SA-LSM: Optimize Data Layout for LSM-tree Based Storage using Survival Analysis

Teng Zhang[†*], Jian Tan[†], Xin Cai[†], Jianying Wang[†], Feifei Li[†], Jianling Sun[*]
Alibaba Group[†]
Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Zhejiang University[*]
{jason.zt,j.tan,frank.cx,beilou.wjy,lifeifei}@alibaba-inc.com,sunjl@zju.edu.cn

ABSTRACT

A significant fraction of data in cloud storage is rarely accessed, referred to as *cold data*. Accurately identifying and efficiently managing cold data on cost-effective storages is one of the major challenges for cloud providers, which balances between reducing the cost and improving the system performance. To this end, we propose *SA-LSM* to use (S)urvival (A)nalysis for Log-Structure Merge Tree (LSM-tree) key-value (KV) stores. Conventionally, the data layout of LSM-tree is determined jointly by the write and the compaction operations. However, this process by default does not fully utilize the access information of data records, leading to a suboptimal data layout that negatively impacts the system performance. *SA-LSM* utilizes the survival analysis, a statistical learning algorithm commonly used in biostatistics, to optimize the data layout.

When put into perspective of LSM-tree with proper adoptions, *SA-LSM* can accurately predict cold data using the historical semantic information and access traces. As a concrete realization, we implement our proposal in X-Engine, a commercial-strength open-source LSM-tree storage engine. To make the deployment more flexible, we also design a non-intrusive architecture that offloads CPU-intensive work, e.g., model training and inference, to an external service. Extensive experiments on real-world workloads show that it can decrease the tail latency by up to 78.9% compared to the state-of-the-art techniques. The generality of this approach and the significant performance improvement show great potentials in a variety of related applications.

PVLDB Reference Format:
Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, Jianling Sun. *SA-LSM*: Optimize Data Layout for LSM-tree Based Storage using Survival Analysis. PVLDB, 15(10): 2161 - 2174, 2022.
doi:10.14778/3547305.3547320

1 INTRODUCTION

The amount of data keeps growing at an unprecedented rate [5]. Driven by the trend to host data services on cloud platforms, reducing the storage cost on cloud databases has become one of the primary challenges for cloud vendors. This challenge is more pronounced for OLTP databases that are more sensitive to latencies since they often serve mission-critical tasks such as online e-commerce, instant messaging and real-time gaming. Figure 1 reports the prices of three typical Elastic Container Service (ECS) configurations at Alibaba Cloud. As can been seen, depending on the storage types, the lower-latency media can even account for the lion's share of the total cost, especially in presence of the ever growing data volume.

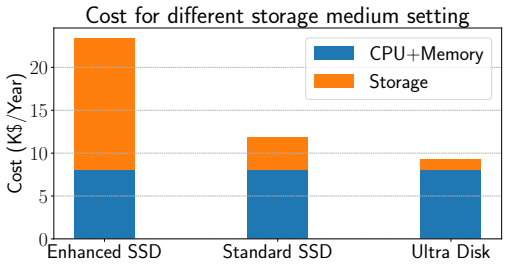


Figure 1: Cost to host a database instance for 1 year using different storage media on Alibaba Cloud ECS *ecs.hfg6.8xlarge* instance with 2 TiB storage [1]; Enhanced SSD (ESSD) storage cost can be more than 2× the cost of CPU and memory

To reduce the storage cost, LSM-tree has become an increasingly popular architecture. It introduces multiple layers for heterogeneous storages [8, 46, 59], where upper layers are mapped to fast storages, e.g., Solid State Drive (SSD), and lower layers to slow storages, e.g., Hard Disk Drive (HDD). One key component therein is the compaction strategy that determines how to dynamically allocate and move data records across different layers.

Conventionally, the data layout of LSM-tree is determined jointly by the write and the compaction operations, where a background compaction operation periodically merges the data records onto the persistent storage to ameliorate the read/write and space amplifications [42]. However, this process by default does not fully utilize the access information of data records, leading to a suboptimal data layout that negatively impacts the system performance. This effect is even more pronounced as different LSM layers are mapped to heterogeneous storages. To this end, we design *SA-LSM* to enhance the compaction strategy, based on survival analysis [32], a statistical learning algorithm commonly used in biostatistics, to optimize the data layout in this process.

We compare the performance using the default compaction strategy and *SA-LSM*, by replaying a 3-day representative workload of Alibaba e-commerce business. The heterogeneous storages are configured with layers L_0 and L_1 residing on ESSD and layer L_2 on HDD. To compare with the performance limit, we build an ideal baseline using the homogeneous fast ESSD storages for all three layers. As shown in Figure 2, the heterogeneous storage is

Proceedings of the VLDB Endowment, Vol. 15, No. 10 ISSN 2150-8097.
doi:10.14778/3547305.3547320

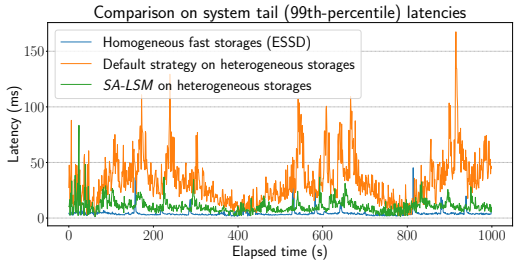


Figure 2: Compaction strategy severely impacts the system performance of LSM-tree; the default strategy drastically deteriorates the performance while *SA-LSM* significantly narrows the gap between the heterogeneous storages and the ideal baseline using only homogeneous fast storages; heterogeneous storages are cost efficient but only effective if cold and hot data can be well separated

dramatically slower than the homogeneous one under the default compaction strategy. Specifically, the average latency of the former is 130% higher than that of the latter, with the 99-th percentile latency being even 7 times longer. Notably, after applying *SA-LSM*, the system performance with heterogeneous storage is almost close to the best achievable limit using homogeneous storages, as it can better restructure the data layout.

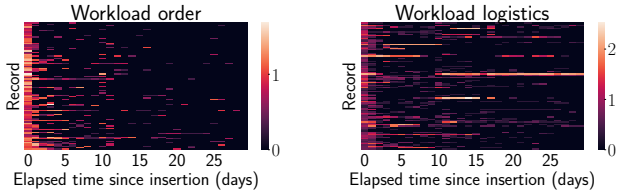
Motivation. To reduce operational cost, it is important to optimize the data layout on different LSM-tree layers. A typical three-tier storage hierarchy uses SSD/DRAM to build a low-latency tier, SATA HDD as a high-density capacity tier, and tape libraries as a low-cost archival tier [50]. To improve the system performance, accurately identifying cold data is critical in optimizing the compaction strategy. However, most existing solutions are relatively simple on a coarse granularity without fully utilizing the information from the collected traces, which limit the system performance. For example, Cassandra [35] relies on a round-robin strategy and RocksDB [19] uses a random strategy. Interestingly, traditional cache eviction policies, e.g., LRU, have been employed for compactions by Mutant [59] and PrismDB [46]. Although other strategies with various heuristics could be applied [30, 41], we argue that a rigorous analytic framework that fully utilizes the historical semantic information and access traces can further improve the system performance in a more robust manner.

But why survival analysis? Firstly, for many data intensive applications, caching systems are presented with a challenge that the popularities and statistics of the data items are time-varying, with a significant fraction becoming cold over time and thus very rarely accessed. Also, new data are dynamically generated, often with time-to-live constraints. In these cases, even though the observed training data seem abundant, in fact a small data challenge arises in a big data setting. Specifically, only the most recent, and thus relatively scarce, data access information is used to train the model, which requires to be updated periodically. Secondly, due to the dynamic and heterogeneous access patterns of different data tables, we find that maintaining a dedicated and simple survival model for each table is better than a single and complex model shared by all tables. The details are presented in Subsection 4.2.2. Since a model is only for one table, the training data are split among the models, which are further reduced. Lastly, the time series of the observations naturally incur right censoring [32], which means future events are known to be beyond an observation window but are unknown by how much. Survival analysis can properly address this challenge; see Figure 14 for the fractions of the censored data in real workloads. In Section 4.2.3, we conduct ablation studies by varying the observation window sizes to change the fraction of censored data, which demonstrate the superior performance of *SA-LSM* for time-varying, dynamic, and heterogeneous caching applications.

Metrics. To measure the quality of cold data identification and evaluate the system performance for different algorithms, we introduce the cold data false identification rate (*cold_fir*) based on LSM-tree storage characteristics and also adopt the commonly used c-index [32] in the survival analysis literature. The former (*cold_fir*)) computes the fraction of the misclassified cold data with respect to the total amount of data to be migrated to a cold storage tier, which is formally defined in equation (6). The latter (c-index) is used to evaluate the quality of a survival model, which is formally defined in equation (5).

Challenges. To identify cold data and optimize their layout in an LSM-tree storage present four challenges.

