreme hint [55]. Each open zone is also assigned a lifetime hint which is equal to the first SST file allocated to the zone. In the allocation part, SST files with x lifetime hint will be allocated to the minimal lifetime hint zone that the hint is larger than x.

CAZA [27] CAZA algorithm finds that there are two inaccurate prediction phenomena in LIZA: 1) SST files having overlapping key ranges at adjacent levels of RocksDB can be placed in different zones and invalidated at the same time by compaction, and 2) compaction can invalidate SST files across zones with different lifetime hint values. To solve these two problems, CAZA proposed that files with key-range overlaps in adjacent two levels should be placed in the same zone first. Therefore, for SST file S which will be compacted to $level_i$, CAZA builds a set of SST file $S_{overlap}$ by searching $level_{i+1}$ files which have key-range overlap with S. Then, it also builds a zone set Z containing SST files from $S_{overlap}$ sorted in descending order by the zone remaining capacity and allocating S to Z in order. CAZA further proposes two Fallback mechanisms 1) allocate S to a zone that has the closest key with S, and 2) fall back to LIZA for the long-term segregation effect.

Lifetime-Leveling Compaction Algorithm [28] Lifetime-Leveling Compaction finds that in Round_Robin Compaction Policy, there are some SST files with an extremely long lifetime while others with an extremely short lifetime. Therefore, they modify the Compaction policy: 1) For the extremely long lifetime SST file, when compaction is triggered at lower levels, it will be forced to compact, regardless of whether it has an overlap with other SST files. 2) For the extremely short SST file, it will be written into two new SST files. One SST file has no overlap with the lower level, so it won't be compacted in a short time. Therefore, the extra compaction cost is only counted in another new SST file.

LifetimeKV [31] Similar to the LL Compaciton Algorithm, LifetimeKV implements range compaction to solve short live SST files and proposed Overlap Ratio Lifetime Victim SST Selection to prioritize long live SST files for compaction. With this new compaction policy, the lifetimes of SST files at the same level will be more similar to each other. Although it reduces write amplification, it increases the amount of GC Migrate data, requiring the use of parameter $alpha$ for further tradeoffs.

ZNSKV [30] ZNSKV attempted to use a quantitative method to study the lifetime of SST files, but their prediction method is more workload-dependent with low accuracy. They defined f to represent the current level and g to represent the overlapping number of the SST file with $leveli + 1$, using $f + g$ as the predicted lifetime of the SST file. However, in our analysis, we found that the situation regarding the deletion of SST files is more complex. Additionally, combining two unrelated variables as the predicted result for the lifetime prediction lacks a certain level of logical coherence.

D. Motivations

To reduce the WA of LSM-KV stores on ZNS SSD for a longer lifespan, the SST files with a similar deletion time should be allocated to the same zone. Therefore, a precise lifetime prediction algorithm is a must. At the same time, we should design a dynamic allocation algorithm to take into account the prediction error. The integration of GC and compaction is not fully explored, we expect to propose an algorithm to combine the GC and compaction to optimize WA without extra overhead. Previous studies have tried to solve these problems, but they have the following limitations:

- **Prediction**: None of the previous studies predicts SST file lifetime in a quantitative way. ZenFS only uses the levels to represent an SST file's lifetime. However, the lifetime distribution at the same level is not always similar. CAZA algorithm finds that files with overlaps in adjacent levels have a more similar lifetime, it also fails to make a clear estimation of the lifetime for each SST file due to the high complexity.
- **Allocation**: CAZA ignores the open zone limitations and does not have a dynamic adjustment for allocations when the lifetime prediction is not accurate. LIZA considers the hotness change as LSM-KV stores runs and tries to allocate the SST file to a zone with higher hotness. However, all of these studies do not have a quantitative separation of lifetime coverage between zones and adjust the allocation as the LSM-KV store state changes.
- **GC and compaction**: When space utilization is insufficient, the previous studies only discussed how to use GC-only or compactions-only schemes to optimize the throughput and WA. However, tradeoffs between them are not clearly explored. For example, aggressively applying compaction to delete the SST files in a zone to avoid the file migration overhead may cause extra SST file I/Os and CPU utilization, which can lead to an even shorter SSD lifespan and lower performance.

To address the aforementioned limitations and optimize WA for LSM-KV stores on ZNS SSDs, we propose Prophet, a comprehensive solution for optimizing LSM-KV stores on ZNS SSDs with a quantitative SST file lifetime prediction algorithm, a dynamic allocation algorithm, and a zone cleaning scheme with compaction compensated GC algorithm. The benefits of the Prophet can be summarized as follows:

- **Prediction**: 1) Prophet designed a physical clock FC-Ticks based on the properties of LSM-Tree to quantify the lifetime of SST files. 2) Prophet does not require modification of compaction, thus enabling it to reduce

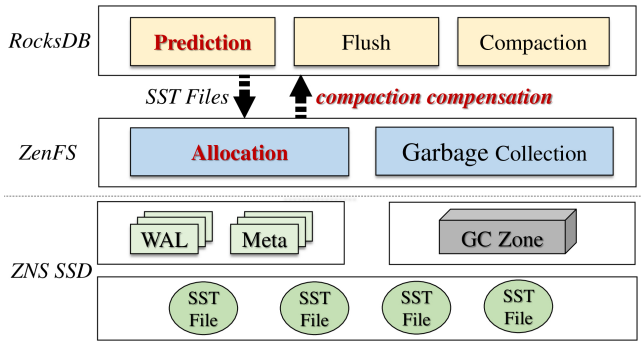


Fig. 1. Architecture of Prophet and its major components.

write amplification without any additional overhead. 3). Prophet can adapt to any existing compaction policy, such as Min_Overlapping_Ratio, Round_Robin, and others. 4). Prophet's predictions are designed based on the properties of the LSM tree rather than the variations in workload. Therefore, it demonstrates excellent optimization performance in both random workload and skewed workload.

- **Allocation**: We designed the allocation algorithm based on the prediction results from the LSM-tree and the characteristics of ZNS SSD. The elegance of the allocation algorithm lies in its isolation from prediction. This means that if users implement their own prediction algorithm, there is no need to modify the allocation algorithm.
- **GC and compaction**: The GC and compaction algorithm will determine whether an SST file needs to be GC or compacted based on the predicted results and the current timestamp, thereby further reducing write amplification. The GC and compaction algorithms are isolated from prediction and allocation.

III. SST File Lifetime Prediction

In this paper, we propose a comprehensive solution to optimize the WA of LSM-KV stores on ZNS SSDs. Specifically, we will use RocksDB [42] to analyze and evaluate our design and implementation. We propose to resolve the WA with a complete solution by combining SST file lifetime prediction, file allocation, and GC with compaction compensation. We will first present a quantitative and precise SST file lifetime prediction algorithm in this section. Then, the zone-based allocation and GC with compaction compensation will be discussed in Sections IV and V respectively. Figure 1 shows the architecture of Prophet and its major components including prediction, allocation, and compaction compensation in GC.

A. LSM-KV Store State Clock with FC-Tick

The previous studies do not answer the following questions clearly when they classify the SST files in different zones: 1) what is the appropriate clock to be used to define the lifetime of an SST file? 2) How precisely can we predict the SST file lifetime?

To answer the aforementioned questions and design a meaningful and feasible SST file lifetime prediction algorithm, we need to select an appropriate clock to measure the duration between SST file creation and deletion. The most straightforward idea and the widely used scheme in all related studies [27], [28], [30], [31] is to use the system default clock, which uses seconds or milliseconds to measure the time duration. However, LSM-KV stores state changing and SST file creation/deletion are irrelevant to the nature of time changes. Without the new KV-pair insertions (e.g., system idle time), flush and compactions will not be triggered no matter how long it is. Therefore, it is hard to make an accurate SST file lifetime prediction based on the nature time.

Since LSM-KV store state change is only related to the operations of flush and compaction, we novelly propose to use the event of flush and compaction as the new "Tick-Tock" of the clock. We define it as the **LSM-KV store state clock**, and the time unit is defined as an "Flush-Compaction Tick" (**FC-Tick**). Therefore, the "time" moment of an LSM-KV store can be defined as $time = flush\_number + compaction\_number$, which starts from LSM-KV store opening. There are two benefits of using LSM-KV store state clock in the SST file lifetime prediction: 1) LSM-KV store state will change when flush or compaction happens and the SST file lifetime is only related to LSM-KV store state change. Therefore, it can successfully filter out the LSM-KV store idle time, and we can better describe the duration than using the nature time; And 2) only these two operations can create or delete SST files. Flush creates SST files in Level 0 and compaction deletes some files in $level_i$ and $level_{i+1}$ and creates new files in $level_{i+1}$. **In the following, we will use the number of FC-Ticks to describe, analyze, and predict the lifetime of an SST file.**

B. Lifetime Prediction in Regular Compactions

To predict the lifetime of the SST file S with the time measurement of the LSM-KV store state clock, we first analyze the SST file deletion scenarios of S in regular compactions. In the special case, the trivial move compaction will be discussed separately (i.e., S does not have key-range overlap with the next level and it is directly registered in the next level without any actual I/Os).

