Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads

DINGHENG MO, Nanyang Technological University, Singapore

FANCHAO CHEN*, Fudan University, China

SIQIANG LUO†, Nanyang Technological University, Singapore

CAIHUA SHAN, Microsoft Research Asia, China

LSM-trees are widely adopted as the storage backends of key-value stores. However, optimizing the system performance under dynamic workloads has not been sufficiently studied in previous work. To fill the gap, we present RusKey, a key-value store with the following new features: (1) RusKey is a first attempt to design LSM-tree structures online to enable robust performance under the context of dynamic workloads; (2) RusKey is the first study to use Reinforcement Learning (RL) to guide LSM-tree transformations; (3) RusKey includes a new LSM-tree design, named FLSM-tree, that facilitates efficient transitions between different compaction policies, which addresses the key bottleneck for dynamic key-value stores. We justify the superiority of the new design with theoretical analysis; (4) RusKey requires no prior workload knowledge for system adjustment, in contrast to state-of-the-art techniques. Experiments show that RusKey exhibits strong performance robustness across diverse workloads, achieving up to 4x better end-to-end performance than the RocksDB system under various settings.

CCS Concepts: • **Information systems → Data management systems**; • **Theory of computation → Data structures design and analysis**.

Additional Key Words and Phrases: LSM-tree, key-value stores, compaction policy, system design, reinforcement learning

1 INTRODUCTION

Log-structured-merge-trees (LSM-trees) are widely adopted as the backbones of mainstream key-value stores, such as LevelDB [24] at Google, RocksDB [21] at Facebook, Cassandra [31] at Apache, Voldemort [35] at LinkedIn, and WiredTiger [13] at MongoDB. An LSM-tree stores data as key-value entries, offering efficient entry updates and key lookups. It organizes the data into multiple levels with exponentially increasing capacities. The data in each level is organized into sorted *runs*. An LSM-tree appends the new key-value entry into the main memory buffer, whose filled-up will

---

Authors' addresses: Dingheng Mo, Nanyang Technological University, , Singapore, ; Fanchao Chen, Fudan University, Shanghai, China, ; Siqiang Luo, Nanyang Technological University, , Singapore, ; Caihua Shan, Microsoft Research Asia, Beijing, China, .

---

sort the buffered entries and compact them as a larger sorted run merged into the next level. The process can cascade down whenever the affected level reaches its capacity. The compaction policy in a level determines how frequently the level compacts its data.

In LSM-tree based key-value stores, one important topic is to handle dynamically changing workloads. As demonstrated by prior studies (e.g., [6, 14, 22, 40]), workloads for real-world applications are dynamically changing with daily patterns and shifts, which underscores the necessity of processing various portions of key lookups and entry updates. For example, Cao *et al.* [10] collected a trace of 14-day operations processed by UDB [5], which is a database storing Facebook's social graph data. They concluded that the numbers and portions of queries and updates vary significantly over time. This is not surprising as modern databases have billions of users who are actively accessing or updating database applications in different time spans. Therefore, in this paper we focus on designing an LSM-tree key-value store that can process dynamic workloads with high performance.

1.1 Prior Work

There has been fast development on LSM-tree optimizations. Behind the ingenuity of an optimized LSM-tree often we see an elaborate design of its key components. Examples include optimizing compaction policies [12, 17, 18, 26, 27, 48], developing update-friendly compaction schemes [3, 20, 47, 49, 59, 61, 62], enhancing Bloom filters [16, 19, 33, 44, 67, 68], enhancing range filters [29, 39, 63], adopting unconventional hardwares [2, 25, 54, 57, 58, 66], narrowing the gap between LSM-trees and update-in-place designs [53], separating keys and values [11, 36], keeping hot entries in the buffer and selective flushing [7], improving concurrency [23, 50], reducing tail latency [8, 37, 49], improving memory management mechanisms [9, 28, 38], and exploiting data characteristics [1, 46, 60]. These works offer promising improvement in various aspects.

Most of the works [1–4, 7–9, 11, 16, 19, 20, 23, 25, 28, 29, 36–39, 46, 47, 49, 50, 54, 57–60, 63, 66–68], unfortunately, are not from the angle of optimizations under dynamic workloads, because their designs are not workload-aware. Recent studies working towards workload-aware LSM-trees lie in tuning the *compaction policy* [12, 17, 18, 26, 27]. Compaction improves the read efficiency in an LSM-tree because it reduces the sorted runs to be probed in each query and cuts down space by eliminating redundant entries. However, compaction itself is a costly process that affects write efficiency. The more aggressive in compaction, the lower read amplification and the higher write amplification. Based on these observations, Dayan and Idreos [17] further point out that a more frequent compaction optimizes the lookup cost but increases the update cost, and hence, various customized compaction policies based on workload characteristics are proposed to improve the overall performance of the LSM-tree.

There are still several deficiencies in these closest works when applied to dynamic workloads. Most of them [12, 17, 27] compute an optimized system setting when given a specific application workload. For example, Cosine [12] and Dostoevisky [17] require users to input a workload with a specific ratio between lookups and updates, so that they can model the expected performance of various possible compaction policies, and choose the best one. While these models are very promising in analyzing a workload-aware system, it remains open regarding how to design reasonable key-value stores when the workload is not known a priori. One possibility is to combine them with workload forecasters such as those designed by Ma *et al.* [40], but this incurs an additional forecaster training cost. More importantly, even though the additional workload forecaster may be integrated, the obstacle is in efficient adaptation of the underlying LSM-tree from one state to another in response to workload changes – the main bottleneck of designing dynamic LSM-trees. For example, changing a compaction policy at a level usually requires resizing the runs and remapping each data block to the runs, which incurs a high cost. This issue has been largely overlooked in the

literature and the only known strategy is a greedy transition mentioned by Dayan and Idreos [17] as an extended discussion. Nonetheless, this strategy is sub-optimal as we discuss in Section 4.3.

1.2 The Problem and Our Solution

We aim to design an LSM-tree based key-value store that can automatically adapt itself in an online manner to continuously changing workloads while keeping a robust end-to-end performance (i.e., close to the performance of the optimal setting in the hindsight). In contrast to prior works on autonomous DBMS, our focus on NoSQL key-value stores (particularly, LSM-trees) brings in drastically different technical challenges. We thus explore new ways of modeling key-value stores so as to achieve our goal. Below we give a preview of our new ideas.

Modeling Key-Value Stores Using Reinforcement Learning. We model (See Section 3 and Section 5) the LSM-tree based key-value store as a reinforcement learning process [51, 52], which is a machine learning approach that trains a model with an action-reward series in an online manner. We remark that most existing methods for tuning compaction policy [12, 17, 18, 26, 27] use a different way in modeling, which captures the relationship between compaction policy and LSM-tree performance via a sophisticated formula, referred to as a white-box cost model. While these models show promising results in many aspects, there are limitations in modeling the end-to-end system performance. First, white-box models often focus on I/O complexity analysis to predict the I/Os incurred by each lookup or update, neglecting the CPU overheads that can be significant. For example, Zhu *et al.* [68] reported that the hashing of Bloom filters in LSM-trees may incur high CPU overhead that even exceeds I/O overhead with modern storage hardware. Neglecting the CPU overhead is therefore a source of inaccuracy. Second, modern systems are extremely sophisticated in design, and are often cumbersome, if not unpromising, to be unfolded for detailed complexity analysis of internal components. For example, memory cache can significantly affect the performance, but white-box formulas are often unable to model such bottom-level details.

In contrast to white-box formula based models, models based on RL for tuning treat the system as a black-box, where *action* modifies the LSM-tree compaction policies and *reward* traces system performance. The tuning is automatic in a machine-learning style. Modeling the process with RL is not trivial, as discussed in Section 5.

It is also important to note that we do not aim to completely replace classic white-box models which are still helpful in many aspects. Instead, our main purpose is to explore the potential of applying RL in tuning LSM-tree based systems, which complements the existing classic methods and, possibly, opens the new research direction for machine-learning-based LSM-tree tuning.

Designing Transition-Friendly LSM-trees for Reinforcement Learning. When dynamically tuning LSM-trees, one critical issue is maintaining an efficient transformation of LSM-trees, as a slow transformation may outweigh the benefit of a compaction policy adjustment. This issue motivates us to design FLSM-TREE (See Section 4), a flexible LSM-tree that allows various-sized runs in a level. When the compaction policy changes, FLSM-TREE avoids the modification of most of the data in the affected level. We also provide a theoretical analysis showing that FLSM-TREE promises zero delay, free immediate transition costs, as well as less implicit additional costs.

Integrating White-box Models in Reinforcement Learning. Unlike RL in other typical machine learning scenarios, it is unpromising to gather a large amount of training samples in a key-value store because the samples are obtained by running costly lookup and update operations. To circumvent the problem, we propose a new *level-based* training strategy, where the compaction policies of smaller levels are learned by RL while those of large levels are extended by the lower-level policies through a careful complexity analysis (i.e., a white-box model). This technique largely reduces the number of samples needed to train the model, empowering the tuning ability of RL.

Putting together, we develop the RusKey system, a flexible LSM-tree-based key-value storage system that is self-adaptive to dynamic workloads. We summarize our contributions as follows.

- We make the first attempt at designing RL models to optimize the performance of LSM-tree based key-value systems. We design and implement RusKey which is a key-value system that demonstrates strong robustness in end-to-end performance via an RL-based tuning model. RusKey can efficiently adapt itself to various workloads, based on the instructions of the learning model.

- We address an open problem in dynamic key-value systems – the efficient and transient policy transformation in LSM-tree under dynamic workloads. We present FLSM-TREE, a novel LSM-tree structure that incorporates a flexible compaction policy transition scheme, which swiftly transforms the compaction policy of an LSM-tree with minor additional overhead.

- We present a new level-based RL model which is embedded with a policy propagation formula to reduce the number of samples required, shrink the action space and improve the convergence speed in our RusKey system.

- We conduct extensive experimental evaluations of RusKey in real system environments. We demonstrate that RusKey achieves high performance in all the tested workloads. RusKey achieves up to 4x throughput compared with the baselines.