Granularity to separate cold and hot data: Most existing solutions [8, 35, 59] separate cold and hot data using an SSTable as a unit, which cannot distinguish individual data records in the same SSTable. Figure 3 demonstrates that the data access frequencies within the same SSTable change over time at a record granularity. Therefore, simply compacting the whole data on an SSTable can incur unexpected accesses to the cold tiers since some hot data records may be scattered therein.



(a) Heat map for *order* **(b) Heat map for *logistics***

Figure 3: Popularities (access frequencies) for individual data records (defined by the number of accesses per day) for two real-world workloads (*order* and *logistics*); each row represents a record evolving over time with a color depicting the access frequencies on a logarithm scale; the heat map discloses a dynamic and complex access pattern in an SSTable

Censored data due to an observation window: The key to identify cold data is to infer the next access time based on the past access events. There is an important concept borrowed from survival analysis on censored time-to-event data, since the data accesses may not exist within the observation window. Without properly handling these censored events, inaccuracy and bias could be incurred in the prediction. In this regard, survival analysis is a convenient tool to address this issue. To cast the cold data prediction as a survival model is not necessarily a straightforward task. We carefully design the feature vector from the recurrent access traces, define the events to avoid unnecessary noise, provide the associated labels

for the censored data, and construct the training data by random sampling to avoid bias. Note that we train a dedicated model for each individual workload, i.e., the requests for the same data table, on a refined granularity.

Non-intrusive compaction service deployment: Unlike traditional eviction policies, e.g., LRU, LFU and LIRS, that are light-weight, a compaction strategy based on computation intensive learning algorithms could consume large CPU/memory resources and incur contentions. To this end, we design and implement a non-intrusive method to offload the model training and inference work to an external service. This external service also brings more opportunities and flexibility, e.g., testing other advanced learning algorithms and jointly utilizing the traces from multiple similar workloads.

Targeting workloads: For modern applications, e.g., e-commerce, instant messaging and user generated content (UGC) [53], the popularities of the data records often keep decreasing over time. We call these workloads **a**rchival **w**rite and **p**oint read **i**ntensive (AWPI), as an extension to WPI workload [16]. The characteristics of AWPI workloads make LSM-tree on heterogeneous storage an ideal candidate to archive cold data. To this end, we implement an detection algorithm on the external service, which can adaptively enable *SA-LSM* to proactively conduct compactions depending on whether the workload is AWPI or not.

***Summary of contributions*:**

- We revisit the current design of LSM-tree for heterogeneous storages and conceptually characterize the systems in terms of data granularity and compaction algorithms.
- We design *SA-LSM*, based on survival analysis, to identify the cold data as a time-to-event prediction problem with censored data. This data-driven approach demonstrates great potentials compared to traditional strategies.
- We implement a lightweight communication protocol between the *SA-LSM* external service and the database kernel. This decoupled architecture mitigates the resource contentions caused by the learning algorithms.

We implement *SA-LSM* for X-Engine [28], a commercial-strength open-source LSM-tree storage. Extensive experiments on real-world workloads show that it can decrease the tail latency by ranging from 31.5% to 78.9%, compared to the state-of-the-art solutions. The generality of this approach and the significantly improved performance show great potentials in related applications.

The rest of this paper is organized as follows. Section 2 discusses the design considerations of LSM-tree on heterogeneous storages and explains why survival analysis is suitable in this scenario. Section 3 introduces *SA-LSM*, including the cold data prediction algorithm and the system design for the *proactive compaction*. We use several representative workloads to benchmark our design. The experiments along with the corresponding analysis are covered in Section 4. The related work is presented in Section 5, followed by a conclusion in Section 6.

2 MARRIAGE BETWEEN LSM-TREE AND SURVIVAL ANALYSIS

In this section, we first describe the characteristics and challenges to manage cold data on LSM-tree based storages. Then, we discuss how to cast the cold data prediction problem as a survival model.

2.1 Revisit the current LSM-tree Design

2.1.1 LSM-tree Preliminaries. LSM-tree was designed as an index data structure for high write throughput and low storage cost [42]. Many modern databases use LSM-tree as the backend storage engine [21, 28], including *Dynamo* [17] at Amazon, *Cassandra* [35] at Apache, *LevelDB* [21] at Google, *RocksDB* [19] at Facebook and *X-Engine* [28] at Alibaba. LSM-tree batches the updates to the persistent storage as sorted and compact runs (e.g., SSTable in RocksDB [19]). It organizes these runs onto layers of exponentially increasing capacities, with hot data on the lower layers.

The typical LSM-tree architecture is illustrated in Figure 4. Data is first written to memory, stored in MemTables, often using in-memory skip lists [44]. Once a MemTable becomes full, it is frozen as immutable, waiting to be flushed as a Sorted String Table (SSTable). An SSTable is a file on disk that contains sorted variable-sized key-value entries that are partitioned into a sequence of data blocks. SST files are stored on multiple levels, namely $L_0, L_1 \cdots$, typically with exponentially increasing capacities. Once a level reaches a pre-configured threshold for the total data size, a compaction operation is triggered. During compaction, SST files are merged according to keys in the next level.

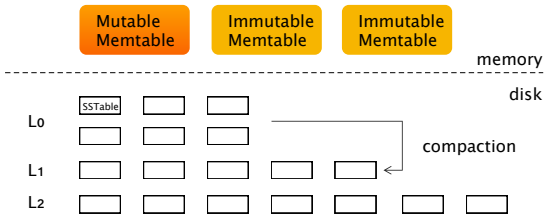


Figure 4: LSM-tree architecture

2.1.2 Challenges for heterogeneous storages. Heterogeneous storages are used to reduce cost, for which however system performance metrics, i.e., the database latencies, are inevitable to be traded off. Conceptually there exists a Pareto frontier when balancing between the above two conflicting objectives. This Pareto frontier characterizes an ideal scenario where an oracle can predictively pin the data that are going to be accessed in the fast storage and only migrate the data that will be barely accessed in the future to the slow storage. Thus there is always a gap between the Pareto frontier and the performance of realistic systems. In practice, to better capture the trace characteristic and identify cold data at a refined granularity, the adopted algorithms usually become more complex. Figure 5 conceptually compares *SA-LSM* with other commonly used solutions, with the Pareto frontier as the limiting boundary. Note that this figure is supported by extensive experiments in Section 4, as RocksDB [8], Mutant [59] and PrismaDB [46] are based on random, exponential smoothing and LRU algorithms, respectively.

Identifying cold data can help reduce read amplification, a common problem for LSM-tree storages, which is defined as a read request accessing multiple layers. To bound the read amplification, some studies have introduced auxiliary data structures to eliminate unnecessary disk accesses, e.g., SST fence pointers and bloom filters [19, 21]. In addition, caching schemes efficient to improve LSM-tree read performance, e.g., block cache and row cache [19, 35, 56]. There also exist works that provide a read-aware LSM-tree, which

Table 1: Design choices for LSM-tree on heterogeneous storage

	Granularity	*Separation algorithm*	*Promotion*	*Demotion*	*Trigger time*	*Integration with DB kernel*
RocksDB [19]	SSTable	Random	No	Yes	Compaction	Coupled
Mutant [59]	SSTable	Exponential smoothing	Yes	Yes	Compaction	Coupled
PrismDB [46]	Record	LRU	Yes	Yes	Compaction	Coupled
SA-LSM	Record	Survival analysis	No	Yes	Active trigger	Decoupled

Promotion is the process of moving data from lower layers to higher layers; demotion represents the opposite direction.

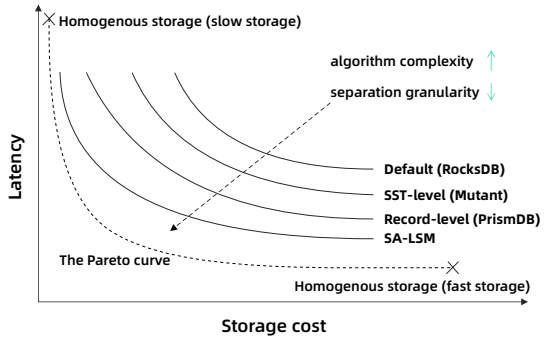


Figure 5: Compare *SA-LSM* and other algorithms for LSM-tree with heterogeneous storages

considers read patterns when reshaping the LSM-tree [46, 59]. From this perspective , *SA-LSM* can contribute by better optimizing the data layout, which is mainly determined by the write operations and the compaction.

The LSM-tree is very efficient for write operations due to the deferred batch updates on persistent storages. By default, a compaction is triggered when the data size of a level reaches a predefined threshold. However, the layered persistent storages naturally increases the read path if the target data resides in the bottom layers. In this case, mapping the bottom layers to slow storages deteriorates the system performance. To select the right data to be migrated onto the slow storage, Mutant [59] assigns SSTables to heterogeneous storage based on the *SSTable temperature*, which is defined as the access frequencies with exponential smoothing in the past epoch divided by the SSTable size. Thus, Mutant does not distinguish cold and hot data within an SSTable and the selected data keys within an SSTable are only dependent on writes. To this end, PrismDB [46] observes that an SSTable contains data records of different popularities. To refine the prediction granularity on the data records, it uses a lightweight mechanism based on the LRU algorithm and introduces *pinned compaction* to allocate the records onto different levels. We summarize the techniques used in these systems in Table 1.