We assume that the SST file S is in $level_i$ and its creation time is C_S. T is one of the SST files that have key-range overlap with S in $level_{i-1}$. There are two cases that an SST file will be compacted and deleted: **Case 1** (indicated as $c1$): S triggers compaction and it merges with some SST files in $level_{i+1}$ with key-range overlaps, and **Case 2** (indicated as $c2$): when T triggers compaction and S will be compacted passively due to the key-range overlaps with T. Suppose the predicted lifetime of SST file S in two cases are PL_S^{c1} and PL_S^{c2}.

For $c1$, we will model the problem by analyzing the process of the compaction, and simplify the model by using the properties of the LSM-tree. Then, we will give a simple formula to calculate PL_S^{c1}. For $c2$, we analyzed the distribution of real-life SST files and found that there are two sub-situations: short-lifetime and long-lifetime. We will analyze

the causes of this phenomenon and propose the prediction algorithm respectively. Finally, we will analyze the special case - trivial move compaction (indicated as $c3$). We propose a solution to identify and predict the SST file lifetime of PL_S^{c3} when $c3$ happens. In contrast, other existing studies just block the trivial move and fully ignore this important scenario.

Lifetime prediction of SST files that trigger compaction by themselves (Case 1) For SST files that actively trigger compaction, the timing of triggering compaction depends on its ranking in $level_i$ under a certain compaction policy being used. Therefore, we use $rank_S$ to indicate the order of the SST file that actively triggers compaction in $level_i$.

For example, in the case of the Round_Robin compaction policy, suppose the position of S in $level_i$ is y and CP_i is the compaction cursor of $level_i$. When S is created and the CP_i is at position x ($x < y$). Based on the RR-compaction, after $level_i$ triggers $y - x$ times compaction, CP_i will point to S, and S will be compacted and deleted. Therefore, we can infer that $rank_S$ equals the absolute difference between x and y, expressed as $rank_S = y - x$. After considering the case where $x >= y$, assume $num_i$is the file number of $level_i$, we will get the complete representation of $rank_S$ under the Round_Robin compaction policy as shown in formula 1. Under the kOldestSmallestSeqFirst compaction policy, $rank_S$ is equivalent to the index obtained by sorting all SST files in that level based on their smallest_seqno attribute.

$$rank_S = \begin{cases} y - x & x \leqslant y \\ num_i - (x - y) & x > y \end{cases} \tag{1}$$

Next, we need to figure out how many FC-Ticks it takes for $level_i$ trigger $rank_S$ compactions. We already know the Tick is the number of the flush and compaction in this duration. If we are able to figure out the compaction order between different levels and the compaction execution order in each level, we are able to estimate the lifetime of PL_S^{c1}.

Based on the compaction process in RocksDB, RocksDB will pick the first level which has a $score > 1$. The score is calculated by the predefined formula in RocksDB Compaction Picker $score(i) = \frac{Sum_i}{M_i}$. Here M_i is a constant number for each level which represents the maximal file number of $level_i$, and Sum_i is the total size of SST files in $level_i$. When $level_i$ triggers compaction, one SST file will be deleted in $level_i$ so $score_i$ will decrease and it will be smaller than 1 (i.e., once the $score_i > 1$, the compaction will be triggered immediately, so after the compaction, the $score_i$ will be smaller than 1). More files will be created in $level_{i+1}$, which increases its $score_{i+1}$. Consequently, if $score_{i+1}$ is very close to 1 before the new SST files are added in, there is a very high probability that $score_{i+1}$ will be greater than 1 after $level_i$ finishes its compaction. In this way, $level_{i+1}$ will be selected to apply compaction in the next run. Otherwise, if $score_{i+1}$ is much smaller than 1 and $score_{i+1}$ will still be smaller than 1 after the compaction. According to the structure of the LSM-tree, if $level_{i+1}$ is much smaller than the size limit, $level_{i+2}$ will have a very high probability that $score_{i+2} << 1$. Therefore, the higher levels ($level > i + 1$) have a very low probability to be selected to apply compaction.

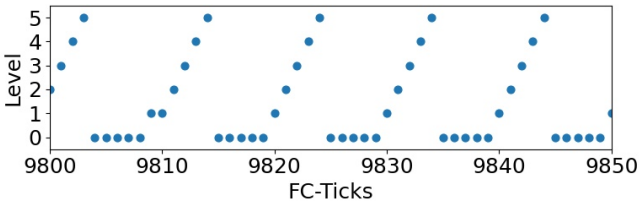


Fig. 2. Flush/compaction level distribution.

Note that, after $level_i$ finishes the compaction, the scores for $0 \leqslant level \leqslant i-1$ are also below 1. This is because if there is a j that $j < i$ and $score_j > 1$, RocksDB will first pick j as compaction level instead of selecting level i. Therefore, the next compaction will happen when the flush fully fills $level_0$ and the next compaction trigger level will be $level_0$. After $level_0$ triggers compaction, $level_1$ may trigger compaction and cause $score_2$ increases, and $level_2$ may trigger the compaction next time. Thus, we propose a conjecture: after $level_i$ triggers compaction, either $level_{i+1}$ triggers the compaction or waits for flush to fully fill $level_0$. Most likely, compaction will be triggered in order from $level_0$.

To verify our conjecture, we run RocksDB (default configurations) with RR-compaction under random workloads via db_bench (60 million insertions in total) and plot out the time and number of compactions in different levels under LSM-KV store state clock as shown in Figure 2. The X-axis is the FC-Tick (from 9800 to 9850 when $level_6$ is the highest level) and the Y-axis is the level that triggers the compaction. We can see that the distribution of flush and compaction exactly match our conjecture: After $level_i (i \in [0, 4])$ finishing the compaction job, $level_{i+1}$ will be selected to execute compaction because $level_1$ to $level_5$ is nearly full. After $level_5$ triggers compaction, $level_6$ will not trigger compaction because it is far away from full. At this moment, all the levels satisfy $score_i < 1$ and RocksDB will wait for flush to fill $level_0$ with newly generated SST files. After that, the compaction will be triggered from $level_0$ again.

Suppose max_level represents the highest level that is full, and $flush\_num$ is the number of SST files that are needed to fully fill $level_0$. We have $C = max\_level + flush\_num$. For example, in Figure 2, $C = 6 + 4 = 10$. We try to prove another conjecture: $\forall level_i (i \in [0, max\_level])$, The FC-Ticks costs between adjacent compactions are always the same as C. From Figure 2, we can see the distribution of flush and compaction is periodic and suppose $level_i$ triggers compaction at this moment. After $max\_level - i$ clock, the max_level will trigger compaction and after $flush\_num + i$ clock, $level_i$ will trigger compaction once again. The total time duration is $max\_level - i + flush\_num + i = max\_level + flush\_num = C$. Therefore, the time duration for $level_i$ to trigger compaction will likely be a constant number C.

Based on the analysis, we are able to estimate the number of FC-Ticks for $level_i$ to trigger $rank_S$ compactions. It will

cost $rank_S \times C$ FC-ticks, therefore, the predicted lifetime of SST file S in case $c1$ is: $PL_S^{c1} = C \times rank_S$

Lifetime prediction of SST files that are passively compacted (Case 2)

Predictions for $Case_2$ can be categorized into two scenarios: if there is an overlap between S and $level_{i-1}$, it may be a short-lived SST file; otherwise, it is a long-lived SST file. The concept of short-lived SST files has been previously proposed, yet prior studies consistently aimed to circumvent them by modifying compaction strategies [28]. In contrast, Prophet identifies and forecasts short-lived SST files non-intrusively. We delineate $Case_2$ into these two scenarios.

* Case 2A (indicated as $c2A$, and PL_S^{c2A} indicate the predicted lifetime of $c2A$): S compacted by lower-level with a long lifetime.
* Case 2B (indicated as $c2B$, and PL_S^{c2B} indicate the predicted lifetime of $c2B$): S compacted by lower-level with a short lifetime.

For $c2B$, the short-lived SST files have the same scenario: they have overlaps with the lower-level SST files when it was created. We explain it in an example shown in Figure 3. If SST file 1 in $level_i$ triggers compaction, files 1, 3, and 4 will be compacted and create files 6, 7, and 8. When the $level_i$ triggers compaction again, if SST file 2 triggers the compaction which deletes files 2, 5, and 8. SST file 8 will be deleted. According to the distribution of flush and compaction, the lifetime of those files will be about $rank_{SSTfile2} \times C$ (the time cost is SST file 2 triggers compaction). Therefore, suppose the file at $level_{i-1}$ which has overlap with S is T, we can estimate $PL_S^{c2B} = rank_T \times C$.