2 BACKGROUND

LSM-tree. The log-structured merge tree (LSM-tree) [42] is a persistent multi-level index structure that stores key-value entries. When a key-value entry is inserted, an LSM-tree buffers it in the main memory. Once the buffer is filled up, it sorts the entries by keys and flushes them into a disk-resident sorted run in Level 1. When the size of the disk runs in a level exceeds a certain threshold, the LSM-tree sort-merges them into a larger run and puts it into the next level. This process of sort-merging multiple smaller sorted runs in Level i to a larger sorted run in Level $i + 1$ is called *compaction*. As such, the capacity of Level $i + 1$ is T times larger than that of Level i, where T is the capacity ratio.

Lookup (Query) in LSM-tree. In modern key-value systems, LSM-tree is coupled with Bloom filters and fence pointers to improve the key lookup performance. The Bloom filter and fence pointer are built for each sorted run and will be loaded into the main memory when the system starts. To check whether a key exists in a run, the Bloom filter associated with the run is first probed. If it returns false, then the run does not contain the key and no I/O cost will be incurred. Otherwise, accessing the run in the disk is necessary, which incurs $O(1)$ I/Os guaranteed by fence pointers. In general, the Bloom filter largely prevents unnecessary disk reads. The I/O-cost reduced

Notation	Description
T	Capacity ratio between adjacent levels
K_i	Compaction policy of the i-th level
E	Size of a key-value entry (in bytes)
B	Size of a disk page (in bytes)
C_i	Capacity of the i-th level (in bytes)
f_i	False positive rate of Bloom filters at the i-th level
D_i	Total data size in the i-th level (in bytes)

Table 1. Frequently used notations.

by the Bloom filter depends on its FPR (False-Positive Rate). In particular, if the run does not contain the queried key, its Bloom filter may still mistakenly return true for the key, incurring a false positive. While mainstream designs uniformly set the FPRs for Bloom filters at all LSM-tree levels, state-of-the-art techniques [12, 16–18, 26] employ the Monkey allocation scheme [16], which arranges exponentially higher FPRs to Bloom filters in larger levels.

Compaction Policy. The compaction policy K_i at each Level i can be independent, as presented in Dostoevsky [17]. The policy K_i means that there are at most K_i sorted runs in the level. Level i has an active run that admits the merged run from Level $i - 1$. The capacity for each active run is $1/K_i$ of the level capacity. All entries in a level are eventually merged and flushed down to the next level when the level reaches its capacity. Dostoevsky proposes Lazy-Leveling which applies an aggressive compaction policy of at most 1 sorted run in the largest level while all other levels use a lazy compaction policy in common. They give a way of applying different compaction policies at different levels, but within the same level the compaction policy is always the same. Our framework give a significant extension by allowing various run sizes in a level.

Amplification. Read amplification and write amplification are commonly used to measure the performance of an LSM-tree. Read amplification refers to the average number of disk pages read during a lookup, since a query may probe multiple sorted runs in an LSM-tree to search for the queried key. Write amplification refers to the average number of physical rewrites that an entry incurs, because each data entry is constantly rewritten to storage as a result of the LSM-tree's compaction.

3 RUSKEY OVERVIEW

This section presents RusKey, a key-value system empowered with deep reinforcement learning. The purpose of RusKey is to tune the compaction policy in each LSM-tree level to best process the workload. There is no doubt that a fixed policy is still possible to be applied under dynamic workloads, but as we have mentioned, changing policies can bring in benefits as long as the LSM-tree structure transformation cost is small. RusKey includes two major designs: (1) A level-based deep reinforcement learning tuning model, named **Lerp** and (2) A new flexible LSM-tree with new compaction transition method, named **FLSM-tree**. As we will explain shortly, Lerp incorporates the cost analysis across levels for fast convergence and less sample-demanding training. FLSM-tree embeds flexible transition to avoid high transition cost, hence being more cost-friendly to a compaction policy tuner like Lerp. We will first describe the general workload of RusKey in this section, and then discuss the detailed mechanism of FLSM-tree and Lerp in Section 4 and Section 5. Table 1 lists the frequently used notations.

3.1 System Workflow

In RusKey, the *application workload* is a series of *lookups* or *entry updates* operations input to the database. A lookup finds the corresponding value for a given key, and an update inserts or deletes a key-value entry. RusKey allows for arbitrary arrivals of operations, which means that knowledge of the query/update ratio is not known in advance. A *mission* in RusKey is a collection of operations (e.g., lookups or updates) in the workload. Conceptually, the workload is divided into missions. After processing each mission, RusKey may adjust its internal LSM-tree structure for the coming missions. RusKey also maintains a *statistics collector* that keeps track of necessary statistics of RusKey and application workload over time. Besides overall statistics of the FLSM-tree, it tracks statistics separately for each FLSM-tree level to support the level-based training scheme in Lerp. It also collects the operation composition in each mission for detecting changes in the application workload.
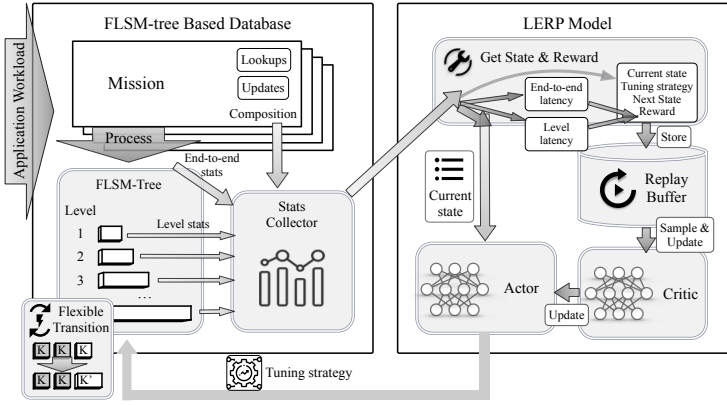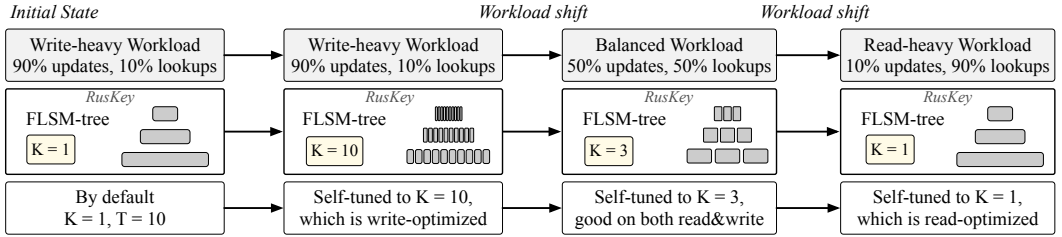
Fig. 1. The components and workflow of RusKey.

Fig. 2. Running example of RusKey. This is a simplified example where we assume that each level has the same compaction policy. In fact, RusKey allows the FLSM-tree to set various compaction policies across different levels.

Figure 1 illustrates the workflow of RusKey. Suppose a real-world application employs RusKey as its data storage backend. RusKey processes operations from the application workload with the FLSM-tree and uses the tuning model Lerp to adjust the compaction policy. Lerp and FLSM-tree interact after each mission. At the end of a mission, Lerp gives tuning suggestions to the FLSM-tree and updates the actor-critic network. Meanwhile, the statistics collector sends statistics of FLSM-tree in this mission to Lerp, which extracts experience samples from the statistics and stores them in the replay buffer. The replay buffer stores some quadruples (called experience samples), each of which contains the essential information for reinforcement training. The information includes the action of policy change, the FLSM-tree state before the action, the FLSM-tree state after the action, and the subsequent mission, as well as the reward of the action. Through learning from these samples, Lerp effectively updates its internal actor-critic network [34]. Then Lerp selects a suitable tuning strategy for the compaction policy of the FLSM-tree in the next mission according to the current state of the FLSM-tree. RusKey then updates the FLSM-tree based on the suggested compaction policy with the flexible transition. Under a stable application workload, the actor-critic network converges in a number of missions, shifting FLSM-tree towards the optimal compaction policy. Once the application workload changes, the actor-critic network is no longer in convergence, and Lerp will restart to exploit compaction policies under the new workload.

Running Example. Figure 2 shows a running example of RusKey, which is simplified by assuming each level has the same compaction policy. Initially, the compaction policy is set as 1, and the capacity ratio is 10. The dynamic workload is write-heavy at first with 90% of updates and 10% of lookups. It then shifts into a read-write-balanced workload (50% of updates and 50% of lookups), followed by a read-heavy workload (10% of updates and 90% of lookups). At the beginning, RusKey tunes the compaction policy of the FLSM-tree from $K = 1$ to $K = 10$ to achieve the best write

performance under write-heavy workloads, and we note that during the tuning the system is not fed with the workload statistics. With the benefit of flexible transition, the FLSM-TREE could transform its policy $K = 1$ to $K = 10$ without a transition cost and delay. Once the workload shifts to the balanced one, RusKey will re-tune the compaction policy of the FLSM-TREE from $K = 10$ to $K = 3$, which is more effective in a balanced workload. Similarly, when the workload shifts to read-heavy, RusKey will adjust its compaction policy from $K = 3$ to $K = 1$ to achieve an optimized read efficiency.

4 FLSM-TREE IN RUSKEY

This section discusses two baseline transition methods (Section 4.1), followed by presenting our design of FLSM-TREE (Section 4.2) and its desired properties (Section 4.3).

4.1 Greedy and Lazy Transitions

RusKey changes the underlying LSM-tree based on the suggested compaction policy by the tuning model. However, in the classic LSM-tree, continuously changing the compaction policy demands the reallocation of memory and potentially reorganizes the data in the LSM-tree. Enforcing a change of compaction policy at a level can have two straightforward solutions, namely, *greedy transition* and *lazy transition*. However, these solutions are not ideal.

Greedy Transition [17]. Once a level receives a tuning strategy, it merges and flushes all its data into its next level. The current level then needs to be rebuilt based on the new compaction policy, as shown in Figure 3 (top). We note that such a greedy transition is not effective due to its dreadful cost and the write stall incurred during data flush, and hence significantly degrades the performance. Moreover, if an unfilled level is merged with the next level, it incurs a larger write amplification than an LSM-tree without transitions.

Lazy Transition. The other extreme is the lazy transition that an LSM-tree level changes its compaction policy only after it is merged to the next level, as shown in Figure 3 (bottom). Although such a lazy transition will not bring an immediate compaction cost, it affects the timeliness of the information fed into the RL model. To explain, a given (policy change) action may have no effect on the LSM-tree level until the level eventually becomes full and gets merged, and the delay can be long at a large level, ultimately reducing the effectiveness of the model.