2.2 Using Survival Analysis to Archive Data

Survival analysis is commonly used in clinical studies, where the time-to-event prediction is usually on the occurrence of a naturally observed end point of interest, such as relapse or death, for individual participants, e.g., patents. In contrast, cold data prediction is on recurrent events, based on a sequence of access points. Recall that a unique feature of survival analysis is the censored events since some data accesses do not occur within the observation window.

To cast the cold data prediction problem as a survival model requires to address the above points. Then, we predict the earliest access event in the future for an individual data record. All of the data records are ranked according to their predicted access events, where the top ranked ones are considered to be hot data. As a result, the cold data can be compressed or migrated to the cold tiers.

In the following, we first define the AWPI workload. By automatically detecting the presence of such workload, *SA-LSM* is adaptively enabled, as detailed in Section 3.2. Then, we re-examine the compaction strategies from the perspective of technological advances. Last, we describe the algorithm details of *SA-LSM*, including designing the feature vector, defining the event and labels, generating the training data set and utilizing the time-to-event prediction to rank the popularities of the data records.

2.2.1 AWPI Workload. We identify a typical class of workload, named **a**rchival **w**rite and **p**oint read **i**ntensive (AWPI). It is specially suitable for LSM-tree based heterogeneous storages, with three main characteristics.

Firstly, it has been shown that the write and point read operations represent a significant fraction of the workload and the proportion of write operations in modern applications keeps increasing [48]. In addition, only 20% of data is actively accessed while the rest 80% remains cold. The cold data has a 60% cumulative annual growth rate, identified as the fastest growing storage segment [11, 39, 50]. Due to the orders of magnitude performance difference between DRAM and persistent layers, enterprise databases serving OLTP workloads often use flash-based SSDs to narrow that gap.

Secondly, the popularities of the data records tend to decrease over time. This is quite common in OLTP applications such as e-commerce and instant messaging, where the popularity of an order or a message decreases over time. We track 50K data records and investigate their popularities over time. Figure 6 plots the normalized access frequencies of a typical archival workload within 90 days. The access counts are aggregated on each day and normalized by the number of accesses on the first day when the records are created. The overall popularity decays over time, which even decreases to less than 1% after 50 days and remains at a low level afterwards. Hence, it is reasonable to migrate the cold data to a cheaper storage. In order to further investigate the data records studied in Figure 6, we measure the lifetime of a data record, which is the interval between the creation time and the last access time point of each data record in 90 days. We plot the lifetime distribution in Figure 7. As can be seen, the distribution is long-tailed. Thus, simply using a fixed time threshold is difficult to separate the cold and hot data. Interestingly, the records with a 1-day lifetime constitute the largest portion of the workload.

Thirdly, a large portion of the requests are *covered queries*, which can be served using an index without examining the data records. Correspondingly, the compaction strategy needs a carefully design

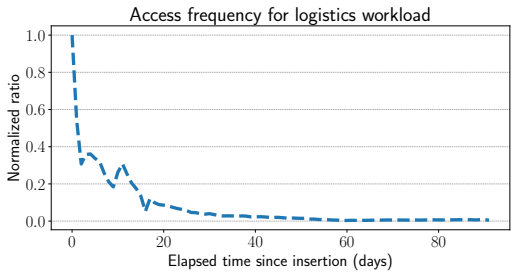


Figure 6: The number of data accesses over time normalized by that number on the first day when records are created

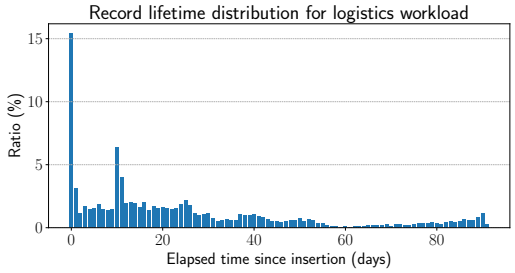


Figure 7: The distribution of lifetime for *logistics* workload

for these complexities, e.g., to properly define the event for the survival analysis to avoid noise, as shown in Definition 2.1.

2.2.2 Re-examine compaction strategies. Due to the technological advance of modern storages, data records with decaying long-tailed access frequencies need a careful treatment to avoid unnecessarily touching the slow mediums when accessing wrongly classified cold data. Interestingly, the traditional eviction policies employed in caching mechanism, e.g., LRU [45, 51], have also been widely used for LSM-tree compactions. These algorithms are based on simple data structures and thus are efficient for fast processing. However, it has relatively limited precision in identifying cold data especially with decaying long-tailed popularities. Applying traditional algorithms in this case could lead to a sub-optimal performance. Note that LSM-tree compactions are not very frequent. Thus, relatively heavier computations for more advanced algorithms are often allowed.

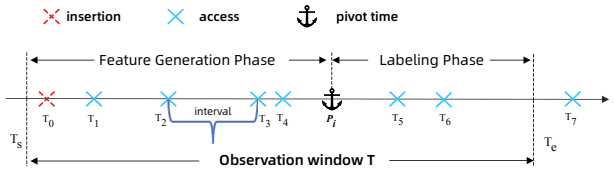
From a historical perspective, cache eviction policies such as LRU may also not be the best option for a compaction strategy. In 1987, Jim Gray et al. [26] established the five-minute rule "Pages referenced every five minutes should be memory resident". They arrived at this value by computing the break-even interval at which the cost of holding a page in memory matches the cost of performing I/O to fetch the page from HDD. The five-minute rule would still work if the critical technological aspects, e.g., capacity, latency and bandwidth, advance with the same pace. However, new technologies [25] such as flash memory [24] require to recompute the break-even interval for modern storage hierarchy [9]. For the latest technology, the break-even interval for SSD and HDD even increases to one day, which indicates that all performance critical data will soon, if not already, reside only on DRAM and SSD, with HDD being relegated to a high-density storage for cold data. This also motivates us to design *SA-LSM*, which utilizes more complex and effective algorithms.

2.2.3 Cast Cold Data Identification as a Survival Analysis problem. *SA-LSM* formulates the cold data identification as a ranking problem using the predicted access events. For a survival model, we need to define the censored data, which have some unique features for this problem due to the recurrent data access points.

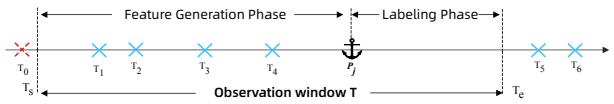
As shown in Figure 8, we introduce the concept of *observation window* $[T_s, T_e]$, in which most of the data records should have sufficient observations for training the model. Note that the observation window should be long enough, otherwise the prediction accuracy could be compromised. However, due to the decaying effect described earlier for the workload, the window length should not be too long as well, since the obsolete data access information would not help and only increase the training time and the overhead of storing the data. Regarding how to optimize the length of the observation window, we simply set it as a hyper-parameter, and use the commonly adopted cross validation approach to select a good window length.

Pivot selection: In order to successfully train a survival model through supervised learning, we need to define the training data by extracting the features and providing the corresponding labels. Note that the label for survival analysis consists of two parts, namely the event time and the indicator that shows whether an event occurs or not.

To this end, we split the observation window into two phases using a random pivot, as shown in Figure 8. The first phase is to extract features for the data records and the second phase is to form labels for the training data. For example, Figure 8 shows two different pivots, P_i and P_j selected for data record i and j, respectively. Each of the pivot time splits the observation window T into two phases. For data record i, the phase $[T_s, P_i]$ before the chosen pivot time P_i is the *feature generation phase*. It forms the features for data record i using the observations therein. The phase after the pivot time in $[P_i, T_e]$ is the *labeling phase*.



(a) A sample generated from record i



(b) A sample generated from record j

Figure 8: Random pivots split the observation window into two phases for each of the data records

To properly select the random pivot requires a careful treatment for a sequence of access time points. Due to the inspection paradox for a renewal process [13], if we simply select random time points,

then those larger intervals will be selected with a higher probability, which causes a bias for estimating an event time.

Mathematically speaking, the inspection paradox states that for a random time point P, the interval containing P is stochastically larger than the length of a randomly selected interval V. In this case, the interval $V^{(r)}$ between the pivot P and its next access point follows a residual distribution, which satisfies

$$\mathbb{P}[V^{(r)} \leq t] = \frac{\int_0^t \mathbb{P}[V > x]dx}{\mathbb{E}[V]}, t \geq 0.$$

To avoid the sampling bias caused by the inspection paradox, we randomly select *n pivot time points* for each data record according to a two-step sampling scheme, as shown in Algorithm 1. Since the first step randomly selects an index that is independent of the interval length, we can remove the sampling bias for a renewal process.