Predicting $c2A$ is challenging due to the scarcity of exploitable conditions. Therefore, we need to design the prediction algorithm from a higher-level perspective. Previous studies have indicated that in LSM trees, as the level of SST files increases, it signifies colder data. Additionally, for data at the same level, they exhibit similar lifetimes. For instance, while the LIZA algorithm assumes that SST files at the same level possess identical lifetimes (though our analysis reveals otherwise). For c2A prediction, Prophet follows a similar logic. As Prophet quantifies the lifetime of SST files, we utilize the average lifetime of all SST files that have been passively compacted in $level_i$ as the prediction for c2B. Specifically, when S is created, we get $Average_i$ which is the previous $c2A$ average lifetime in $level_i$. We have $PL_S^{c2A} = Average_i$. It should be noticed that if there is no $c2A$ before in $level_i$, we think that S will not be deleted in case $c2A$.

The final question is: which of the following predicted lifetimes (PL_S^{c1}, PL_S^{c2A} and PL_S^{c2B}) can be used as the final lifetime estimation of file S? Obviously, the true lifetime of SST file S depends on the minimum predicted value, therefore we choose $min(PL_S^{c1}, PL_S^{c2A}, PL_S^{c2B})$ as our final result.

C. Lifetime Prediction of Trivial Move

There is a special case that needs to be resolved separately: **SST file trivial move compactions**. If S triggers compaction and there is no SST file that has key-range overlap with S in $level_{i+1}$, S will be moved to $level_{i+1}$ logically in LSM-KV stores metadata (i.e., Manifest and Vstorage in RocksDB). But actually, the file continues to be maintained in the storage system. It will significantly increase the actual lifetime of S and cause the long time zone occupation in ZNS SSDs. We summarize it as **Case 3** ($c3$): S in $level_i$ triggers trivial move and moves to $level_{i+1}$, which will be deleted in $level_{i+1}$. Previous study LL algorithm [28] also encountered the same problem but they ignored it in their design and turned off it in evaluation and prototype. However, the trivial move can have a high ratio in RocksDB compactions (e.g., 20% based on our evaluation) and it has a very high impact on the WA.

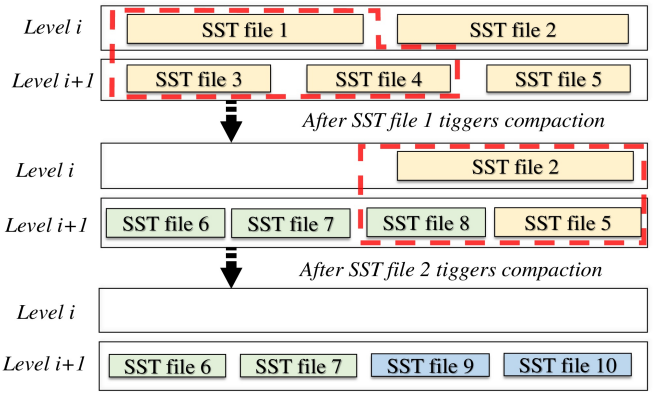


Fig. 3. The example of Case2B.

The first step is to identify the file S that will trigger the trivial move. There are two conditions: 1) S triggers compaction. In our prediction algorithm, we find that if $PL_S^{c1} < min(PL_S^{c2A}, PL_S^{c2B}$ then S will trigger compaction; and 2) S has no key-range overlap with files in $level_{i+1}$, which can be checked easily by comparing the key-ranges of SST files in this level. Suppose $trigger\_time(S)$ is the time duration between S creation time and the trivial move time, $compacted\_time(S)$ is the lifetime of S in $level_{i+1}$. We have $PL_S^{c3} = trigger\_time(S) + compacted\_time(S)$. The $trigger\_time(S)$ equals to PL_S^{c1} because the time of triggering trivial move and compaction is the same. When S is an SST file of $level_{i+1}$, it will be deleted either by compaction triggers from S ($c1$) or compacted by an SST file T that has overlap with it ($c2$) in $level_i$. Therefore its lifetime will be extended and more closer to SST files in $level_{i+1}$. The structure of LSM-tree will change when S is compacted and the prediction algorithm based on $rank_S$ in $c1$ will be inaccurate. Since The average lifetime of SST files in $level_{i+1}$ is stable, therefore we have the following estimation $compcated\_time(S) = Average_{i+1}$. Finally, we have $PL_S^{c3} = PL_S^{c1} + Average_{i+1}$. The lifetime prediction algorithm will run as Algorithm 1.

In general, the proposed SST file lifetime prediction schemes can have board impact and have been used in other LSM-KV store optimization directions like tiered storage [18], [56]–[58], performance optimizations for disaggregated storage [8], [59], and SST file caching [60].

Algorithm 1: SST file Lifetime Prediction.

Input: SST file S, level L_S
Output: predicted lifetime
Function `Predict`(S, L_S)**:**
  Lifetime $\leftarrow$ INF
  PL_1 $\leftarrow C \times Dis(S)$
  PL_2A $\leftarrow$ Average (L_S)
  if $L_S \neq 0$ *and has_overlap (S, L_S - 1) = true* **then**
    PL_2B $\leftarrow$ C
  end
  if *PL_1 ¡ PL_2A and has_overlap (S, L_S+1)=false* **then**
    PL_3 $\leftarrow$ PL_1 + Average (L_S+1)
    Lifetime = PL_3
  end
  else
    Lifetime $\leftarrow \min(PL\_1, PL\_2A, PL_B)$
  end
  return Lifetime;

IV. Deletion-Time Aware File Allocation

A. Allocation Algorithm Design

In Section III, we discussed the insight and algorithm to predict the lifetime PL_S of SST file S. We are able to calculate the predict the deletion time $PD_S = PL_S + C_S$, and C_S is the creation time of S. In this section, we will use the predicted deletion time PD_S to select an appropriate zone to allocate the file S such that the WA can be effectively optimized. The previous studies [21], [26]–[28] have the same optimization objective of the allocation algorithm: we should allocate the SST files that have similar deletion times to the same zone. However, they do not have a comprehensive allocation design to address all the possible scenarios.

Ideally, when we accumulate all the SST files and their predicted deletion time, we can easily classify them into N groups, and each group satisfies the zone size limitation and minimizes deletion deviations. However, in reality, we can only make the decision once the SST file is created, which is challenging and difficult since we are not able to know the deletion of SST files generated in the future. We propose to assign a deletion time range for each newly opened zone. We first define a zone Z's deletion time range as $[L_Z, R_Z]$. It indicates that the predicted deletion time of the SST files in this zone should be in the range of $[L_Z, R_Z]$. For example, if the deletion time range of a zone is [5, 8], it indicates that the smallest predicted deletion time is 5 and the largest predicted deletion time is 8. We can allocate S to the zone Z that satisfies $L_Z \leqslant PD_S \leqslant R_Z$. Therefore, we need to solve the following two problems: 1) How to estimate the L and R of one zone when the zone is opened. And 2) For SST file S, if we cannot find a suitable zone Z that satisfies $L_Z < PD_S < R_Z$, how to resolve this issue?

If $open\_zone\_number < max\_open\_zone\_number$ and we cannot find an open zone that satisfies $L_Z < PD_S < R_Z$, We will open a new zone for S and set L and R as its deletion time range. To make it simple, we set $L = PD_S$ to make sure that the deletion time range can include this newly generated SST file. For example, in Figure 4, when the first file with deletion time 5 is created, we open Zone1 and set $L_1 = 5$. It should be noted that the setting of L can still be optimized, which will be discussed in Section optimizationIV-B;

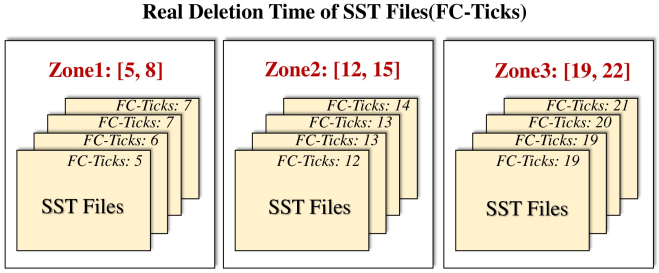


Fig. 4. Ideal deletion time distribution.

The estimation of R of the newly opened zone depends on the total SST file whose deletion time is in $[L, R]$ and the zone size. Suppose the size of SST file S is S_{size} (by default all the SST files have same size)and the size of zone Z is Z_{size}. $Num(L, R)$ is the number of SST files with deletion time in the $[L, R]$ range. In the ideal case, the SST files with $PD_S \in [L, R]$ should fully fill this zone. Therefore, R should meet the following equation:

$$Num(L, R) \times S_{size} = Z_{size} \tag{2}$$

Now, the problem is how to estimate $Num(L, R)$. $Num(L, R)$ depends on two values: 1) the compaction trigger number in $[L, R]$, and 2) the deleted file number of each compaction. For the compaction trigger number, we assume C_{rate} is the proportion of compaction in flush and compaction, therefore we can infer that $C_{rate} = \frac{compaction\_num}{compaction\_num + flush\_num}$ in time $[L, R]$. From compaction distribution analysis in Section III-B, we know that the flush/compaction distribution is periodic. The duration of the cycle is C and the compaction number is $max\_level + 1$ (max_level is the maximal full level). Therefore, the $C_{rate} = \frac{max\_level+1}{C}$.