4.2 Main Design of FLSM-TREE

We design a **F**lexible **LSM**-tree, abbreviated as FLSM-TREE, to accommodate a flexible transition between compaction policies. A novel design in FLSM-TREE is that it allows different-sized runs to exist at the same level. This extends the classic LSM-tree structure, allowing more room for an LSM-tree to morph the structure gradually. At each level, there is an active run that can admit
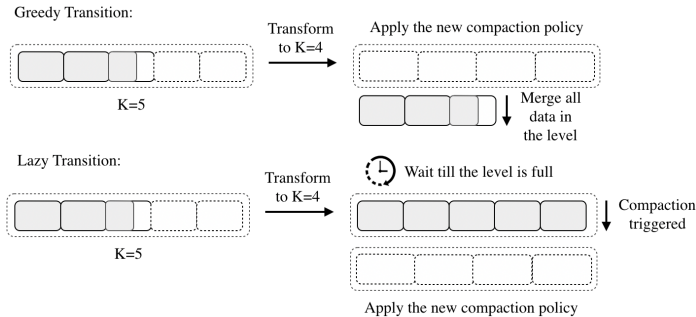

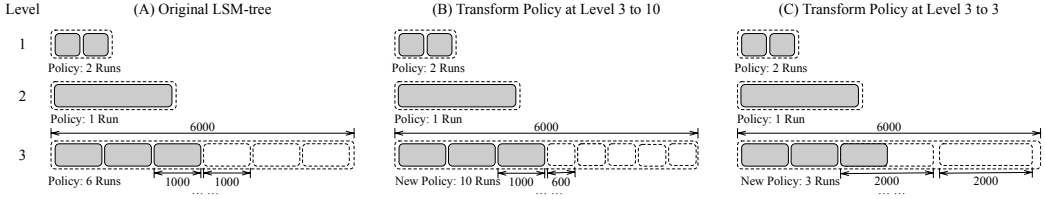Fig. 3. Greedy and lazy transitions of compaction policy.

Fig. 4. Flexible compaction policy transition in a FLSM-tree.

Transition Alg.		Greedy	Lazy	FLSM-TREE (ours)
Transition Cost		$\frac{C}{2B}$	0	0
Delay (secs)		0	$\frac{C}{2N_uE}$	0
Additional	$K < K'$	$\frac{TC(1-x)}{2BK}$	$\frac{TC(1-x)(K'-K)}{2BKK'}$	0
Cost	$K > K'$	$\frac{TC(1-x)}{2BK}$	$\frac{fC(1-x^2)(K-K')\gamma}{2E(1-\gamma)}$	$\frac{fC(x-x^2)(K-K')\gamma}{E(1-\gamma)}$
Case Study		125 I/Os	3.75 I/Os	2.5 I/Os

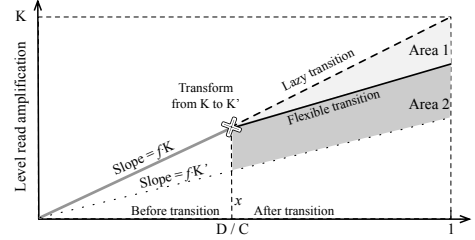Table 2. Transition costs and delays.

Fig. 5. Flexible transition incurs an additional cost = Area 2; lazy transition incurs an additional cost = Area 1 + Area 2.

more key-value entries. When the active run in the level reaches its capacity, it is sealed, and a new active run is created. When a new policy is applied on a level, it never affects the runs that have been sealed, and only changes the capacity of the active run and the runs to be formed later. In this way, the change of the policy is immediate and the LERP model gets the correct reward by applying the policy changes without incurring any immediate transition costs.

Let the capacity of Level i be C_i, the new compaction policy be K'_i, and the original compaction policy of Level i is K_i. The discussion is divided into two cases.

If $K'_i < K_i$, FLSM-TREE will increase the capacity of the active run, keeping the other runs intact.

If $K'_i > K_i$, FLSM-TREE will decrease the capacity of the active run. One may have a concern that when the new policy is known, the active run size can already be larger than the expected size C_i/K'_i. In this case, we immediately seal the active run and create a new active run. The runs, when sealed, remain untouched until the level holding them reaches its capacity and triggers a compaction during which all runs will be merged to the next level.

Example. Figure 4 presents examples of transforming compaction policy in FLSM-TREE. A grey square represents a sorted run filled with data, and the dashed line denotes logical capacity of a level or an incoming run. In Figure 4 (A), the compaction policy of Level 1 is 2. Level 1 includes two runs, each having a size that is half of the level capacity. The compaction policy of Level 2 is 1, rendering a single run in the level. For Level 3, initially the compaction policy is 6 and the level capacity is 6000, giving an expected run size 6000/6=1000. If we transform the policy of Level 3 to 10, it will turn to the state of Figure 4 (B), such that the expected run size under the new compaction policy would not affect existing sealed runs, but the expected size of an incoming run decreases to 600 (=6000/10). If we transform the policy to 3, it will turn to the state of Figure 4 (C). The expected size of an incoming run becomes 6000/3=2000, and the current active run size also increases to 2000.

4.3 Desired Properties of FLSM-TREE

To check the advantages of our FLSM-TREE design, we focus on the following measures: (1) *Transition Cost*, which is the immediate compaction cost incurred by a transition. We aim at small immediate

transition cost to avoid serious performance glitches; (2) *Delay*, which is the time span between the new policy's arriving at the system and the time at which the transition actually takes effect on the LSM-tree; (3) *Additional Cost*, which is the total additional I/O cost incurred by a transition, as compared to an ideal LSM-tree born with the new compaction policy.

We illustrate the concept of *additional cost* incurred by transition using Figure 4 (C) as an example. After transforming the policy in Level 3 from 6 to 3, the incoming runs will follow the new compaction policy 3, while the existing sealed runs stay with the previous compaction policy 6. In this instance, there will be at most 4 runs in Level 3 after the flexible transition, whereas the new compaction policy indicates 3 runs in the level, such that accessing the level would incur more I/Os than expected. Here we only discuss the additional cost, and hence the cost is lower bounded by 0. We note that sometimes additional benefits may be involved, for which we will have a further discussion in the *overall comparison* paragraph at the end of this section.

Notations. To facilitate our discussion, we first introduce the necessary notations for the analysis in this sub-section. We fix an LSM-tree level whose capacity (in bytes) is C. Let D be the total data amount (in bytes) of key-value entries currently stored in the level. Let B be the size (in bytes) of a disk page. Let E be the size (in bytes) of a key-value entry. Let N_u be the number of updates arriving per second. Let x be the percentage of the stored data in the current level when a compaction policy transition is applied. Let f be the FPR of the Bloom filter in the level. Let γ denote the percentage of lookups in the workload.

FLSM-tree incurs zero transition cost and delay. Our FLSM-tree has the minimum transition cost and delay cost. The greedy transition merges all data in that level to the next level. This process incurs $\frac{D}{B}$ I/Os. When a transition occurs, D may be any random value between 0 and C, leading to an amortized I/O cost $\frac{C}{2B}$ for a greedy transition. In contrast, the transition cost of a lazy transition is always zero. As for the delay, in the lazy transition, the new compaction policy takes effect only after the level is full and a whole-level compaction is triggered. The delay can be estimated by $\frac{C-D}{N_u E}$ seconds. The amortized delay of the lazy transition is $\frac{C}{2N_u E}$ considering D ranges from 0 to C. Regarding the transition mechanism in a FLSM-tree, the transition cost is always zero, since it only changes the metadata of the active run and does not incur any disk reads. Meanwhile, the flexible transition in a FLSM-tree comes into effect immediately with zero delay[1]. Table 2 summarizes the transition cost and delay of the three transition methods.

FLSM-tree incurs less additional cost. FLSM-tree also incurs the minimum additional costs compared with baselines. We first examine the additional cost of each transition method.

Additional cost of the greedy transition. A greedy transition compacts the level right after the policy transition, giving zero additional read cost. However, this compaction suffers a larger write amplification because it merges fewer data into the next level than a normal full-level compaction.

We now analyze the additional write amplification incurred by a greedy transition. Let us first consider a normal compaction without policy changes. Without loss of generality, we assume that the subsequent level's compaction policy is K. In expectation, the size of the active run in the subsequent level is $\frac{TC}{2K}$. A compaction reads expected $\frac{TC}{2K}$ bytes of data and rewrites them into the disk in order to merge C bytes of data (i.e., the full current level) into the subsequent level. Hence, the active run size $\frac{TC}{2K}$ divided by C, which is $\frac{T}{2K}$, is the write amplification of the current level. The additional cost caused by the greedy transition is $\frac{T}{2K \cdot x}$ because it only merges data that is x times the level capacity, which leads to $\frac{T}{2K \cdot x} - \frac{T}{2K}$ additional write amplifications.

---

[1]Zero-delay means the LSM-tree immediately reacts when policy changes, marking that upcoming updates will follow the new policy. We note that the policy changes will not have any effect for the extreme case where the workload only contains reads. In this case, flexible transition simply degenerates into lazy transition and will not bring additional negative impact.

The expected additional I/O cost is computed by multiplying the additional write amplification with the number of pages merged by the greedy transition into the subsequent level:

$$Cost_{greedy} = \left(\frac{T}{2K \cdot x} - \frac{T}{2K} \right) \cdot \frac{C \cdot x}{B} = \frac{TC(1-x)}{2BK} \tag{1}$$

Additional cost of lazy transition. If $K > K'$, lazy transition avoids the immediate transition cost and the additional cost from write amplification, at the expense of incurring additional read cost.

We illustrate this expense in Figure 5, where the horizontal-axis refers to the level's filling ratio, which is the percentage of data stored at the level among the capacity of the level, i.e., D/C. The vertical-axis refers to the read amplification of the level, which is the expected number of disk pages a lookup needs to probe in the level. In a process of inserting data from an empty level to a full level, the filling ratio increases from 0 to 1. In expectation, the read amplification of a level is the product of three values – its false positive rate f, its compaction policy K, and its filling ratio. To explain, when the level is full, there are K runs in the level to be probed during each query and each run probe requires a disk I/O only when its Bloom filter returns positive. Therefore, the read amplification can be expressed by $f \cdot K \cdot \frac{D}{C}$.