Algorithm 1: Two-step sampling for pivot selection to avoid bias

Input: A sequence of access time points $\{T_1, T_2, \cdots, T_k\}$ within the observation window for data record i.
Output: n randomly selected pivots.

```
for i ← 0 to n do
    ξ ← select an index uniformly at random from [1, 2, ··· , k] without replacement
    P_i ← sample a time point uniformly at random on [T_{ξ−1}, T_ξ]
end
return P_1, P_2, ··· , P_n
```

Label generation: One straightforward way to define an event is to use the first data access after the pivot time in the labeling phase. However, it has been reported that occasionally some rare accesses can introduce noise [52, 57] to wrongly promote a cold data record to become hot. To address this difficulty, we instead define an event to be an access with its associated interval, which is the time between this access and its previous access, being less than a specified threshold.

Definition 2.1 (τ-event). A τ-event of a data record i is the first access at time T_v after the pivot time P such that the interval $T_v - T_{v-1}$ between this access and its immediate previous one is less than τ. The corresponding event time e_i is defined to be $T_v - P$.

To better understand an event, we use an example in Figure 8 (a) for illustration. The interval for T_5 is defined to be $T_5 - T_4$. If $T_5 - T_4 > \tau$, then we skip the access time T_5 and keep searching the next coming point T_6. Suppose its interval $T_6 - T_5 < \tau$, then T_6 is considered to be an event. The corresponding event time is equal to $T_6 - P_i$ in this case. Based on the definition of τ-event, we can define censored data. For example in Figure 8 (b), the event occurs at T_6 if assuming $T_5 - T_4 > \tau, T_6 - T_5 < \tau$ and P_j being the pivot time. This event is censored since it is outside the labeling phase $[P_j, T_e]$. In contrast, the event for data record i occurs before T_e, which is not censored.

Now, we can formally define the label (y_i, δ_i) for an event of an data record i. Here δ_i is a binary event indicator, *i.e.*, $\delta_i = 1$ for an uncensored event and $\delta_i = 0$ for a censored event; and y_i is the label time equal to the minimum of the event time and the interval length of the labeling phase, i.e.,

$$y_i = \begin{cases} e_i & \text{if } \delta_i = 1 \\ T_e - P_i & \text{if } \delta_i = 0 \end{cases} \tag{1}$$

In Figure 8, the label is $(T_6 - P_i, 1)$ for pivot time P_i in (a) and $(T_e - P_j, 0)$ for pivot time P_j in (b).

Note that for each pivot time, we can generate one sample for a data record. Therefore, one data record can produce multiple samples based on a sequence of access times together with the corresponding pivots. Extensive experiments with real workloads show that setting τ to be one day is a good configuration for the tested AWPI workload.

Feature Selection: Carefully designed features are critical for a survival model. In the feature generation phase, the last access interval has been used by traditional cache eviction policies, e.g., LRU, to evict a data record as a part of the compaction strategy. Therefore, this access interval is chosen as part of the feature. In addition, we can incorporate more information due to the flexibility of the model allowing heavy computations.

We identify two set of important features, which are concatenated to form a feature vector X_i for data record i.

- Access features: In addition to the aforementioned last access interval, which is precisely defined to be the interval between the pivot and the previous access, we further keep looking backward to consecutively select another ξ (e.g., 7) number of access intervals. Then, we add the time difference between the insertion time and *pivot time*, where the insertion time could be either before or after T_s. We also use the time difference between the last update time and the *pivot time* and the time difference between the insertion time and the last update time.
- Semantic features: We also measure the total numbers of *UPDATE* and *SELECT* operations per day over the last 20 days as well as the individual numbers on each of the columns, padding zeros if the feature generation phase is less than 20 days. This hyperparameter 20 is based on the discussion in Section 2.2.2 and cross validation.

2.2.4 Survival Model. The objective is to predict the time-to-event for data record i after training the survival model based on the training data $(X_i, y_i, \delta_i), i = 1, 2, \cdots$. In general, a survival function for an event time T is given as follows:

$$S(t) = Pr(T \geq t). \tag{2}$$

Another commonly used function is the hazard function $h(t)$, which is defined as follows:

$$\begin{aligned} h(t) &= \lim_{\Delta t \to 0} \frac{Pr(t \leq T < t + \Delta t | T \geq t)}{\Delta t} \\ &= \lim_{\Delta t \to 0} \frac{F(t + \Delta t) - F(t)}{\Delta t \cdot S(t)} = \frac{f(t)}{S(t)}, \end{aligned} \tag{3}$$

where $F(t) = 1 - S(t)$ is the cumulative death distribution function. The cumulative hazard function (CHF) can be obtained as follows:

$$H(t) = \int_0^t h(u)du. \tag{4}$$

To measure the quality of a survival model on a data set, c-index (i.e., concordance index) is one of the most commonly used performance measures. It is defined as follows:

$$\hat{c} = \frac{1}{M} \sum_{i:\delta_i=1} \sum_{j:y_i<y_j} I\left[\hat{S}(y_j|X_j) > \hat{S}(y_i|X_i)\right], \tag{5}$$

where $i, j \in \{1, ..., N\}$ and $I[\cdot]$ being an indicator function. Here M denotes the total number of formed pairs and $\hat{S}(\cdot)$ is the estimated survival probabilities.

As different survival models have been extensively compared in empirical studies [14, 18, 33, 40], we compare two of the most popular survival models: the Random Survival Forest model [29] (RSF) and the Cox model [20]. Extensive comparisons between Cox model and RSF have been conducted [14, 18, 33, 40]. We also compare them for the cold data prediction. The detailed experimental results can be found in Section 4. Due to the robust performance of the Random Survival Forest model, it is used as the default one for *SA-LSM*. For the completeness of the presentation, we describe RSF in Algorithm 2 based on a standard reference [29].

Algorithm 2: RSF for Cold Data Identification [29]

1. Generate the training data by following the procedure in Section 2.2.3 on n pivots for each data record.
2. Draw B bootstrap samples. Note that each bootstrap sample excludes on average 37% of the data [29], called out-of-bag (OOB) data.
3. Grow a survival tree for each bootstrap sample. At each node of the tree, randomly select p feature variables. The node is split using these variables to maximize a survival difference between its child nodes.
4. Grow the tree to full size such that a terminal node should not have less than $d_0 > 0$ events.
5. Calculate a CHF (equation (4)) for each tree; average to obtain the ensemble CHF.
6. Using OOB data, calculate prediction error for the ensemble CHF.

3 *SA-LSM* IN PRACTICE

SA-LSM is computationally more demanding than the traditional compaction strategies. In order to mitigate resource contentions, we design a proactive method that relies on an external service to schedule the compactions in Section 3.1. Then, we discuss the system support and deployment considerations in Section 3.2.

3.1 Proactive v.s. Passive Compactions

After the model is trained, we can utilize it to predict the records' next event times. These predicted times can be used to rank the data records, so as to optimize the data layout through proactively conducting compactions for the LSM-tree. This should be distinguished with the default *major compaction* [42] that is routinely conducted to maintain an LSM-tree. For example, such compactions between L_1 and L_2 are passively triggered when the data size of L_1 exceeds a specified threshold. Therefore, we term such a default operation as a *passive compaction*. In contrast, *SA-LSM* is based on machine learning to proactively schedule a compaction, which thus is called a *proactive compaction*.

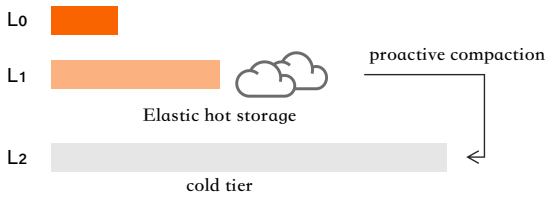


Figure 9: *Proactive compaction* adjusts the size of the LSM-tree L_1 layer tailored to individual workloads

Algorithm 3 describes the detailed procedure. The hyperparameter α is called the time-to-event threshold. By controlling this hyperparameter, only when the predicted event time is less than this threshold should the corresponding data record remain on L_1. Otherwise, it will be added to the list of candidates to be moved to L_2. Notably, the total size of the data residing on L_1 now becomes dynamic, depending on each workload's characteristics. This feature is suitable for elastic cloud storage that can autoscale. It enables self-tuning and workload-aware capabilities for an elastic LSM-tree storage. Figure 9 shows an architecture based on the elastic cloud storage. Therein the hot layer can adaptively autoscale its capacity to improve the cost-efficiency for various workloads.