We define D_{num} to represent the average deletion file number in each compaction (also means the average file number with the same deletion time). We can collect D_{num} through statistical data during LSM-KV store running time. Therefore, the $Num(L, R) = (R - L) \times C_{rate} \times D_{num}$ so that $(R - L) \times C_{rate} \times D_{num} \times S_{size} = Z_{size}$. Finally, we have $R = L + \frac{Z_{size}}{S_{size} \times C_{rate} \times D_{num}}$.

For example in Figure 4, each zone can hold up to 10 SST files, after L_1 is set to 5, R_1 should satisfy the number of the deletion time between L_1 and R_1 is 10 so that a zone can be exactly fully filled. It's $C_{rate} = \frac{6}{9}$ and $D_{num} = 4$.So we have $R = 8$ and there are 10 files that deletion time is in $[5, 8]$.

If $open\_zone\_number = max\_open\_zone\_number$, we will allocate S to an existing zone and violate the deletion time range requirement. In this case, for any open zone Z, we have either $PD_S < L_Z$ or $PD_S > R_Z$. We first try to

allocate it to one of the zones with $PD_S < L_Z$ because PD_S will not prolong the reset time of Z. It should be noticed that we should find the lowest L_Z in all zones that meet $PD_S < L$ to make sure that the deletion time distribution is similar. Otherwise, we need to allocate S to the remaining zones that satisfy $PD_S > R_Z$. We will also choose the zone with the largest R_Z to maintain the deletion time distribution similarly.

B. Allocation Optimizations

We can further optimize the aforementioned allocation algorithm with range rounding and level segregation.

Rounding the range $[L, R]$. As discussed in Section IV-A, we set the $L = PD_S$ at the beginning. If there are two SST files S_1 and S_2 with $PD_{S_1} = 15$, $PD_{S_2} = 13$, and $T = 10$(Suppose $T = \frac{Z_{size}}{S_{size} \times C_{rate} \times D_{num}}$), we will open a new zone Z_1 when allocating S_1 with $L_{Z_1} = 15, R_{Z_1} = 25$. However, we need to open another zone Z_2 with $L_{Z_2} = 13$ and $R_{Z_2} = 23$ for S_2. Therefore Z_1 and Z_2 have deletion time range overlaps and it causes space waste. To avoid this, we will round L and R as follows.

$$L = \lfloor \frac{PD_S}{T} \rfloor \times T \tag{3}$$

$$R = (\lfloor \frac{PD_S}{T} \rfloor + 1) \times T - 1 \tag{4}$$

After rounding, $L_{Z1} = 10, R_{Z1} = 19$ so $S1$ and $S2$ will allocate to the same zone.

Level Segregation. We find that for lower levels, the lifetime will be much smaller than those of higher levels. The low-level SST files will be quickly deleted after creation. Therefore, if we allocate these files together, then the zone will be reset soon. We call this zone a "short-lived" zone. Therefore, we set a SHORT_THRESHOLD and allocate the SST file which $level \leqslant SHORT\_THRESHOLD$ to the same zone. In section III-B, we use C to predict short-lifetime SST files, we will allocate these files to short-lived zone directly as well. SHORT_THRESHOLD needs to be determined through experiments. If the value is set too small, the level aggregation will not cover more files. If the value is too large, the file deletion time on the short live zone will be uneven.

We propose to design the level-based segregation: we should not allow higher-level SST files to be allocated to the short-lived zones even if their predicted deletion time satisfies the deletion time range of these short-lived zones. The reason is that SST files at higher levels usually have large deviations in deletion time prediction. There is a high probability that the higher-level file may prolong the reset time of the zone if it is allocated to a short-lived zone. For example, if there is a short-lived zone with a deletion time list [1, 2, 3, 4]. A high level S with the predicted deletion time = 5 is actually deleted at FC-ticks = 50 and it is allocated to this zone. Then the zone will not be reset until FC-ticks = 50. Allocating lower-level SST files to a zone that stores SST files from the higher levels are permitted since they will be deleted earlier and will not prolong the reset time of the zone.

We summarize the SST file allocation algorithm runs as follows (Suppose L_S is the level of S): 1) If $L_S < SHORT\_THRESHOLD$ or deletion type of S is $c2B$, allocate it to a short-lived zone (or open a new zone if there is no short-lived zone); 2) Try to allocate S to a zone Z suck that $L_Z < PD_S < R_Z$; 3) Open a new zone if $open\_zone\_number$ did not reach limit; 4) Try to allocate S to the zone with smallest L_Z with $PD_S < L_Z$; 5) Try to allocate S to the zone with largest R_Z which $PD_S > R_Z$.

V. GC with Compaction Compensation for Zone Cleaning

When we select one zone to apply zone cleaning, we need to take care of the valid SST files in this zone. First, in Prophet, we follow the space-oriented zone-selecting policy, which chooses the zone with the smallest valid data size to clean. This policy has been used in most of the existing studies [27], [28], [55]. After one zone has been selected, existing studies use either GC or compaction to remove the valid SST file from the zone to ensure data correctness. GC is an operation to migrate the valid data to another open zone before the original zone is reset, which is widely used in applications for zone-based storage [21], [26], [27]. The previous study LL algorithm [28] and GC-free design of GearDB for SMR [36] avoid GC via redesigning the compaction to actively delete all the valid SST files in the cleaning zone.

However, the performance and trade-offs between GC and compaction are not well explored. We believe, in some situations, applying GC can achieve a better performance than using compaction to actively delete the SST files with a very small WA overhead. However, if we are able to avoid triggering the unscheduled extra compactions, the active compaction design can effectively reduce WA with minimal performance overhead. Therefore, in this section, we explore the tradeoffs between GC and compaction and propose to use compaction as a compensation operation for GC in some cases to avoid valid data migration.

We conduct an experiment to compare the performance, I/Os, and compaction information of full GC and full active compaction schemes. We load 100GB of data to RocksDB on ZNS SSD. We find that: 1) active compaction causes the performance regression due to the resource contention with the foreground writes, 2) although active compaction achieves low device level WA, it actually causes 1.6X RocksDB I/Os than the full GC scheme due to the extra SST file reads and writes, and 3) there is a portion of the compaction jobs triggered by zone cleaning are the same compaction jobs between triggered in full GC scheme, which indicates that these compaction jobs are automatically scheduled and will not introduce extra I/Os and performance regression.

Ideally, when we actively compact the valid SST files in a cleaning zone, it has the same effect as executing the future scheduled compaction jobs in advance. Therefore, we do not trigger the extra RocksDB I/Os and introduce the extra performance overhead. However, since we do not know the relationship between the compaction job scheduling order and

the workloads, a GC-free scheme (i.e., active compaction) can easily trigger a large number of extra compaction jobs and bring in heavy SST file reads and writes. The actual total device I/Os can be even higher.

Based on the observations and analysis, we propose to use compaction as the compensation for GC to achieve a lower actual device I/Os than the full GC or GC-free schemes. We are targeting to achieve explicit smaller actual device writes than the full GC or active compaction schemes. However, it is hard to predict whether this compaction job will happen in the future or not. We find that a file S can be used for compaction compensation only if it will trigger the same compaction job in the near future. Therefore, during the zone cleaning, Prophet checks each valid SST file and applies compaction to delete the SST file if the following two conditions are satisfied: 1) S will be compacted as $c1$ (Case 1 discussed in Section III-B), and 2) $PD_S < MAX\_DELETION\_TIME$ (the maximal deletion time of the SST files that will be deleted). However, since $MAX\_DELETION\_TIME$ is determined by workload, we use the current FC-ticks T and PD_S to determine whether S will eventually trigger the same compaction job in the future. If $T \geq PD_S$, the lifetime prediction of S is incorrect and its real lifetime is unpredictable. Therefore, we will migrate S via GC. If $T < PD_S$, it indicates that S might be compacted in the future and we will compact S in advance during the cleaning. The detailed algorithm of GC with compaction compensation runs as follows:

1) Apply the Reset command if no valid data on the zone.
2) Compact the file S on the zone which the deletion type is $c1$ and $PD_S >$ FC-Ticks in advance.
3) Apply GC to the remaining SST file.

VI. Implementation and Evaluations

A. Prototype Implementation

We implement Prophet based on RocksDB v7.6 and ZenFS v2.1.0, and open-sourced it at [61], [62]. We collect SST file information in the RocksDB compaction jobs and predict the file lifetime before it is written to ZNS SSD. Note that we need to increase the size of the write buffer to ensure that the file will not be allocated to a zone before the largest key is decided (the largest key is needed to check the key-range overlap in prediction 1). We implement the zone allocation algorithm of Prophet in ZenFS. The total code change is 3879 lines added with 309 lines removed.

B. Experimental Setup

Since CAZA, LL, LifetimeKV, and ZNSKV algorithms are not open-sourced, we use LIZA in ZenFS as the baseline to compare with Prophet. All the experiments are carried out on a Dell server with Intel (R) Xeon (R) Silver 4210 CPU, 192GB memory, and the 1TB ZNS SSD prototype (WD ZN540) (100GB is reserved in evaluation with max_open_zone 14 and each zone is 1GB). We use db_bench, a benchmark tool released with RocksDB, to generate the workloads for RocksDB. The key size is 8B and the value size is 256B. We set the max_level0_size = 4, max_level1_size = 4, and the multiplier = 4. We set different SST file sizes for different experiments. To evaluate the prediction algorithm, we set the SST file size to 1MB to generate a large number of SST files to achieve a larger-scale evaluation. In the rest of the evaluations, we set the SST file size to 64MB and SHORT_THRESOLD=2 in Prophet to make the system evaluation more realistic. All the experiments are executed at least 3 times and we present the average results.