As shown in Figure 5, after the transition, the read amplification of the level is higher than it should be under the more aggressive compaction policy K' (which is represented by the dotted line). This is because the level still adheres to the foregoing K compaction policy with lazy transition. This lasts until the level is full, at which point all data is merged into the following level via compaction. Therefore, the sum of Area 1 and Area 2 in Figure 5 represents the overall additional read cost of a lazy transition. A lazy transition's additional cost in I/Os can be expressed by Equation 2.

$$Cost_{lazy} = \frac{fC(1-x^2)(K-K')\gamma}{2E(1-\gamma)} \tag{2}$$

If $K < K'$, a lazy transition incurs an additional write cost. Upon changing the policy, the expected write amplification for this level becomes $\frac{T}{2K'}$. However, until the level becomes full and triggers compaction, the actual write amplification of the lazy transition remains $\frac{T}{2K}$. Throughout this period, this level will merge a total of $\frac{(1-x)C}{B}$ blocks, incurring an additional cost of $\frac{TC(1-x)(K'-K)}{2BKK'}$.

Additional cost of a flexible transition. The additional cost of a flexible transition can be derived in a similar way to a lazy transition. Before changing a compaction policy, the growth rate of read amplification equals the current compaction policy K. When a new compaction policy K' is applied, different transition methods bring in distinctions. With a lazy transition, the growth rate of read amplification stays the same after the change of the policy. With a flexible transition, the growth rate reduces from $f \cdot K$ to $f \cdot K'$. Similar to a lazy transition, a flexible transition also incurs additional read amplification when the level is full. This is because part of the data in this level is inserted before the transition, where the inserted data adheres to the foregoing compaction policy K. Therefore, Area 2 in Figure 5 represents the overall additional cost of flexible transition, which is

$$Cost_{flexible} = \frac{fC(1-x)(K-K')\gamma}{E(1-\gamma)} \tag{3}$$

Overall Comparison. Table 2 summarizes the additional cost of different transition methods. It is obvious that the additional cost of lazy transition is always larger than flexible transition. For comparison between flexible transition and greedy transition, it is obvious that greedy transition is always more costly when $K < K'$ because the additional cost of a flexible transition is zero in this case. When $K > K'$, by removing the constant coefficients in cost functions (Equations 2, 3), the additional costs of a flexible transition and a greedy transition are $O(\frac{fC}{E})$ and $O(\frac{C}{B})$, respectively.

Note that the FPR f in an LSM-tree is often less than 1%, and the entry size E is usually in the same magnitude as the disk page size B. Therefore, the cost of a flexible transition can be up to two orders of magnitude smaller than that of a greedy transition.

We give an example in real scenarios for ease of comparison. We let $x = 1/2$ and $\gamma = 1/2$ since x and γ are both distributed between 0 and 1. We follow the common settings in key-value stores and set $T = 10$, $B = 4096$, $E = 1024$, $C = 1024000$, and $f = 0.01$. Finally, setting $K = 5$ and $K' = 4$ gives us $\frac{TC(1-x)}{2BK} = 125$, $\frac{fC(1-x^2)(K-K')\gamma}{2E(1-\gamma)} = 3.75$, and $\frac{fC(x-x^2)(K-K')\gamma}{E(1-\gamma)} = 2.5$. Hence, a flexible transition incurs a much smaller additional cost under this scenario.

Furthermore, if we consider the overall benefit that takes into account both additional costs and additional benefits, flexible transition in general still has a greater benefit than that of lazy transition, assuming that a newly learned policy outperforms the previous one in terms of current workload and has lower latency. Since flexible transition does not involve any delay compared to lazy transition, it allows for a prompt transformation to the new and improved policy, leading to earlier benefits.

5 LERP: THE TUNING MODEL IN RUSKEY

Reinforcement learning (RL) is a machine learning paradigm that trains an agent to cognize and decipher its environment. The agent gets reward or penalty for each action it takes, and decides the next action based on previous experience. The goal of RL is to improve an agent's performance by learning the optimal policy that maximizes the expected cumulative reward.

It is not a trivial task to design a suitable RL model for tuning an LSM-tree-based key-value system. First, it remains unexplored how to relate the key-value system properties such as LSM-tree compaction policy, the running time cost, the level properties, and the workload characteristics to the fundamental RL elements such as states, actions, and rewards. The model should also avoid over-amplifying the action space and state space. Second, to make RusKey work in an online manner under dynamic workloads, the model should converge fast. Unfortunately, the high data volume in key-value systems makes the deeper LSM-tree levels experience much longer time to be affected. As a result, the corresponding feedback to the RL model is significantly delayed. Considering that RL requires sufficient feedback from the actual environment to adjust its policy, efficiently and effectively learning the policy for larger levels is notorious challenging. To address the challenge, we design a novel **Le**vel-based **R**einforcement learning model with policy **P**ropagation, abbreviated as LERP. Next, we introduce the level-based model of LERP in Section 5.1, and policy propagation techniques in Section 5.2.

5.1 The Level-based Model

We model the *reward*, *state*, *environment*, and *action* to capture the system process, and then apply a RL procedure.

5.1.1 State and Environment. The *state* captures the parameters related to the FLSM-TREE and the workload within a mission. Our model state consists of internal statistics of the LSM-tree, such as the number of read and write I/Os, the level capacities, and the current compaction policies at each level. It also includes workload statistics such as the read/write ratio in the previous mission. These statistics effectively capture the system state which is related to the system performance. We model the *environment* by the compaction policy of a level. For Level i, its compaction policy K_i is defined as the maximum number of sorted runs in the level, and obviously K_i needs to be in $[1, T]$. A smaller K_i means a more aggressive compaction policy. Particularly, $K_i = 1$ implies a leveling compaction policy, and $K_i = T$ implies a tiering compaction policy.

5.1.2 Action Space: From $O(2^L)$ to $O(L)$. The *Action* changes the compaction policy of an FLSM-TREE. A compaction policy setting of an L-level FLSM-TREE is a list $(K_1, K_2, \ldots, K_L)$, where K_i refers to the policy of Level i. Intuitively, an action can change a policy setting $\mathbf{V} = (K_1, K_2, ..., K_L)$ to another $\mathbf{V}' = (K_1', K_2', ..., K_L')$, giving a massive action space of size $O(T^L)$, where T indicates the number of policies in a level. To reduce the action space, our main idea is two-fold. First, the compaction policy is independent among levels, and hence the training can be done separately for each level. Second, the characteristics of real-world workloads often change *gradually* over time. This hints that the policy should also be roughly *continuous*, and a drastic change of the policy may not be preferred.

Therefore, we design a *level-based* model that trains an independent RL model for each level, where the reward, state, and action are only tailored to the corresponding level. Such level-decoupled models have drastically smaller action and state space, and hence easier to train. To explain, the action space in the model at each level (or level model) is only $O(T)$, and the total action space is $O(TL)$. Furthermore, we only allow continuous change of the policy in level models, i.e., the next action for policy K_i can only be one of $K_i - 1$, K_i, or $K_i + 1$. This further reduces the action space to $O(L)$.

5.1.3 Reward. In RL, the reward function provides feedback from the environment indicating the correctness of the actions that have been taken. In RusKey, the reward function is obliged to reflect the performance of an FLSM-TREE level. Our reward function combines both the end-to-end system latency and a specific level-based latency. The former reflects the global impact while the latter reflects the local impact. Let t_i denote the level-based latency of the i-th level and t' denote the end-to-end latency of FLSM-TREE. Lerp's reward function on Level i is $\alpha \cdot t_i + (1 - \alpha) \cdot t'$. Where α is a parameter between 0 and 1, which controls the weight of t_i and t'. We do not deliberately set different weights to read latency and write latency, because the relative speed of reads and writes is reflected in the reward (end-to-end latency), and automatically learnt in RL.

5.1.4 Incorporating Actor-Critic Network. With the fundamental mapping between the major RL elements and FLSM-TREE, we now discuss the specific design of the RL model. Note that our level-based design can be applied with different RL models, and we select the Deep Deterministic Policy Gradient (DDPG) model [34], which has been shown to be more effective compared with the classic models such as DQN [41].

DDPG integrates DQN with the actor-critic network from DPG (Deterministic Policy Gradient). The actor-critic network can be regarded as a temporal difference variation of the policy gradient. It parameterizes a critic function and an actor function with neural networks. The critic function maps the current state and action taken into a Q-Value, which is an estimation of the expected reward. The actor function maps the state into a probabilistic distribution of available actions along the direction suggested by the critic function. The policy gradient technique is the foundation for the actor's learning. Comparatively, critics assess the actor's performance by computing the value function. We refer interested readers to the original paper [34] for more details of the model.

Under the Lerp model, actors are in charge of selecting the best compaction strategy, and critics assist in assessing the effectiveness of a compaction strategy. In addition, each level has its own actor and critic functions with the level-based setting. Based on this idea, it is natural to have DDPG-based training steps for each level, while we take the level state of the previous round as input.

5.2 Optimize Training with Policy Propagation

The training data of the RL model comes from system runtime metrics. For larger levels in the FLSM-TREE, training data is sparse because the compaction in a deeper level is exponentially less

frequent than at a shallower level. Nonetheless, the size of a deeper level is exponentially larger than a shallower level and hence needs more training data. This gives rise to a crucial question: how to effectively learn the compaction policy at a large level while keeping a fast learning convergence?

To address this issue, we present a *policy propagation* method based on a cost analysis. As the query cost analysis is related to the bits-per-key at Bloom filters in each level, discussions are divided divided into two cases:

- **Case 1: Uniform Bits-Per-Key:** Every level assigns the same bits-per-key to the Bloom filter. This is the default setting in many key-value stores such as RocksDB [21].
- **Case 2: Monkey Allocation:** Level i assigns exponentially lower bits-per-key than Level $i + 1$. It is shown in [16] that this scheme achieves better amortized read performance, and has been adopted in other advanced systems such as Dostoevisky [17] and Cosine [12].

5.2.1 Propagation for Case 1. In this case, the read/write cost ratio is very similar across different levels, for the reason that each level has the same read amplification and write amplification. Therefore, we can use the RL model to learn the policy of Level 1 and propagate the policy to the other levels. Specifically, we employ the per-level model to obtain the optimal policy at Level 1. The compaction policies of other levels remain unchanged throughout this process so as not to interfere with the end-to-end latency which contributes to the reward. Then, we transform the compaction policies at all other levels into the policy we learned at Level 1 with the flexible transition. It should be noted that our previous comments about the potential for independent policy settings at each level still stand. Our earlier mention of independence referred specifically to the design space of the LSM-tree and did not necessarily contradict the possibility of level correlation within an optimal scheme.