Algorithm 3: Proactive compaction

Input: Model M, the p dimensional feature vectors $X_i \in \mathbb{R}^p, i = 1, 2, \cdots, n$ for the n samples, time-to-event threshold α.

```
Initialize 𝕊.
for i ← 0 to n do
    pred = M.inference(X_i)
    if pred > α then
        𝕊.append(i)
    end
end
External agent notify database to perform compaction
foreach key ∈ L_1 do
    if key ∈ 𝕊 then
        Append to SSTable
    end
end
Compact data records in SSTable to L_2
```

How to optimally schedule the proactive compactions is beyond the scope of this paper, since a number of different objectives can be taken into considerations. For example, we can use a total cost budget for the storage or judge the service of quality based on collected performance metrics on different layers. In our practice, we choose to schedule the proactive compactions at the time periods when the transaction rate of the hosting database is low. This can avoid resource contentions caused by compactions.

3.2 System Support and Deployment

After separating cold and hot data using *SA-LSM*, the system still needs to manage the cold data on the layered storage.

3.2.1 Implementation of the standalone external compaction service. To make the deployment more flexible, we design a non-intrusive

architecture that offloads the CPU-intensive work, e.g., model training and inference, to an external service. Figure 10 shows the system implementation of *SA-LSM* and the workflow of a proactive compaction, with the key steps, from step 1 to step 4, being highlighted therein. We explain the details as follows.

Step 1: To automate the proactive compaction for a database table, we utilize a data definition language (DDL) in MySQL. Note that each table has a dedicated trained survival model. To this end, a *compaction_strategy* column is added in the *information_schema* table, which indicates whether the proactive compaction is enabled or not for an individual table. To be specific, adding the keyword ***proactive_compaction*** enables this functionality when creating the table, as is shown in the following example:

```
CREATE TABLE table_name (column_name
 column_type) proactive_compaction;
```

Adjusting the compaction mode on an existing table is also supported, which modifies the *compaction_strategy* column. Note that this mechanism allows an adaptive implementation, which can dynamically disable proactive compactions when the workload is heavy and enable it when light. This adaptive approach can increase the robustness of the system. Specifically, we keep monitoring the QPS and the I/O utilization on the cold tier. If the key performance metrics exceed some pre-configured thresholds, the external service automatically disables the proactive compactions and falls back to the default passive compaction strategy.

```
ALTER TABLE table_name compaction_strategy=
 proactive;
```

Step 2: The external service checks the *information_schema* table to determine the compaction strategy for each individual table. Once a proactive compaction is requested, the external service fetches the log data of the involved tables from the *Query store*. Each table trains a dedicated survival model as the access patterns could vary significantly shown by the Kaplan-Meier curves in Section 4.2.1.

Step 3: As described in Section 2.2.1, the proactive compaction is suitable for *AWPI* workload. Therefore, we provide a mechanism to detect whether the target workload is an *AWPI* workload or not before training the survival model. Specifically, we analyze the workload trace in an observation window by calculating the percentage of the point writes and reads of all the requests. Only when this number is more than a pre-configured threshold (e.g., 90%) should we conduct the proactive compactions.

Step 4: The external service identifies the keys for the cold data by Algorithm 3. Then, it transfers the results to the database. The external service notifies the database to perform the compaction by running the following command:

```
SET GLOBAL xengine_compact_cf=$subtable_id +
   ($task_type << 32);
```

The above command is the same as the one for the default passive compaction of RocksDB [8]. When it is sent to the kernel, a compaction task is created and put into a queue, waiting the background compaction thread to execute. When the compaction task is schedule in the kernel, the compaction thread loads the identified keys of the cold data into memory. Then, after traversing the records on L_1, it archives the matched data to the cold tier. After the proactive compaction is finished, the cold keys are released.

3.2.2 Discussions on the external service. Integrating a complex machine learning algorithm into the database kernel could result in large computation overhead and memory footprint. An external service could reduce the resource contention. More importantly, it also brings the opportunities to utilize more advanced analytic algorithms to further improve the accuracy of identifying cold data by using multiple data sources. For example, one can use different workloads from multiple database instances, and also take into considerations other complex factors like holidays and market events.

Computation overhead: Since we rely on relatively complex learning algorithms, the model training process could consume far more CPU resource than directly applying traditional methods. In our experiments, to train the survival model for different workloads typically takes about one hour using 8 CPU threads. This is quite an overhead compared with the normal passive compaction tasks, which usually take minutes to finish. Note that even the passive compactions have been reported to impact system performance [28, 60].

Regarding the memory footprint, a typical size of a data record ranges from tens of bytes to hundreds of bytes. This length is comparable to the memory to store the feature. Therefore, the model training stage also consumes large memory resource. For example, in our experiments, the model for each workload is about several megabytes. Considering a database instance may have hundreds of tables, the overhead incurred by training the survival models could throttle the normal operations of the database if not using the external service.

Future research for external service: An external service can serve multiple database instances, which brings together multiple data sources for possibly making better decisions. In addition, more advanced algorithms, e.g., meta learning [54] and transfer learning [55], can be adopted to improve the prediction accuracy, shorten the training time and reduce the required samples. For example, one can use pre-trained base models from multiple similar database instances to accelerate the training process.

4 EVALUATION AND ANALYSIS

In this section, we demonstrate the superior performance of *SA-LSM* using extensive experiments on real workloads. We also conduct detailed analysis on the system performance.

4.1 Experimental setup

***Testbed*:** We evaluate *SA-LSM* on a server featuring Intel Xeon Platinum 8163 2.5 GHz 20-core CPUs with two-way hyper-threading and a 88 GB Samsung DDR4-2666 main memory, running Linux 5.10.23. The machine is equipped with an elastic hot storage (Enhanced SSD) and a RAID 0 consisting of 8 HDDs as the cold tier.

***Workloads and Datasets*:** We collect six workloads from real-world applications to evaluate the system performance. Each workload is generated by collecting the trace data for two months. They have the following two characteristics:

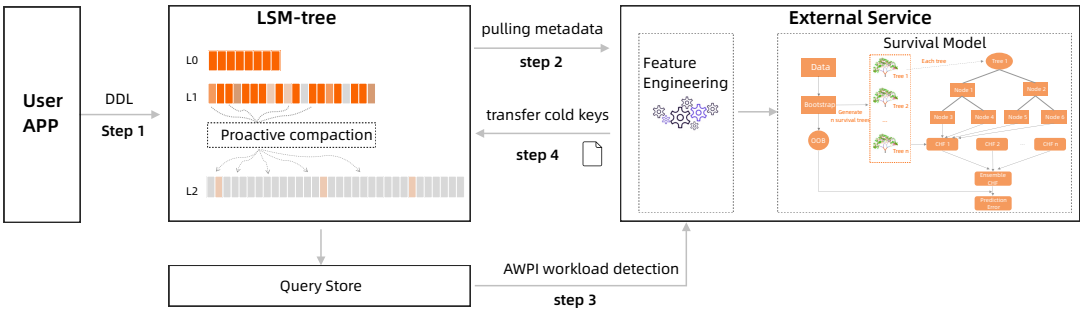


Figure 10: The system implementation of *SA-LSM* and corresponding protocols between the external agent and the database kernel; *SA-LSM* is able to identify and archive cold data at record granularity and does not suffer from the contention with the database kernel via offload the algorithm as a standalone service

(1) The majority of the SQL templates are point queries and point updates.
(2) The popularities, i.e., access frequencies, of data records keep decreasing over time since being inserted into the database. Therefore, they are the representative *AWPI* workloads defined in Section 2.2.1.

The details of the statistics of the workloads and corresponding datasets are summarized in Table 2. Note that we set the pivot number to be 10 in Algorithm 2, which means that each record can generate 10 samples. We randomly split the records and form the training set (80%) and test set (20%).

Table 2: Summary of workloads and corresponding datasets.

	#traces	#records	r/w ratio	#samples	censored %
AUCTION	95 million	754 k	3.7	4,081 k	55.2%
DISPUTE	68 million	192 k	42.5	963 k	71.6%
FUNDS	41 million	6,295 k	1.9	32,780 k	91.8%
LOGISTICS	173 million	2,826 k	16.8	18,732 k	67.8%
ORDER	117 million	4,505 k	3.6	28,570 k	81.9%
PAY	124 million	1,736 k	13.8	10,017 k	45.0%

We implement all of the above algorithms in the external agent and replay real-world workloads on X-Engine via CloudBench [2]. CloudBench is a tool to replay the workloads at a specified rate that also preserves the order of transactions. For the experimental results, we report QPS, the 99-th latency, CPU utilization and IO utilization on the cold tier.

***Baseline and Metrics*:** We compare *SA-LSM* not only with the traditional algorithms, e.g., least recently used (LRU) [7], largest frequently used (LFU) [6], LRU-k [41] and LIRS [30], but also with machine learning based methods, e.g., gradient boosted decision tree (GBDT) [12].