C. FC-Ticks Evaluations

In Section III-A, we discussed the advantages of FC-Ticks, the FC-Tick duration will not change when database writes are halted or idle. However, when using a physical nature clock (e.g., seconds) as the time unit, it continues to increment even when KV-pair write stops, leading to a big bias in the prediction of SST file lifetimes.

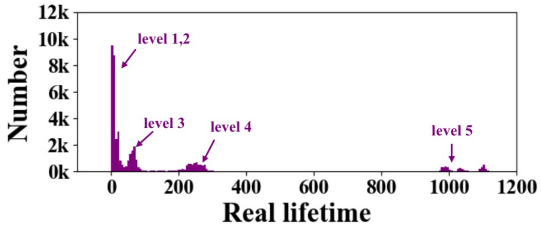
To validate the effectiveness of the proposed FC-Ticks, we modified the db_bench to achieve KV-pair ingestion rate variation over time based on the production workload characterized in [43]. We use Round_Robin compaction policy, random workload, and construct test data such as: at 0-2500s, the KV-pair ingestion rate is unlimited, at 2500-4500s, the KV-pair ingestion rate is set to 0 to simulated write stop or idle period, and at 4500-7000s, the KV-pair ingestion rate is recovered back to unlimited, which we call "VariableTest". In other test, we use random workload to continuously execute 5000s of KV-pair ingestion of unlimited rate, which we call "FixedTest". Under two types of tests, for FC-Ticks Prediction and Real-Time Prediction (Prediction using physical clock seconds), we separately calculated the SST file numbers predicted incorrectly due to write stopping which under the condition of $real\_lifetime > 3 \times predicted\_lifetime$ and $real\_lifetime > gap\_time$ (The reason for using 3 as the threshold is that when the deviation is greater, it is highly likely to be placed in the zone with a larger deletion time deviation). In Real-Time Prediction, gap_time is set to 2000s. In FC-Ticks Prediction, gap_time will be converted to the average number of FC-Ticks corresponding to the 2000s period under continuous write conditions. Table I shows the results of the evaluation, in the VariableTest, Real-Time Prediction would result in 152 wrong SST file predictions (i.e., at least 3 times larger actual time compared with predicted results). While, in FC-Ticks Prediction, there were no SST files predicted incorrectly in both tests.

TABLE I
FC-Ticks Evaluation

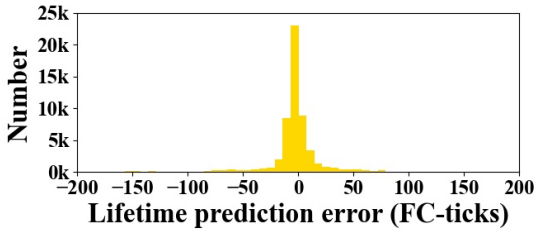
Number	VariableTest	FixedTest
FC-Ticks Prediction	0	0
Real-Time Prediction	152	0

D. Lifetime Prediction Analysis

In this experiment, 8GB of key-value pairs (KV-pairs) were loaded to the RocksDB instance randomly and 8,192 SST files were created in the loading process. After all the KV-pairs are inserted, 1388 SST files are alive in the database.



(a) Real lifetime distribution



(b) Lifetime prediction error distribution

Fig. 5. SST file lifetime prediction accuracy analysis.

Overall File Lifetime Prediction. We first analyze the lifetime prediction distribution and accuracy of all the SST files. Figure 5(a) shows the distribution of the real SST file lifetime in FC-ticks. The X-axis is the real lifetime in FC-ticks and the Y-axis is the number of files. There are three observations that align with our analysis in Section III: 1) the lifetime of most SST files is short, 2) SST files at a lower level usually have a shorter lifetime, and 3) the average lifetime of SST files in the same level is usually similar. For example, SST files at level 4 have a similar lifetime and the average is about 250. Differently, the average lifetime of SST files at Level 5 is about 1000.

Figure 5(b) shows the lifetime prediction error distribution (*predicted lifetime* minus *real lifetime*). The X-axis is the lifetime prediction error and the Y-axis is the number of files. In general, most of the predictions are accurate and 79% of the files have a prediction error smaller than 20 FC-ticks.

Prediction Type Analysis. To analyze the prediction accuracy of Prophet in detail, we print the accuracy in different types of compactions in $level_4$ as an example, since level 4 is large enough and full, which can cover all the cases. We set an error bar for each level to evaluate the accuracy. The error bar is different. In the lower level (Level 1 or 2), we can accommodate smaller prediction errors. Here, we set $errorbar = 5 \times (level + 1)$. For level 4, the error bar is 25, max_file_number is 256, and the average real lifetime is 194.

We measure the accuracy of each compaction case separately. Suppose D_x is the deletion type of SST file S. D_1 means S is deleted by the current level. Note that we use D_2 to include case $c2A$ or $c2B$. P_x is the predicted SST file deletion type for each case. For example, P_{2A} means we predict S will be deleted by $c2A$. $P_{2A} \cdot D_1$ means we predict the S will be compacted by the compaction triggered from the lower level in Case $c2A$. But actually, it's deleted by the compaction triggered in the current level in case $c1$.

TABLE II
LEVEL4 PREDICTION DISTRIBUTION.

Type	Accuracy	Number
$P_1 \cdot D_1$	99.2%	2890
$P_1 \cdot D_2$	5%	101
$P_{2A} \cdot D_1$	60.6%	493
$P_{2A} \cdot D_2$	67.6%	6695
$P_{2B} \cdot D_1$	23.1%	26
$P_{2B} \cdot D_2$	92.8%	1356
$P_3 \cdot D_1$	0%	5
$P_3 \cdot D_2$	22.9%	774
$Total$	74.0%	12340

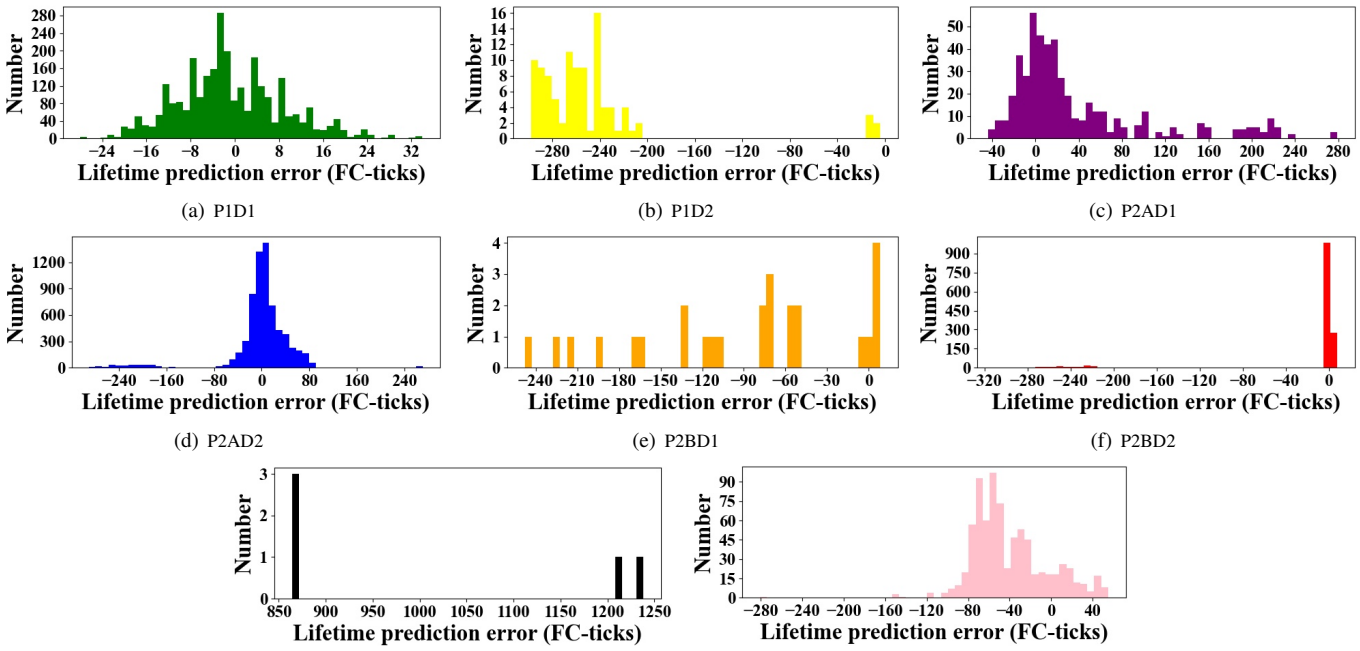
Table II shows the prediction accuracy for each type and the detailed prediction result distribution is shown in Figure 6. We can find that in most cases, we have a high accuracy and the prediction error is low. The detailed analysis is as follows: 1) $P_1 \cdot D_1$: we achieve almost 100% of accuracy prediction on Case $c1$; 2) $P_{2A} \cdot D_1$: it indicates that we mistakenly identified $c1$ as $c2$. So the predicted lifetime is larger than the real lifetime; 3) $P_{2A} \cdot D_2$: the error is caused by the trivial move prediction and the predicted lifetime is smaller than the real lifetime; 4) $P_{2B} \cdot D_2$: we have accurately identified most short-lifetime SST files (92.8%); 5) $P_3 \cdot D_2$: although the accuracy rate is very low, the distribution of results is acceptable, and most of them are correctly identified. We find the increasing of KV store lifetime also increases the $Average$, which leads to a low prediction accuracy of case $c3$. 6) Although $P_1 \cdot D_2$, $P_{2B} \cdot D_1$, $P_3 \cdot D_1$ has low accuracy, their number is small and will not have a huge impact.