5.2.2 Propagation for Case 2. In this case, the read/write cost ratio can be drastically different, and hence the policy at each level can vary. Under the Monkey scheme, if we intend to optimize the expected cost of an operation (lookup or update), there is an optimal policy. We mainly focus on analyzing the zero-result case following Dostoevsky [17]. We apply the following lemma to guide us in making appropriate inferences on compaction policies.

LEMMA 5.1. *Consider that the three consecutive levels $(i - 1, i, i + 1)$ in a FLSM-TREE employ the Monkey scheme. Let K_i^* be the optimal merge policy that minimizes the expected cost per operation at Level i. We have the following equation:*

$$\frac{1}{K_{i+1}^*} = \sqrt{\frac{1}{K_i^{*2}} + T \cdot \left(\frac{1}{K_i^{*2}} - \frac{1}{K_{i-1}^{*2}} \right)} \tag{4}$$

PROOF. We follow the notations in Section 4.3 (e.g., K_i, B, E, γ) unless otherwise specified.
(1) <u>Lookup cost.</u> Let f_i denote the FPR of the Bloom filters in Level i. In expectation, the lookup will incur $f_i \cdot K_i$ false positives in Level i in the zero-result case. As each false positive causes one disk page read, the time cost that a lookup causes in Level i is $f_i \cdot I_r \cdot K_i$, where I_r is the average time of a read I/O. Let c_r be the cost of probing the metadata of a sorted run in the main memory. Then, the total lookup cost in Level i is $f_i \cdot I_r \cdot K_i + c_r \cdot K_i$.
(2) <u>Update cost.</u> Let I_w denote the average time cost of a write I/O. According to [27], the write amplification of Level i is T/K_i, signaling that on average the key-value entry of the update will take part in T/K_i compactions in Level i. This entry is read into memory and written into the disk once at each compaction. Therefore, the update incurs an overhead of $\frac{T \cdot E}{B \cdot K_i} \cdot (I_r + I_w) + \frac{T}{K_i} \cdot c_w$, where c_w is the CPU cost a key incurs in compaction (such as merge sorting and space allocation).

Then, the expected time overhead per operation in Level i is

$$\underbrace{f_i \cdot I_r \cdot K_i \cdot \gamma}_{\text{Query I/O cost}} + \underbrace{c_r \cdot K_i \cdot \gamma}_{\text{Query CPU cost}} + \underbrace{\frac{T \cdot E}{B \cdot K_i} \cdot (I_r + I_w) \cdot (1 - \gamma)}_{\text{Update I/O cost}} + \underbrace{\frac{T}{K_i} \cdot c_w \cdot (1 - \gamma)}_{\text{Update CPU cost}} \tag{5}$$

Monkey requires $f_i = T^{i-1} f_1$, where f_1 is the FPR of the Bloom filters at the first level. Applying the method of Lagrange multipliers, the time overhead is minimized when

$$K_i^{*2} = \frac{T \cdot E \cdot (I_r + I_w) \cdot (1 - \gamma) + B \cdot T \cdot c_w \cdot (1 - \gamma)}{B \cdot f_1 \cdot I_r \cdot T^{i-1} \cdot \gamma + B \cdot c_r \cdot \gamma}$$

Or equivalently,

$$K_i^{*2} = \frac{X}{Y \cdot T^{i-1} + Z}$$

where $X = T \cdot E \cdot (I_r + I_w) \cdot (1 - \gamma) + T \cdot B \cdot c_w \cdot (1 - \gamma)$, $Y = B \cdot f_1 \cdot I_r \cdot \gamma$, and $Z = B \cdot c_r \cdot \gamma$. This further gives

$$\frac{1}{K_{i+1}^{*2}} - \frac{1}{K_i^{*2}} = \frac{Y \cdot (T^i - T^{i-1})}{X} = T \cdot \frac{Y \cdot (T^{i-1} - T^{i-2})}{X} = T \cdot \left(\frac{1}{K_i^{*2}} - \frac{1}{K_{i-1}^{*2}} \right),$$

Note that under the Monkey scheme, the optimal policies always satisfy that $K_i^* \leq K_{i-1}^*$, and hence

$$\frac{1}{K_{i+1}^*} = \sqrt{\frac{1}{K_i^{*2}} + T \cdot \left(\frac{1}{K_i^{*2}} - \frac{1}{K_{i-1}^{*2}} \right)}$$

$\square$

Lemma 5.1 shows that the optimal policy of two consecutive levels can help us infer the optimal compaction policy of larger levels even if we are unaware of the specific values of the system related parameters X, Y, and Z. Therefore, we let the tuning model only tune the compaction policy of Level 1 and Level 2, and infer the compaction policy for all subsequent levels by Lemma 5.1. When the inferred policy may not be integers, we round it to the closest valid policy.

Example. We assume that the tuning result of Level 1 and Level 2 are 9 and 7, respectively. Then following Equation 4, we set the compaction policy of Level 3 as $K_3^* = \sqrt{\frac{1}{\frac{1}{7^2} + 10 \cdot \left(\frac{1}{7^2} - \frac{1}{9^2} \right)}} \approx 3$. Similarly,

Level 4 has $K_4^* = \sqrt{\frac{1}{\frac{1}{3^2} + 10 \cdot \left(\frac{1}{3^2} - \frac{1}{7^2} \right)}} \approx 1$.

Additional discussion with non-zero-result lookups. The policy propagation still largely remains viable in the presence of non-zero-result lookups. Firstly, according to previous works [12, 26], the hit rate of non-zero-result lookups for each level is in general proportional to the level capacity, which increases exponentially by a factor of T from the smallest to the largest levels. Consequently, the hit rates for all levels, except the largest level, are relatively insignificant, resulting in a negligible impact on Bloom filters and the trade-off between reads and writes in these levels. Hence, our policy propagation remains applicable in these levels. Secondly, under the Monkey scheme, the FPR of the Bloom filter increases exponentially by a factor of T for each level. As a result, the largest level has an immense FPR, which necessitates an aggressive optimal policy ($K = 1$) to remedy excessive read overhead. For instance, in our case study presented in Figure 9, the optimal policy for the largest layer is $K = 1$. Even if non-zero-result lookups may partially mitigate the influence of the Bloom filter in the largest level, the need for an aggressive optimal policy very likely remains unchanged.

6 RELATED WORK

Machine Learning for Database Storage Systems. There are extensive studies on tuning storage systems via machine learning. Pavlo *et al.* [43] and Aken *et al.* [56] propose the concept of a self-driving database that makes decisions automatically with machine learning models. Ottertune [55] includes the techniques that analyze and flit database knobs with machine learning techniques and further recommend parameter settings by learning from collected metrics. Qtune [32], CDB-Tune [64], and CDBTune+ [65] are RL-based database tuning methods, which adopt DDPG to improve their learning effectiveness in large configuration space. Bourbon [15] also incorporates machine learning into the LSM-tree, however, in a different way from our paper. It proposes to augment the fence pointers in the LSM-tree with the learned index structures [30]. We note that most of the studies are concentrated on tuning SQL databases. To the best of our knowledge, we have the first attempt on applying machine learning to NoSQL key-value database tuning.

LSM-tree Optimization There are numerous studies on LSM-tree optimizations [1–3, 7–9, 11, 12, 16–20, 23, 25–29, 36–39, 45–50, 53, 54, 57–60, 63, 66–68], as we summarized in Section 1. Among these studies, some studies [12, 16–18, 26, 27] are more related to RusKey, since they focus on tuning compaction policies as well as RusKey. Monkey [16] includes a theoretical cost model for an LSM-tree by estimating the expected I/O cost. Based on the model, Monkey co-tunes the merge policy, the buffer size, and the Bloom filters' bits-per-key to locate an optimal LSM-tree design with the minimal I/O cost for a given workload. Dayan and Idreos [17] show that existing merge policies (tiering and leveling) are not able to fully trade between read costs and write costs, therefore they propose Fluid LSM-tree that enables hybrid compaction policies. In addition, Dostoevsky further generalizes its theoretical model on hybrid compaction policies. Chatterjee *et al.* [12] propose a synthetic model named Cosine for key-value stores on the cloud. Moreover, they present an I/O cost model that is aware of workload distribution. Huynh *et al.* [26] provide a model-based analysis on workloads with uncertainty. Dayan and Idreos [18] devise a more scalable merge policy for LSM-tree by adjusting the size ratio between adjacency levels. Compared with these works, our study is the first one that employs machine learning techniques and is a pioneer work that considers real-time transition costs between different tuning settings.

7 EVALUATION

This section presents experimental studies of RusKey on various workloads. The result shows that RusKey exhibits strong robustness in performance across different workloads.

Setup. We process our experiments on a server with an Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz processor, 256GB DDR4 main memory, and 1TB NVMe SSD, running 64-bit Ubuntu 20.04.4 LTS on an ext4 partition.

Implementation. We implement RusKey's FLSM-tree on top of Facebook's RocksDB [21], a widely used LSM tree-based key-value storage system. Similar to previous works [12, 17, 18], we extend the compactor API which RocksDB provides to allow hybrid compaction policies across levels. We also implement Monkey [16] by enabling RocksDB to assign different bits-per-keys of the Bloom filters across all levels. Following previous works [17, 26], we enable direct I/Os for read and write. We set the memory buffer size to 2MB, the size ratio to 10, and the default Bloom filter bits-per-key to 8. For RocksDB with Monkey scheme, we lower bits-per-key to 4 since in this case Monkey exploits Bloom filters more effectively. We use the internal counters of RocksDB to acquire necessary statistics for states and measure the level-based latency. We implement the Lerp model with PyTorch. In our experiments, we employ a three-layer fully-connected neural network with 128 neurons per layer using ReLU activation functions for both critic and actor. We

let the end-to-end latency and by-level latency to be equally contributed to the reward by setting parameter α to 1/2.