- **LRU**. Migrate the least recently access data first.
- **LFU**. Migrate the data with the least access frequencies.
- **LIRS**. The Low Inter-reference Recency Set algorithm is an alternative to LRU with an improved performance. It is based on the reuse distance, a locality metric for dynamically ranking accessed data, to identify cold data for compaction.
- **LRU-k**. The LRU-k algorithm is an variant of LRU which keeps track of the time interval between the last Kth access and the last one. In practice k is usually set to 2. Recall our definition of an event in Definition 2.1 that also requires two close consecutive accesses.
- **GBDT**. Gradient Boosting Decision Tree [12], a commonly used machine learning algorithm, is used to separate cold and hot data. The features we use are identical to the ones introduced in Section 2.2.3 for *SA-LSM*.
- ***SA-LSM***. We choose Random Survival Forest (RSF), a survival analysis algorithm based on random forest, to infer which data would not be accessed within a specified time window. To fairly compare with GBDT, we use the identical feature vector to train the model.

Essentially we selectively migrate a certain fraction of cold data to the slow storages using a classification algorithm. Therefore, to quantify the quality of the algorithm, we use the confusion matrix [3] to evaluate the system performance and cost. The confusion matrix is a summary of classification results. According to the consistency of the predicted results and ground truth, we divide the samples into four categories. 1) true positive (TP) for correctly predicted cold data; 2) false positive (FP) for incorrectly predicted cold data; 3) true negative (TN) for correctly predicted not-cold data; 4) false negative (FN) for incorrectly predicted non-cold data.

In addition to the c-index defined in equation (5) to evaluate the quality of a survival model, we introduce the cold data false identification rate based on the above-mentioned FP and TP metrics

$$cold\_fir = \frac{FP}{FP + TP} \tag{6}$$

to measure the precision of identifying the cold data. A good compaction strategy should maintain a low $cold\_fir$ to reduce the amount of hot data to be migrated to a cold storage tier.

4.2 Experimental evaluations

To understand why *SA-LSM* can achieve a superior performance, we conduct the following experimental studies. First, we use Kaplan-Meier analysis to show that the data records from different workloads have distinct life spans. This is the key reason why an individualized treatment of the workload can improve the accuracy of separating cold and hot data. Then, we conduct detailed analysis on the algorithmic aspects and system performance metrics. It demonstrates that *SA-LSM* can have great potentials in the related applications beyond the studies shown in this paper.

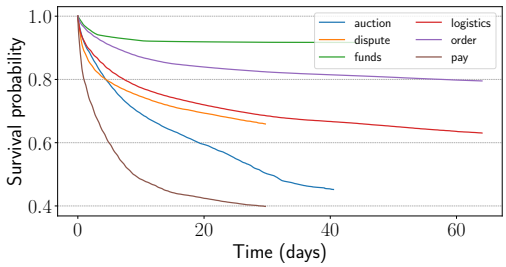


Figure 11: KM curves of the six workloads differ significantly, implying that each individual workload should have a dedicated survival model to capture its unique characteristics

4.2.1 Visualization of Kaplan-Meier curves. The Kaplan-Meier (KM) estimator [22] is a non-parametric statistic used to estimate the survival function from the lifetime data. In this section, we study the Kaplan-Meier (KM) curve for each workload. The estimator of the survival function $S(t)$ is given by:

$$\hat{S}(t) = \prod_{t_i < t} \frac{n_i - d_i}{n_i}. \quad (7)$$

In the above equation, t_i is the access time, d_i is the number of data accesses that have occurred before time t_i, and n_i is the number of data records that have survived up to time t_i. A survived data record refers to one that either has not yet been accessed or is not censored.

For identifying cold data, KM curves can be used to approximate the distribution of the next access time interval. Figure 11 shows the six workloads' KM curves, respectively. The survival probability starts at 1 on the first day, since no accesses have happened yet. As time goes by, data accesses occur on a fraction of the data records and the survival probability keeps decreasing. Each of the curves eventually reaches at a certain survival probability, which represents the fraction of the censored data. For example, the *funds* workload has over 90% of censored samples that are not observed within the observing window. While for *pay* workload, the fraction of the censored data drops to about 40%.

As we can see, the decreasing rates of these curves vary significantly across different workloads. This is the critical reason why we treat each individual workload separately to capture its unique characteristics. Specifically, we train a dedicated survival model for each workload, as described in Section 3.2. Note that for system implementation, each table represents a different workload.

4.2.2 Algorithmic performance analysis. To compare the performance of different algorithms, we report the c-index values, defined in equation (5), for GBDT, ANN, Cox [20], DeepSurv [31], CoxTime [34], CoxCC [34] and RSF [29] on six datasets in Table 3. A c-index value equal to 1 refers to the model making a perfect prediction while a c-index value less than 0.5 means the prediction is worse than random guessing. The experimental results are presented in table 3 for the six real-world workloads. We test five survival models, including RSF, Cox, DeepSurv, CoxTime and CoxCC, where the last three ones are based on deep learning. To demonstrate the superior performance of survival models in this case, we additionally compare with two machine learning models including GBDT and Artificial Neural Network (ANN, also called Multi-Layer Perceptron), which are not based on survival analysis. For deep learning based methods, the neural network consists of 2 hidden layers each containing 32 cells. The network is trained via Adam optimizer with a batch size of 256 and a learning rate of 0.1. Early stopping is performed to find the best training epoch. GBDT outperforms ANN in all six workloads. Considering the deep learning based methods are suitable for a big data setting, we also train a shared model (ANN (shared)) for all workloads. However, the shared model performs worse on the six workloads due to the different access patterns of the workloads. The c-index of RSF is improved by ranging from 8.9% to 30.3% compared with GBDT. It clearly demonstrates the advantages of using survival analysis to predict cold data.

RSF gives the best results in all of the workloads except *pay* workload for this purpose. Note that Cox model is better than GBDT for most of the workloads except *auction* workload, which clearly shows the power of survival model for censored data. RSF has advantages as it is data driven and has no model assumptions, while Cox assumes that a covariate is multiplicative to the hazard rate. For DeepSurv model, the linear relationship in the Cox model is replaced with a neural network. CoxCC model extends Cox with a time-dependent hazard ratio, learned by a neural network. CoxCC is a proportional version of the CoxTime model (the hazard ratio is time-invariant), and only CoxTime model outperforms RSF on *pay* workload. Due to RSF's robust performance, we adopt it as our survival model in the following evaluation.

Table 3: Comparison of c-index between GBDT and survival analysis based methods

Workload	*auction*	*dispute*	*funds*	*logistics*	*order*	*pay*
ANN	0.701	0.695	0.732	0.715	0.701	0.681
ANN (shared)	0.627	0.659	0.725	0.702	0.681	0.651
GBDT	0.750	0.751	0.758	0.747	0.752	0.722
Cox	0.673	0.789	0.918	0.805	0.819	0.747
DeepSurv	0.717	0.809	0.985	0.825	0.843	0.782
CoxTime	0.761	0.824	0.976	0.794	0.887	**0.865**
CoxCC	0.728	0.810	0.983	0.825	0.838	0.794
RSF (v.s. GBDT)	**0.817 (+8.9%)**	**0.864 (+15.0%)**	**0.988 (+30.3%)**	**0.872 (+16.7%)**	**0.909 (+20.9%)**	0.842 (+16.6%)

For each workload, we use the trained survival model to predict the τ-event within the next coming 30 days. For fair comparisons, the baseline methods are configured to archive the same number of data records according to their own policy with *SA-LSM*.

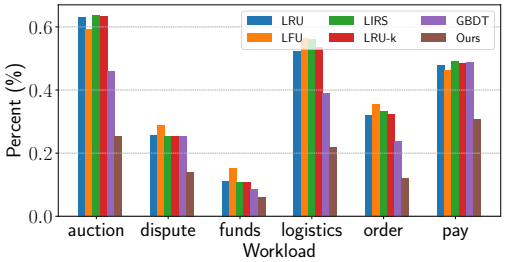
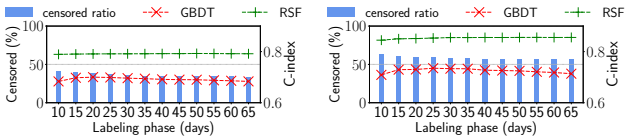


Figure 12: Compare *cold_fir* for different compaction strategies on six workloads

Figure 12 summarizes the comparisons on the *cold_fir* metric for the *logistics* and *order* workloads. Notably, *SA-LSM* can effectively decrease the *cold_fir* metric by ranging from 31.0% to 48.9% compared with the best baseline.

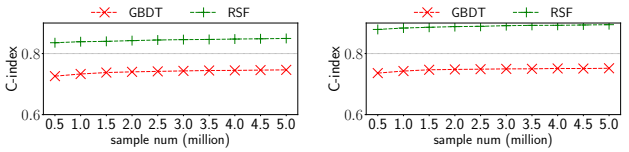
4.2.3 Impacts of the observation window size and the censored data. To justify using a survival model for cold data prediction, we investigate how the labeling window size (recall that observation window is split into feature generation phase and labeling phase in Section 2.2.3) and the fraction of the censored data impact the algorithm performance. Figure 13 plots the c-index under different window sizes. We use 20 days' records to form the training set and vary the labeling window from 10 days to 65 days. We observe that larger labeling window sizes indeed reduce the fractions of the censored data, which drop from 41.2% to 33.4% for *logistics* workload and from 62.9% to 56.0% for *order* workload. Remarkably, RSF still outperforms the GBDT model. Thus, increasing the labeling window sizes is inefficient for *AWPI* workload, since the τ-event has a high probability to be still absent in a long observation window.