In summary, Prophet can give a relatively accurate lifetime prediction result in most cases. SST files deleted by Case 1 and Case 2 compactions consist of 88.7% of the total file deletions and we archive about 79% of accuracy.

E. Compaction Compensation Analysis

To evaluate the effectiveness of the GC with compaction compensation, we set GC_START_LEVEL = 20% and different GC_STOP_LEVEL 30%, 35%, 40% and 45% in this evaluation (i.e., GC will start when the free space is less than 20% and stop when the free space is higher than GC_STOP_LEVEL). With a higher GC_STOP_LEVEL, more zones are selected to be cleaned. "Full GC" means we only use GC to migrate the valid SST files during the cleaning while "Full Compaction" only uses compaction to delete the valid SST files. "GC-CC" is the GC with compaction compensation in Prophet. In this experiment, we set the SST file size to 64MB and inserted 100 GB of data (the KV store is about 53GB at the end). We collected and compared the actual writes from RocksDB and into the ZNS SSDs.

Figure 7(a) shows the RocksDB writes of the three algorithms. The Full GC and GC-CC have similar RocksDB writes while the Full Compaction causes about 100GB more writes, which is caused by a large number of extra compaction jobs. Figure 7(b) and Figure 7(d) show the device writes on ZNS SSDs and the migrated data volume comparison respectively. The GC-CC algorithm can always achieve lower

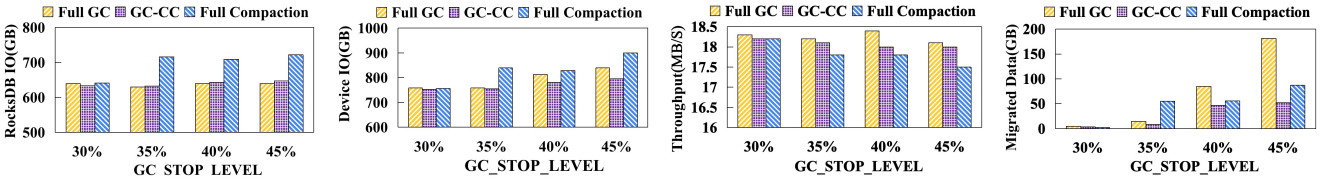


(a) P1D1 (b) P1D2 (c) P2AD1

(d) P2AD2 (e) P2BD1 (f) P2BD2

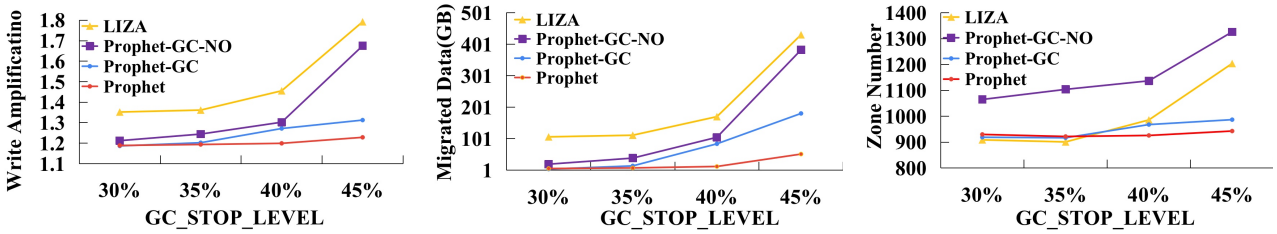
(g) P3D1 (h) P3D2

Fig. 6. The lifetime prediction error distribution analysis of different types.



(a) RocksDB I/Os (b) Device I/Os (c) Throughput (d) Migrated Data

Fig. 7. GC and Compaction Compensation analysis of 3 schemes on metrics of RocksDB I/Os, Device I/Os, etc.



(a) Write Amplification (b) Migrated Data (c) Allocated Zone Number

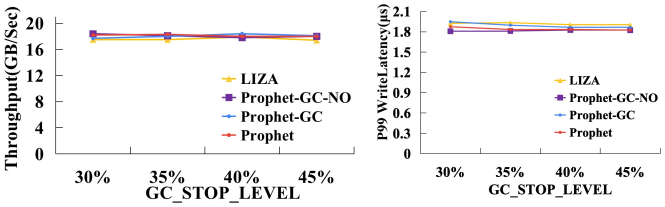
Fig. 8. Overall analysis of 4 schemes on metrics of WA, migrated data, allocated zone number.

device writes than Full Compaction and Full GC. The reason is that some SST files are compacted in advance and we avoid triggering the GC to migrate data. It should be noted that since Full Comparison did not reach the free space of GC_STOP_LEVEL before the end of the program, we still need GC to migrate data. Figure 7(c) shows the difference in throughput. The GC-CC has a tiny lower throughput than GC because we cannot guarantee that all the compaction jobs triggered during the zone cleaning are the compaction jobs that should be executed in the near future. Differently, full compaction has a much lower throughput. In general, GC with compaction compensation design successfully achieves the tradeoffs between GC and compaction, which effectively reduces the device writes and extends the SSD life span.

F. Overall Evaluations

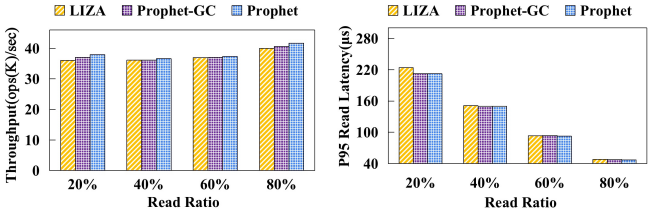
We evaluate Prophet and compare its WA ($WA = \frac{Device\ I/Os}{RocksDB\ I/Os}$) and performance with the other three baselines: 1) LIZA is the default algorithm in ZenFS, 2) Prophet-GC-NO is the scheme with lifetime prediction and allocation (using Full GC and excluding the allocation optimization, 3) Prophet-GC is the scheme that we disable the GC with compaction compensation in Prophet (only full GC is used).

Figure 8(a) shows the the WA of four schemes. Prophet-GC can always achieve lower WA than LIZA, and with the



(a) Throughput (b) P50 Latency

Fig. 9. Throughput and P50 latency Analysis



(a) ReadWrite Throughput (b) ReadWrite Latency

Fig. 10. ReadRandomWriteRandom Analysis.

increase of GC_STOP_LEVEL, the gap between the two schemes becomes larger. Figure 8(b) shows the amount of data being migrated by GC when GC_STOP_LEVEL is 30%, Prophet-GC only needs to migrate about 4GB of valid SST files, while LIZA copied about 100GB of data, which validated the effectiveness of prediction and allocation algorithm in Prophet. In Prophet, SST files with similar deletion times are gathered in the same zones. Therefore, when the zone is selected for cleaning, there are no valid SST files in the zone, and data migration is not needed. With the increase of GC_STOP_LEVEL, both algorithms need to migrate more data, but the amount of data that Prophet-GC needs to migrate is always lower than LIZA. In the extreme case of GC_STOP_LEVEL=45%, in order to reach the desired free space, more zones are selected to be cleaned and GC will migrate much more valid SST files, resulting in extremely high data migration and WA in LIZA. This problem has been well solved in Prophet-GC. Even in the extreme case, the migrated data amount and WA will increase steadily which is only 1.31 (about 26% lower than LIZA) while LIZA is 1.79.

Our two allocation optimizations proposed in Section IV-A have played a significant role in the reduction of WA and space utilization, especially in the GC_STOP_LEVEL is high. When the level segregation optimization is disabled, the WA of the Prophet-GC-NO is 1.67 and it is reduced to 1.31 in Prophet-GC. The rounding optimization effectively reduced space utilization. After we applied rounding optimization in Prophet-GC and Prophet, the total allocated zone number does not increase explicitly when GC_STOP_LEVEL changes. Differently, LIZA and Prophet-GC-NO increase the zone allocation number as GC_STOP_LEVEL increases.