Our previous discussions are based on the assumption that the compaction is conducted per level at a time, in line with many previous works [16–18, 26]. In RocksDB, however, compaction is performed at the granularity of files. A sorted run in these systems is composed of a few disk files. During compaction, a small set of files with overlapping key ranges in each level are merged. File compaction may reduce the actual size of a sealed run, which will be reactivated, and ultimately leads to multiple active runs in one level. Although the compaction granularity is different, a similar transition method can still be applied. To increase the compaction policy K_i of Level i by one, we reduce the capacity of all existing runs in the level from $\frac{C_i}{K_i}$ to $\frac{C_i}{K_i+1}$, where C_i is the level capacity of Level i, and initialize an empty run as the active run with the same capacity. To decrease K_i by one, we seal one of the runs in the level and increase the capacity of other sequences from $\frac{C_i}{K_i}$ to $\frac{C_i}{K_i-1}$. To avoid an additional cost and a performance slump, the sealed run does not trigger any compaction. When other runs in the level trigger a file compaction, overlapping files in the sealed run will also be involved in this compaction. Therefore, the sealed run would gradually get merged through compactions triggered by other runs in the level. In this way, a FLSM-TREE transition still takes effect immediately and incurs zero immediate transition cost.

Baselines. We compare RusKey with three typical compaction policy settings in Fluid LSM-tree [17], including aggressive compaction ($K = 1$), lazy compaction ($K = 10$), and moderate setting ($K = 5$). We name them as Aggressive, Lazy and Moderate, respectively. We also compare RusKey with the state-of-the-art hybrid compaction policy Lazy-Leveling. For fairness, we only compare Lazy-Leveling under Monkey scheme, following the setting of Dostoevisky [17]. In addition, we also include heuristic-based baselines on top of our FLSM-TREE, which greedily selects a more read or write friendly compaction policy for each level if the workload in the level moves towards the respective direction.

Experiment Design. By default, we initially bulk load the empty database with 100 million random sampled key-value entries. Each key-value entry consists of a key of 128 bytes and a value of 896 bytes. The mission size is set to 50000 operations. The setting is determined to ensure a reasonably consistent workload throughout a mission, while also involving sufficient operations in a mission to gather the necessary information for reinforcement learning. We measure the performance of the database with workloads of 100 million operations, which are divided into 2000 missions. Each operation can be *lookup* or *update*, which consists of uniformly and randomly distributed keys and values. Our evaluations are performed with various ratios between lookups and updates. For RusKey, we initiate the FLSM-TREE with leveling, which is the default compaction policy in RocksDB.

We evaluate RusKey under both the uniform Bloom filter scheme and the Monkey Bloom filter scheme. In virtue of policy propagation, we only need to tune the compaction policy Level 1 under the uniform scheme and propagate it to other larger levels. Under the Monkey scheme, we tune Level 1 and Level 2 successively with the Lerp. When the compaction policies of Level 1 and Level 2 stay stable and the RL model converges, we finish the tuning and transfer the policy to larger levels with Equation 4.

RusKey self-navigates to optimal design on static workloads. In Figure 6, we compare the performance of RusKey with the baselines under various workloads, which include read-heavy workload with 90% lookups, write-heavy workload with 10% lookups, and read-write-balanced workload with 50% lookups. Each of these workloads includes 100 million operations which are divided into 2000 missions.
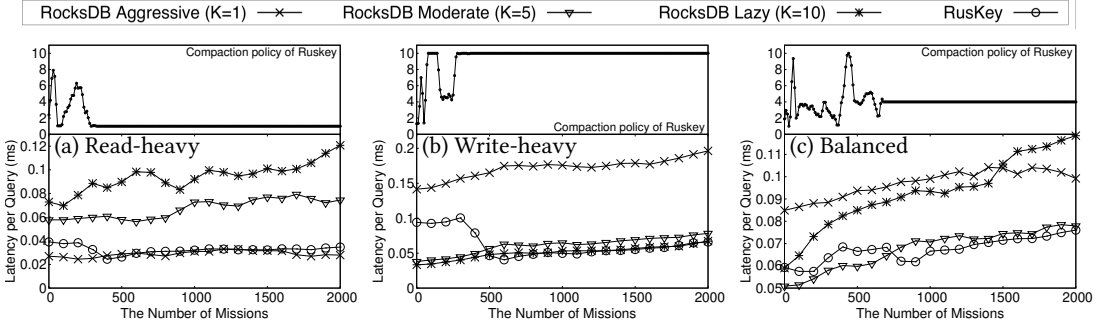
Fig. 6. RusKey self-navigates to the optimal design on static workloads.

The result shows that RusKey achieves near-optimal performance across all workloads. In contrast, the baselines perform sub-optimally on at least one workload. For example, in the read-heavy workload (Figure 6 (a)), Aggressive is the most effective because it maintains a low read cost by merging sorted runs aggressively that promises low read amplification. However, its performance degrades with the write-heavy workload (Figure 6 (b)) because the aggressive merging incurs significant additional write amplification. Moderate achieves moderate performance on each of the workloads. It performs worse than aggressive policy on the read-heavy workload and worse than lazy policy on the write-heavy workload. Meanwhile, its performance is relatively better than the other two baselines on the balanced workload. To explain, the balanced workload contains an equal quantity of lookups and updates. Aggressive and Lazy are respectively much slower on update and lookup operations. In contrast, Moderate processes both lookups and updates with reasonable efficiency. Nevertheless, the end-to-end performance of moderate policy is still lower than RusKey on the balanced workload when RusKey finishes the tuning and reaches a sweet trade-off between lookups and updates.

Overall, RusKey has robust performance across different workloads as it can adjust its compaction policies to accommodate different workload compositions. The sub-figures at the top of Figure 6 show the compaction policy of RusKey tuned by the RL model Lerp during workload processing. Through tracking the tuning strategy of Lerp, we find that Lerp selects an aggressive compaction policy with a read-heavy workload and a lazy compaction policy with a write-heavy workload. In addition, as shown in Figure 6 (c), Lerp selects an intermediate policy $K = 4$ on the balanced workload, which dominates all baselines after around 600 missions. This is because the baselines are sub-optimal on the balanced workload while RusKey self-drives to the optimal compaction policy through the Lerp model. This wise strategy is enabled by integrating the FLSM-tree with the deep RL model Lerp. Note that there are still minor gaps between RusKey and the respective optimal baselines on read-heavy and write-heavy workloads because RusKey requires a small number of missions to conduct the online training in order to find the optimal compaction policy. As shown in the upper sub-figures of Figure 6, it takes Lerp around 300 missions to explore and navigate to the optimal setting with the read-heavy and write-heavy workloads. Tuning on the balanced workload takes longer, which is around 600 missions.

RusKey performs better on dynamic workloads. In this experiment, we evaluate RusKey and the baselines with a comprehensive dynamic workload. The workload is divided into five sessions with different proportions of lookups and updates. They are in the order of read-heavy (10% update), balanced (50% update), write-heavy (90% update), write-inclined (70% update), and read-inclined (30% update). Each section includes 50 million operations which are divided into 1000 missions with 50000 operations for each.
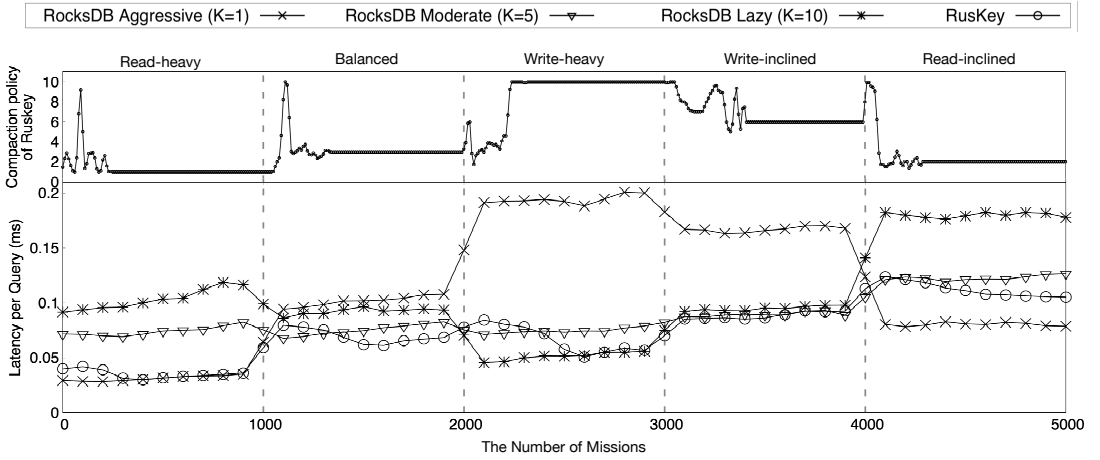
Fig. 7. RusKey outperforms other baselines on dynamic workloads through self-tuning and transforming its compaction policy according to the change of workload.

Method	Performance Ranking					Avg. Ranking
	Read-H	Balanced	Write-H	Write-I	Read-I	
K=1	**1**	4	4	4	**1**	2.8
K=5	3	2	3	**1**	3	2.4
K=10	4	3	**1**	3	4	3
RusKey	**1**	**1**	**1**	**1**	2	1.2

Table 3. RusKey ranks higher in performance on average. "H" stands for "heavy" and "I" stands for "inclined".

Figure 7 presents the latency of RusKey and the baselines as well as the compaction policy that RusKey chooses. All baselines perform sub-optimally on at least one of the sessions. In the meantime, the latency of RusKey is close-to-optimal across all sessions, benefiting from a successful tuning through Lerp and FLSM-tree's ability to flexibly transform the compaction policy. Specifically, RusKey adopts compaction policy settings of $K = 1$, $K = 5$, $K = 10$, $K = 6$, $K = 2$ on the read-heavy, balanced, write-heavy, write-incline and read-inclined sessions, respectively.

Across all workload sessions, RusKey achieves up to 4 times better performance than Aggressive (on the write-heavy session), up to 2.4 times better performance than Moderate (on the read-heavy session), and up to 3.5 times better performance than Lazy (on the read-heavy session). Note that in Figure 7, the latency of RusKey rises slightly at the beginning of each workload session and then drops rapidly. The reason behind this is that Lerp requires a short retraining under a new workload setting.

To further show the robustness of RusKey, we compute the performance ranking of all methods across all five workload sessions after tuning RusKey, which is summarized in Table 3. The performance is compared by the average time cost per operation after the RL model is converged in each session. The result demonstrates that RusKey ranks first for four workloads and ranks second for the remaining one. Overall, it achieves a high average ranking of 1.2, demonstrating the strong robustness of RusKey.