Figure 14 demonstrates the impacts of the censored data and the number of training samples on the algorithm performance. For RSF, we use random sampling to generate the same number of training samples as the GBDT. The metrics of both methods increase as more samples are used. However, RSF can outperform GBDT even when only utilizing 10% of the latter's training samples through properly handling the censored data.



(a) *Logistics* workload **(b) *Order* workload**

Figure 13: Impacts of the observation window sizes



(a) *Logistics* workload **(b) *Order* workload**

Figure 14: Impacts of the fractions of the censored data

4.2.4 System performance analysis. We test *SA-LSM* on the open source X-Engine [28] by using heterogeneous storages on a 3-layer LSM-tree. Specifically, L_0 and L_1 layers are configured as the hot tier and L_2 as the cold tier, respectively.

Table 4: Micro-benchmark on the heterogeneous storages

Layer	Medium	IOPS	Latency (us)	price($/TB/month)
L_0, L_1	ESSD	4380	228	574
L_2	HDD	197	5050	21

***Micro-benchmark on different storage mediums*:** First, we test a micro-benchmark (random 4K read QD1 benchmark) on the I/O performance of the hot and cold tier using fio [4]. Table 4 summaries the performance metrics of the storage mediums on different LSM layers. The IOPS of the hot tier is 22.2 times larger than that of the cold tier. The average latency of HDD is 22.1 times slower than that of ESSD. Thus, there exists a large performance gap between the two storage mediums. Hence, the data compacted to the cold tier need to be carefully selected to avoid unnecessarily accessing HDD. Also, the cost per unit of ESSD is 27.3 times more expensive than HDD. By migrating data to the cold tier, the Total Cost of Ownership can be significantly reduced.

***Micro-benchmark for proactive compactions*:** The transaction throughput and CPU utilizations for *proactive compaction* are presented in Table 5. Compared with the *passive compaction* in LSM-tree (defined in Section 3.2.1), proactive compactions exhibit a similar throughput but with a higher CPU utilization, due to the additional comparsions between the keys in L_1 and the cold keys provided by the external service. Since compactions are not frequently conducted, the performance penalty is tolerable without causing too much overhead to the database kernel.

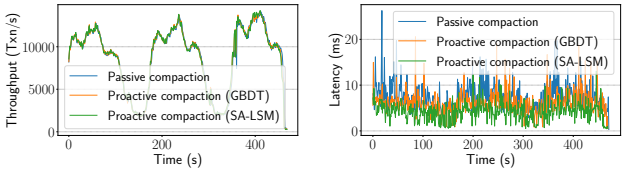
Table 5: Comparison of Proactive and Passive Compactions

Compaction task	Throughtput (MB/s)	CPU utilization
proactive compaction	89.2	53.3%
passive compaction	91.7	35.5%

***Macro-benchmark*:** For each workload, we preload 20 days' traces of the data records in the test set to the system and manually flush the memtable. Then, we compact all of the data on L_0 to L_1 so that all the data resides on L_1 before we conduct compactions between L_1 and L_2. The block cache is set to 30% of the total data size. Then we compare the different compaction strategies. After the compaction is finished, we use ClouDBench to replay the next three days' traces to benchmark the system performance. [1] We use 128 clients to replay the trace and warm up the system for 60s before collecting the metrics. We report the metrics of *passive compaction* and *proactive compaction* using different algorithms. Since the performances of GBDT is the best baseline, we ignore other algorithms in this comparison.

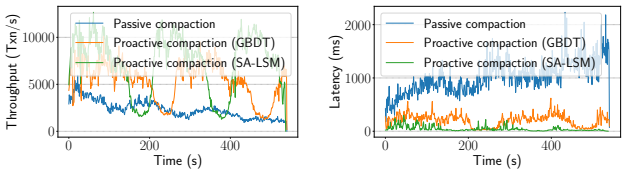
Figure 15 and 16 plot the throughput and the tail (99th percentile) latency. The *passive compaction* has the worst performance since it does not fully utilize the information from data access traces of different workloads. Compared with GBDT, *SA-LSM* reduces the tail latencies by 31.5% for *logistics* workload and 78.9% for *order* workload, respectively. The average latencis are reduced by 5.8% for *logistics* workload and 67.7% for *order* workload, respectively. The reductions of the average latencies are not as significant as the tail latencies due to the existence of the block cache and the hot tier. For *order* workload, the throughput improvement is substantial for *SA-LSM* compared with the best baseline. The reason is that the I/O utilization of *order* reaches the bottleneck of HDD, which results in a higher tail latency in Figure 17. In this case *SA-LSM* can greatly reduce I/O and increase the CPU utilization (Figure 18), yielding a more substantial performance improvement. This can be quantitatively seen by comparing *order* and *logistics* for the fractions of the cold data migrated (51.7% v.s. 28.1%) and the *cold_fir* metrics improvement (48.9% v.s. 43.7%). *SA-LSM* increases the system performance by reducing the number of data accesses on the cold tier. The average I/O utilization is reduced by 57.5% for *logistics* workload and 15.5% for *order* workload.

[1] ClouDBench has a parameter called *rate_factor*, which is a coefficient multiplied to the interval lengths when replaying the trace. To control the replay speed, all our experiments are configured with *rate_factor*= 2×10^{-3} to speed up the process.



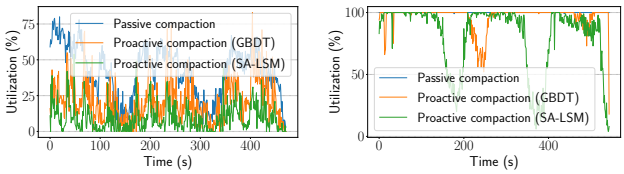
(a) Throughput (b) Tail latency

Figure 15: System metrics for *logistics* workload; The performance improvement on *logistics* workload is less than *order* workload because less data is identified as cold data and migrated to the cold tier.



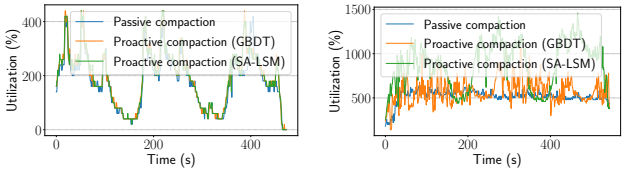
(a) Throughput (b) Tail latency

Figure 16: System metrics for *order* workload; the significant drops of TPS on intervals [130, 200] and [325, 380] are due to the realized high TPS outside these two intervals leaving no enough requests to be performed therein



(a) *Logistics* workload (b) *Order* workload

Figure 17: Cold tier (HDD) I/O utilization for *logistics* workload and *order* workload; I/O utilization of *order* workload becomes the bottleneck of the system and *SA-LSM* improves the tail latency significantly in this case



(a) *Logistics* workload (b) *Order* workload

Figure 18: CPU utilization for *logistics* workload and *order* workload; The CPU resources under the two compared methods for *order* workload are not fully utilized due to their lower TPS throughput

5 RELATED WORK

LSM-tree on Heterogeneous Storages: In order to reduce the total storage cost, several prior works optimize the data layout of LSM-tree based databases on heterogeneous storages. For example, RocksDB [8], Cassandra [35] and Mutant [59] organize SSTables by levels and store cold data on the slower and cheaper storages. However, their adopted cold data identification strategies are relatively simple on a coarse granularity without fully utilizing the information from the collected traces, which limit the system performance. To the best of our knowledge, *SA-LSM* is the first to estimate the popularities on a refined granularity for each of the data records using survival analysis.

Cold Data Identification and Survival Analysis: Recent work on the compaction design mainly focuses on compaction triggering [23, 47] and compaction strategies [15, 16] in order to optimize read and write amplifications. However, the data access patterns also play an important role in determining the cost of accessing data, especially on heterogeneous storages. The current designs of LSM-tree on heterogeneous storages either neglect the diverse patterns of the data accesses, e.g., RocksDB [8], Cassandra [35], or employs a low-complexity algorithm like LRU to track the data popularities, e.g., PrismDB [46] and Mutant [59]. To mitigate the cache invalidation problem in LSM-tree, Leaper [58] integrates decision tree models to predict the hot keys and to prefetch data to the block cache after each flush and compaction operation. Apart from LSM-based storage engines, some work has been done on leveraging the machine learning techniques to solve the cold data identification problem. Octopus [27] and LRB [49] apply gradient boosting decision trees to promote or demote files across the storage tiers in distributed file systems and to guide CDN cache, respectively. Deep learning based methods are also exploited to solve the cache replacement problem [38].