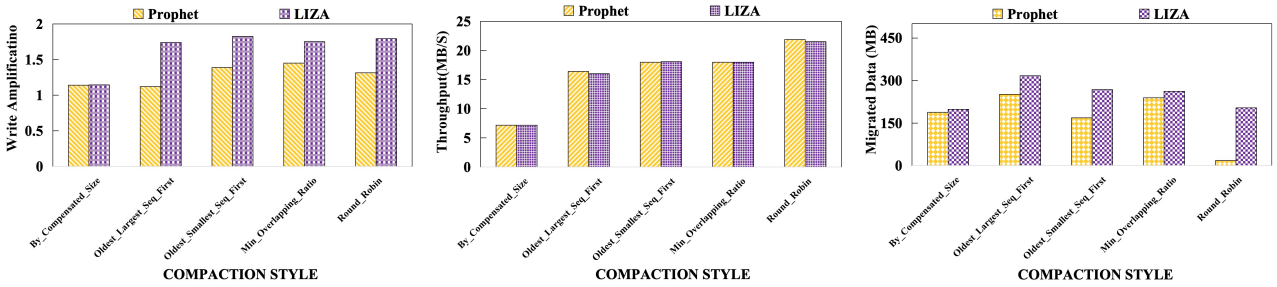
When we apply the GC with compression compensation in Prophet, it further reduces the WA. In GC_STOP_LEVEL=45%, the WA of Prophet is only about 1.22, which is 7% lower than Prophet-GC (WA = 1.31) and about 31% lower than LIZA (WA = 1.77). Importantly, applying the prediction, allocation, and GC with compaction does not cause the performance regressing shown in Figure 9(a) and Figure 9(b).

Throughput and Latency The throughput and latency of all four schemes are similar in different GC_STOP_LEVEL. The NVMe ZNS SSD prototype can achieve more than 1GB/s of sequential write and 2GB/s sequential read throughput, and four schemes all use one background thread to apply zone cleaning (Following the same approach as in ZenFS.). Therefore, zone cleaning with GC does not become a bottleneck for performance. This is consistent with the results obtained by CAZA [27]. LifetimeKV and ZNSKV claim to have achieved throughput improvements of 98% [31] and 32% [30], respectively. This is attributed to their synchronous garbage collection (GC) process, wherein RocksDB is unable to write during GC. This significantly enhances the impact of predictions. However, efficient GC should ideally be performed by background threads (e.g., ZenFS). As a result, there is no significant change in throughput observed during evaluation. We also evaluate Prophet with a read-write mixed workload. In this experiment, we fixed GC_STOP_LEVEL=45 and set the read ratio from 20% to 80%. Figure 10(b) and Figure 10(a) show the throughput and P95 latency of Prophet and LIZA under different read ratios. The throughput and latency of the two schemes are similar and the very slight differences are caused by the monitoring and prediction code paths in Prophet.

G. Evaluations with Various Compaction Policies

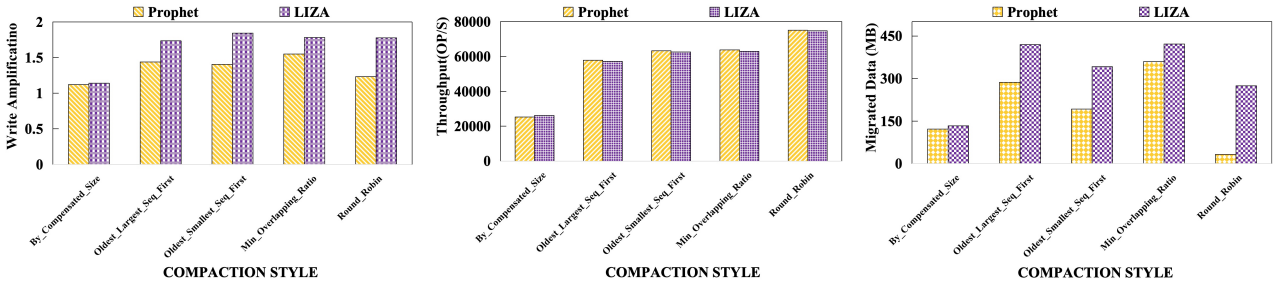
Since the Prophet prediction and allocation algorithm is independent of the specific compaction policies, it can operate under any existing compaction policy. However, it is important to note that if users define a new compaction policy, the corresponding rank calculation function needs to be added to Prophet. In most cases, this is a simple task, often requiring just a few lines of additional code. In our evaluation, we tested the write amplification of Prophet across the existing five compaction policies in RocksDB (By_Compensated_Size, Oldest_Largest_Seq_First, Oldest_Smallest_Seq_First, Min_Overlapping_Ratio, Round_Robin). To better reflect the gap between write amplification, we set GC_STOP_LEVEL=45%, and the remaining settings are the same as VI-F.

Figure 11(a) and Figure 11(c) show the write amplification and migrated data size with different compaction policies. In the Oldest_Largest_Seq_First and Round_Robin compaction policy, Prophet performs best, reducing write amplification by 35% and 31% compared to LIZA. However, under By_Compensated_Size, Prophet and LIZA performed similarly. This is because RocksDB executes more compactions, which makes SST files deleted more frequently. Therefore, zones can be reset earlier, leading to a low write amplification. However, the disadvantage is that this policy will lead to a much lower throughput.



(a) Write Amplification (b) Throughput (c) Migrated Data

Fig. 11. Random Write With Different Compaction Policy Analysis.



(a) Write Amplification (b) Throughput (c) Migrated Data

Fig. 12. Mixgraph Write With Different Compaction Policy Analysis.

H. Real Production Workload Evaluations

Prophet is designed based on the properties of LSM-Tree and it is independent of the specific workloads. Therefore, we tested its write amplification and throughput in production scenarios with hotspot KV range by Mixgraph [43]. Mixgraph is a Rocksdb testing framework that can generate KV-ranges with different access frequencies to simulate real-world workloads. In the benchmark, we set KV-range to 3, and the remaining parameters follow the default settings of mixgraph introduced in [43] appendix. The key and value size and other settings follow the configurations presented in Section VI-F.

Figure 12(a) and Figure 12(c) show the write amplification and migrated data size in Mixgraph benchmark with different compaction policies. In the mixgraph test, Round_Robin showed the best write amplification optimization of 30%, and similarly, Prophet is always better than LIZA.

VII. Conclusion and Future Work

In this paper, we addressed the WA issues of using RocksDB on ZNS SSDs without performance regression. We first proposed the flush and compaction event-based clock to precisely measure the SST file lifetime. Then, we proposed the SST file lifetime prediction and allocation algorithm to gather the SST files with similar deletion times in the same zone. Moreover, we optimized the WA by combining compaction with GC. Based on the evaluation, we achieve 79% lifetime prediction accuracy and 31% WA reduction compared to LIZA in ZenFS. In the future, we will attempt to improve the accuracy of prediction algorithms. We believe that machine learning will be beneficial in solving this problem, for instance, Reinforcement Learning (RL) or other self-improving models will be our future work.

Acknowledgements

We would like to thank our anonymous reviewers for their valuable feedback. We thank all the members of ASU-IDI Lab for providing useful comments. This work was partially funded by the Arizona State University startup fund.

References

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[2] C. Luo and M. J. Carey, "Lsm-based storage techniques: a survey," *The VLDB Journal*, vol. 29, no. 1, pp. 393–418, 2020.

[3] R. Sears and R. Ramakrishnan, "blsm: a general purpose log structured merge tree," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 217–228, 2012.

[4] B. C. Kuszmaul, "A comparison of fractal trees to log-structured merge (lsm) trees," *Tokutek White Paper*, 2014.

[5] H. T. Kassa, J. Akers, M. Ghosh, Z. Cao, V. Gogte, and R. Dreslinski, "Power-optimized deployment of key-value stores using storage class memory," *ACM Transactions on Storage (TOS)*, vol. 18, no. 2, pp. 1–26, 2022.

[6] H. T. Kassa, J. Akers, M. Ghosh, Z. Cao, V. Gogte, and R. G. Dreslinski, "Improving performance of flash based key-value stores using storage class memory as a volatile memory extension.," in *USENIX Annual Technical Conference*, pp. 821–837, 2021.

[7] Z. Cao, H. Dong, Y. Wei, S. Liu, and D. H. Du, "Is-hbase: An in-storage computing optimized hbase with i/o offloading and self-adaptive caching in compute-storage disaggregated infrastructure," *ACM Transactions on Storage (TOS)*, vol. 18, no. 2, pp. 1–42, 2022.

[8] J. Z. V. T. J. W. Qiaolin Yu, Chang Guo and Z. Cao, "Caas-lsm: Compaction-as-a-service for lsm-based key-value stores in storage disaggregated infrastructure.," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–26, 2024.

[9] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb.," in *CIDR*, vol. 3, p. 3, 2017.

[10] S. Sarkar and M. Athanassoulis, "Dissecting, designing, and optimizing lsm-based data stores," in *Proceedings of the 2022 International Conference on Management of Data*, pp. 2489–2497, 2022.

[11] M. Lim, J. Jung, and D. Shin, "Lsm-tree compaction acceleration using in-storage processing," in *2021 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pp. 1–3, IEEE, 2021.