RusKey is effective under the Monkey scheme. In Figure 8, we compare RusKey with the baselines under the Monkey scheme, with the same workload setting as Figure 6. In addition to the three basic baselines, we also compare with Lazy-Leveling [17], which is one of the state-of-the-art compaction policies designed for the Monkey scheme. The result is similar to Figure 6, showing
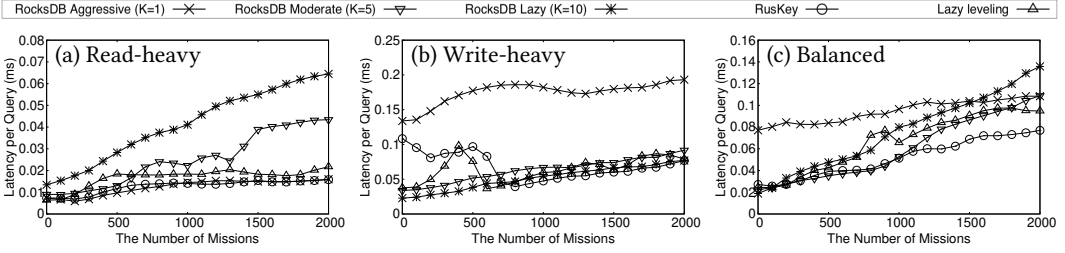
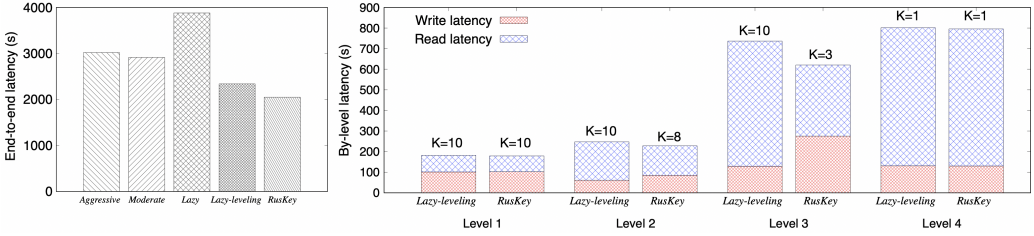Fig. 8. RusKey is effective under the Monkey scheme.

Fig. 9. RusKey adopts novel policy settings compared with previous designs.

that RusKey achieves near-optimal performance across various workloads under the Monkey scheme as well. Moreover, Lazy-Leveling also reaches near-optimal across all workloads, whereas RusKey performs better than Lazy-Leveling on every workload. Specifically, RusKey performs significantly better than Lazy-Leveling and other baselines on the balanced workload. The reason is that RusKey is able to adopt novel policy settings through policy propagation. We will explain this in detail with a subsequent experiment.

RusKey adopts novel policy settings compared with previous designs. We compare the compaction policy setting of RusKey and the state-of-the-art design Lazy-Leveling under the read-write-balanced workload. The experiment is conducted on a workload of 25 million operations under the Monkey scheme after RusKey finishes self-tuning. We report the end-to-end latency (i.e., the running time of processing all the operations) in Figure 9 Left and the per-level latency in Figure 9 Right. Under this case, RusKey adopts different compaction policies at each level. It sets an aggressive policy at the first level. As the level goes deeper, the policy becomes lazier. This matches the intuition of Lazy-Leveling, that is, compactions in smaller levels are less needed than in smaller levels with the Monkey scheme, because the read cost is lower in smaller levels. The superiority of RusKey is that RusKey self-tunes its compaction policy in a more elaborate way than Lazy-Leveling and gains more benefits from this intuition.

RusKey remains effective when for YCSB workloads. We evaluate RusKey under the YCSB standard benchmarks and report the results in Figure 11. We use the default Zipfian distribution, in which the update frequency and access frequency of keys follow the power law. As shown in Figure 11 (a), (b), and (c), the results on YCSB are quite similar in trends to the results on uniform workloads, if we set the composition of workloads the same as the previous experiments. These results further verify that RusKey delivers near-optimal performance across diverse workloads on the YCSB benchmark. Furthermore, we evaluate YCSB benchmark with a workload that contains 50% range lookups and 50% updates, and the results are shown in Figure 11 (d). Among all baselines, Aggressive achieves the lowest latency, and the performance of RusKey is on par with that of Aggressive. This demonstrates that RusKey remains high effectiveness even with range lookups.

(a) Write latency   (b) Read latency
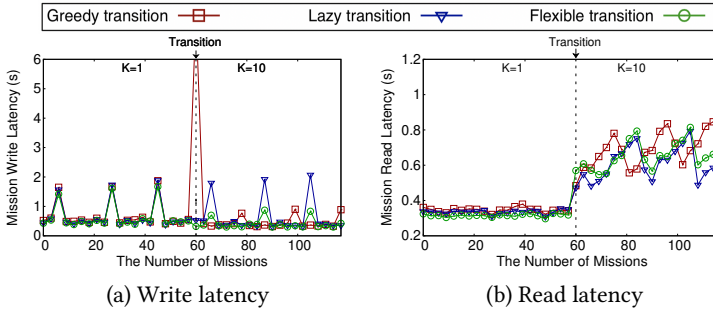
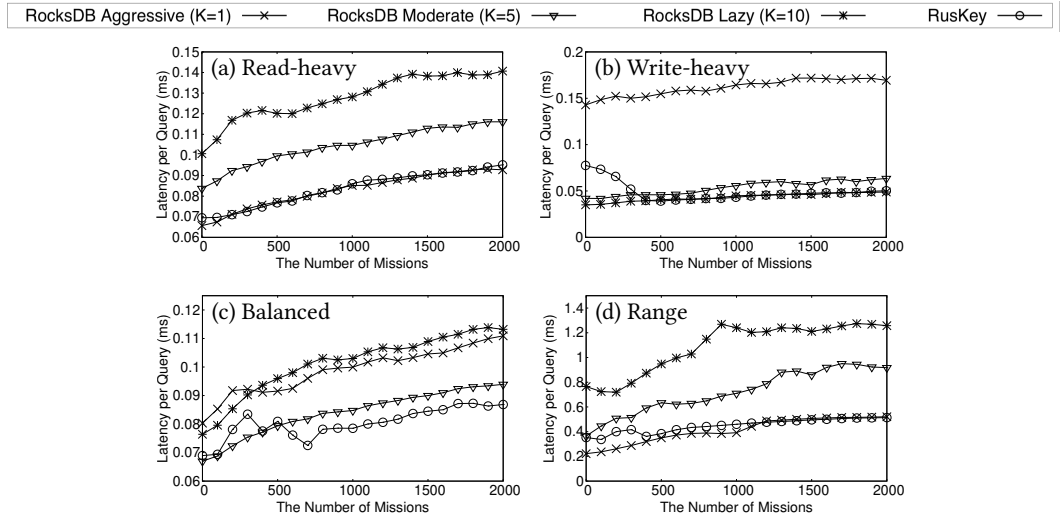Fig. 10. Flexible transition incurs no immediate transition cost and delay.

Fig. 11. RusKey is robust on YCSB benchmarks and can self-tune to a near-optimal policy.

Flexible transition performs better than other transition methods. In Figure 10, we evaluate the greedy, lazy, and flexible transitions using a micro-benchmark with compaction granularity of levels. We employ a read-write-balanced workload (50% lookups and 50% updates) of 120 missions, while each mission contains 1 million operations. Each level's initial compaction policy in the FLSM-TREE is 1. The compaction policy is transformed from 1 to 10 midway through the workload. Specifically, the compaction policy transition happens at the 60th mission, which is represented by the dotted lines in Figure 10. In Figures 10 (a) and 10 (b), the write latency and read latency for each mission using various transition methods are depicted, respectively. The spikes of write latency in Figures 10 (a) are caused by compactions with the level granularity. Figure 10 (a) demonstrates that when a greedy transition occurs, it results in a very high write latency spike since changing the compaction policies at every level necessitates compacting all data in each level to the next level. With a greedy transition and a flexible transition, the write latency spikes brought on by compactions are significantly lower after the policy is changed. The reason is that the new compaction policy of $K = 10$ incurs less write amplification, so the cost of each compaction, i.e. the spikes, would be lower. However, with a lazy transition, the height of spikes stays the same before and after the transition, which means that the compaction policies in larger levels still remain unchanged, because the transition is delayed until the level is completely filled. Moreover, we can find similar results in Figures 10 (b). The read latencies of the greedy and flexible transition are
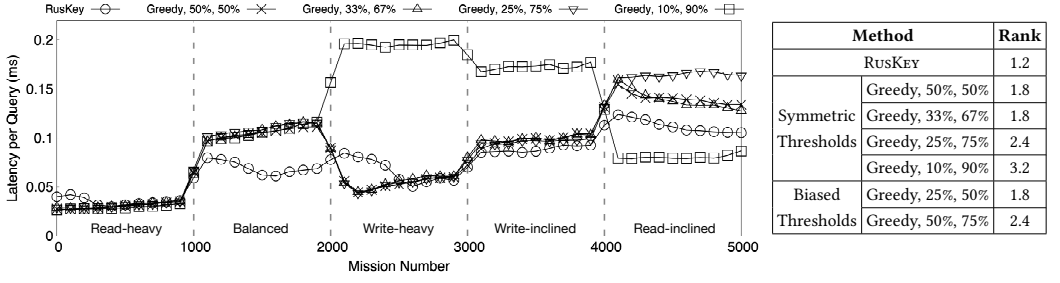
Fig. 12. RusKey is more robust than greedy approaches with various thresholds.
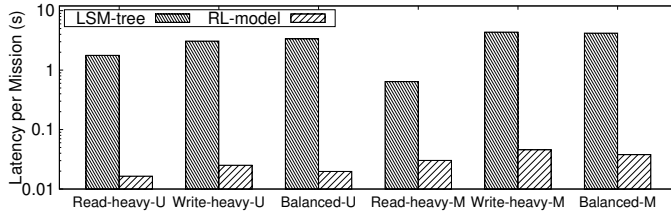

Fig. 13. RusKey's model update time is relatively insignificant.

higher than that of the lazy transition after the transition happens, which also reflects the delay of the lazy transition. In terms of the end-to-end latency, the greedy transition, lazy transition and flexible transition use 51s, 44s, and 40s, respectively. In general, Figure 10 indicates that the flexible transition incurs almost-free immediate transition cost and has no delay, demonstrating its superiority.