Given the universal existence of the observation window for the recurrent data accesses, always a fraction of the access events are not yet observed. Without properly handling these censored data, some useful information may be lost or cause bias to the prediction. Survival analysis, a statistical learning algorithm commonly used in biostatistics, provides various mechanisms to better process the censored data. Apart from biostatistics, it has also been widely used in other application scenarios, such as crowdfunding [37], click-through rate [10] and cloud databases survivability prediction [43]. Due to the generality and effectiveness of this approach, it has great potential in a variety of related applications [29, 31, 36].

6 CONCLUSION

In this paper, we propose to utilize survival analysis, a statistical learning algorithm commonly used in biostatistics, to effectively compact cold data for LSM-tree based KV stores. This approach fully utilizes the access information of data records, and can better organize the data layout for the heterogeneous storages. To effectively apply survival analysis on cold data prediction, we resolve the difficulty in extracting features, defining censored events, and forming training data with labels from the observed data accesses. To simplify the deployment, we design an external service in conjunction with a lightweight protocol to offload the heavy training and inference operations from the database kernel. We implement our proposal for X-Engine on heterogeneous storage and conduct extensive experiments on typical real-world workloads to demonstrate the superior performance of the new method. The tail latency of the system is decreased by ranging from 31.5% to 78.9%, compared with the state-of-the-art solutions.

REFERENCES

[1] 2022. Alibaba Cloud, Disk overview. https://www.alibabacloud.com/help/doc-detail/25383.htm.

[2] 2022. Aliyun Intelligent stress testing. https://www.alibabacloud.com/help/zh/database-autonomy-service/latest/global-features-intelligent-stress-testing.

[3] 2022. Confusion Matrix. https://en.wikipedia.org/wiki/Confusion_matrix.

[4] 2022. fio. https://github.com/axboe/fio.

[5] 2022. How Much Data Do We Create Every Day? http://bit.ly/2uIeA8Y.

[6] 2022. Introduction to LFU. https://en.wikipedia.org/wiki/Least_frequently_used.

[7] 2022. Introduction to LRU. https://en.wikipedia.org/wiki/Cache_replacement_policiesxw.

[8] 2022. Manual compaction. https://github.com/facebook/rocksdb/wiki/Manual-Compaction.

[9] Raja Appuswamy, Renata Borovica, Goetz Graefe, and Anastasia Ailamaki. 2017. The five minute rule thirty years later and its impact on the storage hierarchy. In *Proceedings of the 7th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*.

[10] Nicola Barbieri, Fabrizio Silvestri, and Mounia Lalmas. 2016. Improving post-click user engagement on native ads via survival analysis. In *Proceedings of the 25th International Conference on World Wide Web*. 761–770.

[11] Renata Borovica-Gajić, Raja Appuswamy, and Anastasia Ailamaki. 2016. Cheap data analytics using cold storage devices. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1029–1040.

[12] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[13] D. R. Cox. 1962. *Renewal Theory*. Methuen Ltd., London.

[14] Frank R Datema, Ana Moya, Peter Krause, Thomas Bäck, Lars Willmes, Ton Langeveld, Robert J Baatenburg de Jong, and Henk M Blom. 2012. Novel head and neck cancer survival analysis approach: random survival forests versus Cox proportional hazards regression. *Head & neck* 34, 1 (2012), 50–58.

[15] Niv Dayan and Stratos Idreos. 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*. 505–520.

[16] Niv Dayan and Stratos Idreos. 2019. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*. 449–466.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[18] Stefan Dietrich, Anna Floegel, Martina Troll, Tilman Kühn, Wolfgang Rathmann, Anette Peters, Disorn Sookthai, Martin von Bergen, Rudolf Kaaks, Jerzy Adamski, Cornelia Prehn, Heiner Boeing, Matthias B Schulze, Thomas Illig, Tobias Pischon, Sven Knüppel, Rui Wang-Sattler, and Dagmar Drogan. 2016. Random Survival Forest in practice: a method for modelling complex metabolomics data in time to event analysis. *International Journal of Epidemiology* 45, 5 (09 2016), 1406–1420.

[19] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[20] John Fox and Sanford Weisberg. 2002. Cox proportional-hazards regression for survival data. *An R and S-PLUS companion to applied regression* 2002 (2002).

[21] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB.

[22] Manish Kumar Goel, Pardeep Khanna, and Jugal Kishore. 2010. Understanding survival analysis: Kaplan-Meier estimate. *International journal of Ayurveda research* 1, 4 (2010), 274.

[23] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–14.

[24] Goetz Graefe. 2009. The five-minute rule 20 years later (and how flash memory changes the rules). *Commun. ACM* 52, 7 (2009), 48–59.

[25] Jim Gray and Goetz Graefe. 1997. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM Sigmod Record* 26, 4 (1997), 63–68.

[26] Jim Gray and Franco Putzolu. 1987. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. 395–398.

[27] Herodotos Herodotou and Elena Kakoulli. 2019. Automating distributed tiered storage management in cluster computing. *arXiv preprint arXiv:1907.02394* (2019).

[28] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 651–665.

[29] Hemant Ishwaran, Udaya B Kogalur, Eugene H Blackstone, and Michael S Lauer. 2008. Random survival forests. *The annals of applied statistics* 2, 3 (2008), 841–860.

[30] Song Jiang and Xiaodong Zhang. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review* 30, 1 (2002), 31–42.

[31] Jared L Katzman, Uri Shaham, Alexander Cloninger, Jonathan Bates, Tingting Jiang, and Yuval Kluger. 2018. DeepSurv: personalized treatment recommender system using a Cox proportional hazards deep neural network. *BMC medical research methodology* 18, 1 (2018), 1–12.

[32] David G Kleinbaum and Mitchel Klein. 2010. *Survival analysis*. Vol. 3. Springer.

[33] Imran Kurt Omurlu, Mevlut Ture, and Füsun Tokatli. 2009. The comparisons of random survival forests and Cox regression analysis with simulation and an application related to breast cancer. *Expert Systems with Applications* 36, 4 (2009), 8582–8588.

[34] Håvard Kvamme, Ørnulf Borgan, and Ida Scheel. 2019. Time-to-event prediction with neural networks and Cox regression. *arXiv preprint arXiv:1907.00825* (2019).

[35] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[36] Changhee Lee, William R Zame, Jinsung Yoon, and Mihaela van der Schaar. 2018. Deephit: A deep learning approach to survival analysis with competing risks. In *Thirty-second AAAI conference on artificial intelligence*.

[37] Yan Li, Vineeth Rakesh, and Chandan K Reddy. 2016. Project success prediction in crowdfunding environments. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*. 247–256.

[38] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*. PMLR, 6237–6247.

[39] A Mendoza. 2013. Cold storage in the cloud: Trends, challenges, and solutions. *Intel, White paper* (2013).

[40] Fen Miao, Yun-Peng Cai, Yuan-Ting Zhang, and Chun-Yue Li. 2015. Is random survival forest an alternative to Cox proportional model on predicting cardiovascular disease?. In *6TH European conference of the international federation for medical and biological engineering*. Springer, 740–743.

[41] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

[42] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[43] Jose Picado, Willis Lang, and Edward C Thayer. 2018. Survivability of cloud databases-factors and prediction. In *Proceedings of the 2018 International Conference on Management of Data*. 811–823.

[44] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[45] Guocong Quan, Kaiyi Ji, and Jian Tan. 2018. LRU Caching with Dependent Competing Requests. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*. 459–467.

[46] Ashwini Raina, Asaf Cidon, Kyle Jamieson, and Michael J Freedman. 2020. PrismDB: Read-aware Log-structured Merge Trees for Heterogeneous Storage. *arXiv preprint arXiv:2008.02352* (2020).

[47] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A tunable delete-aware LSM engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 893–908.

[48] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 217–228.

[49] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. 2020. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 529–544.

[50] HI Strategies. 2015. Tiered storage takes center stage.

[51] Jian Tan, Guocong Quan, Kaiyi Ji, and Ness Shroff. 2018. On Resource Pooling and Separation for LRU Caching. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS'18)* 2, 1, Article 5 (apr 2018), 31 pages.

[52] Jian Tan, Guocong Quan, Kaiyi Ji, and Ness Shroff. 2018. On Resource Pooling and Separation for LRU Caching. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 1, Article 5 (apr 2018), 31 pages.

[53] 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)*. 111–123.

[54] Joaquin Vanschoren. 2018. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548* (2018).

[55] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *Journal of Big data* 3, 1 (2016), 1–40.

[56] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David HC Du. 2020. AC-Key: Adaptive Caching for LSM-based Key-Value Stores. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 603–615.

[57] Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2014. Characterizing Facebook's Memcached Workload. *IEEE Internet Computing* 18, 2 (2014),

41–49.

[58] 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. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1976–1989.

[59] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing storage cost and latency in LSM-tree data stores. In *Proceedings of the ACM Symposium on Cloud Computing*. 162–173.

[60] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. 2020. Fpga-accelerated compactions for lsm-based key-value store. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 225–237.