[12] X. Sun, J. Yu, Z. Zhou, and C. J. Xue, "Fpga-based compaction engine for accelerating lsm-tree key-value stores," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 1261–1272, IEEE, 2020.

[13] B. Zhang and D. H. Du, "Nvlsm: A persistent memory key-value store using log-structured merge tree with accumulative compaction," *ACM Transactions on Storage (TOS)*, vol. 17, no. 3, pp. 1–26, 2021.

[14] A. Amer, J. Holliday, D. D. Long, E. L. Miller, J.-F. Pâris, and T. Schwarz, "Data management and layout for shingled magnetic recording," *IEEE Transactions on Magnetics*, vol. 47, no. 10, pp. 3691–3697, 2011.

[15] R. Pitchumani, J. Hughes, and E. L. Miller, "Smrdb: key-value data store for shingled magnetic recording disks," in *Proceedings of the 8th ACM International Systems and Storage Conference*, pp. 1–11, 2015.

[16] F. Wu, B. Li, Z. Cao, B. Zhang, M.-H. Yang, H. Wen, and D. H. Du, "Zonealloy: Elastic data and space management for hybrid smr drives," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[17] F. Wu, B. Li, B. Zhang, Z. Cao, J. Diehl, H. Wen, and D. H. Du, "Tracklace: Data management for interlaced magnetic recording," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 347–358, 2020.

[18] Z. Cao, H. Wen, F. Wu, and D. H. Du, "Smrts: A performance and cost-effectiveness optimized ssd-smr tiered file system with data deduplication," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pp. 275–282, IEEE, 2023.

[19] F. Wu, B. Zhang, Z. Cao, H. Wen, B. Li, J. Diehl, G. Wang, and D. H. Du, "Data management design for interlaced magnetic recording," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[20] M. Bjørling, "Zone append: A new way of writing to zoned storage," *Santa Clara, CA, February. USENIX Association.*, 2020.

[21] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. Le Moal, G. R. Ganger, and G. Amvrosiadis, "{ZNS}: Avoiding the block interface tax for flash-based {SSDs}," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 689–703, 2021.

[22] R. Liu, Z. Tan, Y. Shen, L. Long, and D. Liu, "Fair-zns: Enhancing fairness in zns ssds through self-balancing i/o scheduling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[23] M. Im, K. Kang, and H. Yeom, "Accelerating rocksdb for small-zone zns ssds by parallel i/o mechanism," in *Proceedings of the 23rd International Middleware Conference Industrial Track*, pp. 15–21, 2022.

[24] I. Song, M. Oh, B. S. J. Kim, S. Yoo, J. Lee, and J. Choi, "Confzns: A novel emulator for exploring design space of zns ssds," in *Proceedings of the 16th ACM International Conference on Systems and Storage*, pp. 71–82, 2023.

[25] C. Lee, D. Sim, J. Hwang, and S. Cho, "{F2FS}: A new file system for flash storage," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pp. 273–286, 2015.

[26] W. D. C. 2022, "Zenfs." https://github.com/westerndigitalcorporation/zenfs.

[27] H.-R. Lee, C.-G. Lee, S. Lee, and Y. Kim, "Compaction-aware zone allocation for lsm based key-value store on zns ssds," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pp. 93–99, 2022.

[28] J. Jung and D. Shin, "Lifetime-leveling lsm-tree compaction for zns ssd," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pp. 100–105, 2022.

[29] S. Sarkar, D. Staratzis, Z. Zhu, and M. Athanassoulis, "Constructing and analyzing the lsm compaction design space," *arXiv preprint arXiv:2202.04522*, 2022.

[30] D. Wu, B. Liu, W. Zhao, and W. Tong, "Znskv: Reducing data migration in lsmt-based kv stores on zns ssds," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pp. 411–414, IEEE, 2022.

[31] B. Liu, Y. Xia, X. Wei, and W. Tong, "Lifetimekv: Narrowing the lifetime gap of ssts in lsmt-based kv stores for zns ssds," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pp. 300–307, IEEE, 2023.

[32] G. Choi, K. Lee, M. Oh, J. Choi, J. Jhin, and Y. Oh, "A new lsm-style garbage collection scheme for zns ssds," in *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[33] K. Han, H. Gwak, D. Shin, and J. Hwang, "Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 147–162, 2021.

[34] G. Oh, J. Yang, and S. Ahn, "Efficient key-value data placement for zns ssd," *Applied Sciences*, vol. 11, no. 24, p. 11842, 2021.

[35] D. R. Purandare, P. Wilcox, H. Litz, and S. Finkelstein, "Append is near: Log-based data management on zns ssds," in *12th Annual Conference on Innovative Data Systems Research (CIDR'22)*., 2022.

[36] T. Yao, J. Wan, P. Huang, Y. Zhang, Z. Liu, C. Xie, and X. He, "{GearDB}: A {GC-free}{Key-Value} store on {HM-SMR} drives with gear compaction," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pp. 159–171, 2019.

[37] A. Manzanares, N. Watkins, C. Guyot, D. LeMoal, C. Maltzahn, and Z. Bandic, "Zea, a data management approach for smr," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[38] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds," *ACM Transactions on Storage (TOS)*, vol. 13, no. 3, pp. 1–26, 2017.

[39] W. D. 2020, "Ultrastar dc zn540 from western digital." https://www.westerndigital.com/products/internal-drives/data-center-drives/ultrastar-dc-zn540-nvme-ssd#0TS2094.

[40] A. H. 2008, "Hbase." https://hbase.apache.org/.

[41] G. 2012, "Leveldb." https://github.com/google/leveldb.

[42] F. 2013, "Rocksdb." http://rocksdb.org/.

[43] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pp. 209–223, 2020.

[44] W. D. 2022, "Rocksdb with zenfs." https://zonedstorage.io/docs/applications/zenfs.

[45] H. Shin, M. Oh, G. Choi, and J. Choi, "Exploring performance characteristics of zns ssds: Observation and implication," in *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pp. 1–5, IEEE, 2020.

[46] T. Stavrinos, D. S. Berger, E. Katz-Bassett, and W. Lloyd, "Don't be a blockhead: zoned namespaces make work on conventional ssds obsolete," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 144–151, 2021.

[47] J. Min, C. Zhao, M. Liu, and A. Krishnamurthy, "{eZNS}: An elastic zoned namespace for commodity {ZNS}{SSDs}," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 461–477, 2023.

[48] W. Qi, Z. Tan, J. Shao, L. Yang, and Y. Xiao, "Indef: An advanced defragmenter supporting migration offloading on zns ssd," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pp. 307–314, IEEE, 2022.

[49] I. L. Picoli, N. Hedam, P. Bonnet, and P. Tözün, "Open-channel ssd (what is it good for).," in *CIDR*, 2020.

[50] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proceedings of the Ninth European Conference on Computer Systems*, pp. 1–14, 2014.

[51] H. H. 2021, "dm-zap: Host-based ftl for zns ssds." https://github.com/westerndigitalcorporation/dm-zap.

[52] M. Bjørling, "From open-channel ssds to zoned namespaces," in *Proc. Linux Storage Filesyst. Conf.(Vault)*, vol. 1, 2019.

[53] W. M. 2018, "Spdk open-channel ssd ftl." https://spdk.io/doc/ftl.html.

[54] N. E. 2022, "Zoned namespace command set specification.." https://nvmexpress.org/wp-content/uploads/NVM-Zoned-Namespace-Command-Set-Specification-1.1b-2022.01.05-Ratified.pdf.

[55] H. H. 2020, "Zenfs, zones and rocksdb - who likes to take out the garbage anyway?." https://snia.org/sites/default/files/SDC/2020/074-Holmberg-ZenFS-Zones-and-RocksDB.pdf.

[56] "Jay zhuang. time-aware tiered storage in rocksdb https://rocksdb.org/blog/2022/11/09/time-aware-tiered-storage.html," Accessed 10 Oct, 2023.
[57] Z. Cao, H. Wen, X. Ge, J. Ma, J. Diehl, and D. H. Du, "Tddfs: A tier-aware data deduplication-based file system," *ACM Transactions on Storage (TOS)*, vol. 15, no. 1, pp. 1–26, 2019.
[58] X. Ge, Z. Cao, D. H. Du, P. Ganesan, and D. Hahn, "Hintstor: A framework to study i/o hints in heterogeneous storage," *ACM Transactions on Storage (TOS)*, vol. 18, no. 2, pp. 1–24, 2022.
[59] S. Dong, S. S. P, S. Pan, A. Ananthabhotla, D. Ekambaram, A. Sharma, S. Dayal, N. V. Parikh, Y. Jin, A. Kim, *et al.*, "Disaggregating rocksdb: A production experience," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–24, 2023.
[60] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Evolution of development priorities in key-value stores serving large-scale applications: The rocksdb experience.," in *FAST*, pp. 33–49, 2021.
[61] P. 2023, "Prophet-rocksdb." https://github.com/Flappybird11101001/prophet-rocksdb.
[62] P. 2023, "Prophet-zenfs." https://github.com/Flappybird11101001/prophet-zenfs.