Comparison with greedy approaches. An interesting question is whether a simple greedy policy can effectively address our problem. To explore this possibility, we implemented a series of greedy strategies for comparison. We utilized a per-level detector to detect the workload within a level and set two thresholds, h_{bottom} and h_{top}. If the percentage of lookups in the level is less than h_{bottom}, the greedy algorithm identifies the workload as write-heavy and increases the compaction policy of the level by one. Conversely, if the percentage of lookups in the level exceeds h_{top}, the greedy algorithm recognizes the workload as read-heavy and decreases the compaction policy by one. We consider a wide range of thresholds, including the symmetric thresholds and biased thresholds shown in the right table of Figure 12, and present four settings in the left figure of Figure 12 for better visualization. The results indicate that some greedy methods can achieve the desired performance for read-heavy or write-heavy workloads. However, for other workloads such as balanced, read-inclined, and write-inclined workloads, they fail to reach the optimal performance. In contrast, RusKey can self-tune to a near-optimal performance across all workloads, thus providing a relatively more robust performance. The robustness of RusKey is attributed to its ability to capture the intricate mapping from workload ratio to the optimal policy through a reward mechanism, which is absent in the greedy approaches.

The right table in Figure 12 summarizes the average performance ranking of all methods, similar to the setting of Table 3. Overall, RusKey achieves the highest average ranking, surpassing the best setting of a greedy approach we tested. Our findings suggest that while greedy policies may be effective in some scenarios, the use of reinforcement learning methods like RusKey may provide more robust and comprehensive optimization for LSM-trees.

The model updating time is insignificant. In Figure 13, we compare the RL model update cost and LSM-tree operation cost per mission on various workloads. "U" and "M" refer to the uniform Bloom filter scheme and the Monkey scheme, respectively. The Y-axis is in logarithmic scale. Our results show that the model update cost is insignificant compared with the total cost of processing the workload by the LSM-tree. For most workloads, the time spent on updating the model is at most 1% of the time for processing LSM-tree operations. For example, on the balanced workload with the uniform Bloom filter scheme, the average query processing cost per mission is 3.36 seconds, while the average model update cost is only 0.019 seconds. The small update cost of the RL model of RusKey is mainly attributed to the reduction of the action space.

Brute-force learning approaches can be impractical. We evaluated two brute-force training approaches: (1) *without a level-based model that reduces the action space*, and (2) *without the policy propagation and train all levels.* We limit the maximum experiment time to 24 hours and use the setting of Figure 6 (c). Our results show that the first approach cannot finish learning within the given time frame while the second approach fails to achieve the optimum policy starting from Level 3 due to insufficient samples. This demonstrates the importance of our optimization strategy.

Limitations. Our work represents a first attempt at using reinforcement learning to optimize LSM-trees, and as such, we have only explored one feasible solution for using machine learning to tune LSM-trees. However, there are opportunities for further improvement of RusKey by incorporating additional design parameters. For instance, we could learn to adjust the memory allocation for Bloom filters by considering a more refined analysis with non-zero result workloads or adapt size ratios based on a given workload. The challenge here is to maintain a practical action space and a reasonable LSM-tree transition cost. Another limitation of our approach is that the flexible transition degenerates into lazy transition in an extreme read-only case since flexible transition only takes effect when there are updates. It is unclear whether there are better transition methods for this scenario, and this remains an open question for future research. In summary, there is still much to be explored regarding the optimization of LSM-trees using reinforcement learning, and these aspects deserve further study in future works.

8 CONCLUSION

We present RusKey, which is an RL-enhanced key-value store that supports dynamic workloads in an online manner. We integrate into RusKey a novel FLSM-tree to perform an efficient compaction policy transformation, where a FLSM-tree improves an original LSM-tree in terms of transition costs and delays. Further, we model the system by a level-based RL model Lerp, which applies DDPG to each level of the FLSM-tree for effective tuning of RusKey under dynamic workloads. Moreover, we propose a policy propagation method across levels, reducing the demand for training samples.

9 ACKNOWLEDGEMENT

REFERENCES

[1] Ildar Absalyamov, Michael J Carey, and Vassilis J Tsotras. 2018. Lightweight cardinality estimation in LSM-based systems. In *Proceedings of the 2018 International Conference on Management of Data*. 841–855.

[2] Muhammad Yousuf Ahmad and Bettina Kemme. 2015. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment* 8, 8 (2015), 850–861.

[3] Wail Y Alkowaileet, Sattam Alsubaiee, and Michael J Carey. 2019. An LSM-based Tuple Compaction Framework for Apache AsterixDB (Extended Version). *arXiv preprint arXiv:1910.08185* (2019).

[4] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. 2014. AsterixDB: A scalable, open source BDMS. *arXiv preprint arXiv:1407.0454* (2014).

[5] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.

[7] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 363–375.

[8] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 753–766.

[9] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better memory organization for LSM key-value stores. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1863–1875.

[10] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[11] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 1007–1019.

[12] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: a cloud-cost optimized self-designing key-value storage engine. *Proceedings of the VLDB Endowment* 15, 1 (2021), 112–126.

[13] Source Code. 2022. WiredTiger. https://github.com/wiredtiger/wiredtiger.

[14] Carlo Curino, Evan PC Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 313–324.

[15] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 155–171.

[16] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.

[17] 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* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 505–520. https://doi.org/10.1145/3183713.3196927

[18] 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.

[19] Niv Dayan and Moshe Twitto. 2021. Chucky: A Succinct Cuckoo Filter for LSM-Tree. In *Proceedings of the 2021 International Conference on Management of Data*. 365–378.

[20] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3071–3084.

[21] Facebook. 2022. RocksDB. https://github.com/facebook/rocksdb.

[22] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. 2007. Workload analysis and demand prediction of enterprise data center applications. In *2007 IEEE 10th International Symposium on Workload Characterization*. IEEE, 171–180.

[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] Google. 2022. LevelDB. https://github.com/google/leveldb/.

[25] 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*. 651–665.

[26] Andy Huynh, Harshal Chaudhari, Evimaria Terzi, and Manos Athanassoulis. 2021. Endure: A Robust Tuning Paradigm for LSM Trees Under Workload Uncertainty. *arXiv preprint arXiv:2110.13801* (2021).

[27] Stratos Idreos, Niv Dayan, Wilson Qin, Mali Akmanalp, Sophie Hilgard, Andrew Slavin Ross, James Lennon, Varun Jain, Harshita Gupta, David Li, and Zichen Zhu. 2019. Design Continuums and the Path Toward Self-Designing Key-Value Stores that Know and Learn. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[28] Taewoo Kim, Alexander Behm, Michael Blow, Vinayak Borkar, Yingyi Bu, Michael J Carey, Murtadha Hubail, Shiva Jahangiri, Jianfeng Jia, Chen Li, et al. 2020. Robust and efficient memory management in Apache AsterixDB. *Software: Practice and Experience* 50, 7 (2020), 1114–1151.

[29] Eric R Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A Self-Designing Range Filter. In *Proceedings of the 2022 International Conference on Management of Data*. 1670–1684.

[30] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

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

[32] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. 12, 12 (Aug. 2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[33] Meng Li, Deyi Chen, Haipeng Dai, Rongbiao Xie, Siqiang Luo, Rong Gu, Tong Yang, and Guihai Chen. 2022. Seesaw Counting Filter: An Efficient Guardian for Vulnerable Negative Keys During Dynamic Filtering. In *Proceedings of the ACM Web Conference 2022*. 2759–2767.

[34] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[35] LinkedIn. 2022. Voldemort. http://www.project-voldemort.com.

[36] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[37] Chen Luo and Michael J Carey. 2019. On performance stability in LSM-based storage systems (extended version). *arXiv preprint arXiv:1906.09667* (2019).

[38] Chen Luo and Michael J Carey. 2020. Breaking down memory walls: adaptive memory management in LSM-based storage systems. *Proceedings of the VLDB Endowment* 14, 3 (2020), 241–254.

[39] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2071–2086.

[40] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*. 631–645.

[41] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[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] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, Vol. 4. 1.

[44] Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. 2018. A comparative study of secondary indexing techniques in LSM-based NoSQL databases. In *Proceedings of the 2018 International Conference on Management of Data*. 551–566.

[45] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 497–514.

[46] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.

[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] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2022. Constructing and analyzing the LSM compaction design space. *arXiv preprint arXiv:2202.04522* (2022).

[49] 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.

[50] Pradeep J Shetty, Richard P Spillane, Ravikant R Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building Workload-Independent Storage with VT-Trees. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. 17–30.

[51] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[52] Richard S Sutton, Andrew G Barto, et al. 1998. Introduction to reinforcement learning. (1998).

[53] Yu, Geoffrey X and Markakis, Markos and Kipf, Andreas and Larson, Per-Åke and Minhas, Umar Farooq and Kraska, Tim. 2022. TreeLine: an update-in-place key-value store for modern storage. *Proceedings of the VLDB Endowment* 16, 1 (2022), 99–112.

[54] Risi Thonangi and Jun Yang. 2017. On log-structured merge for solid-state drives. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 683–694.

[55] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[56] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. 2021. An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1241–1253.

[57] Tobias Vinçon, Sergej Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. 2018. Noftl-kv: Tackling write-amplification on kv-stores with native storage management. In *Advances in database technology-EDBT 2018: 21st International Conference on Extending Database Technology, Vienna, Austria, March 26-29, 2018. proceedings*. University of Konstanz, University Library, 457–460.

[58] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. 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*. 1–14.

[59] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 71–82.

[60] 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.

[61] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Qingxin Gui, Fei Wu, and Changsheng Xie. 2017. A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores. In *Proc. 33rd Int. Conf. Massive Storage Syst. Technol.(MSST)*. 1–13.

[62] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Fei Wu, and Changsheng Xie. 2017. Building efficient key-value stores via a lightweight compaction tree. *ACM Transactions on Storage (TOS)* 13, 4 (2017), 1–28.

[63] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*. 323–336.

[64] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.

[65] Ji Zhang, Ke Zhou, Guoliang Li, Yu Liu, Ming Xie, Bin Cheng, and Jiashu Xing. 2021. CDBTune+: An efficient deep reinforcement learning-based automatic cloud database tuning system. *The VLDB Journal* 30, 6 (2021), 959–987.

[66] 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.

[67] Yueming Zhang, Yongkun Li, Fan Guo, Cheng Li, and Yinlong Xu. 2018. ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.

[68] Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, and Manos Athanassoulis. 2021. Reducing bloom filter cpu overhead in lsm-trees on modern storage devices. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*. 1–10.