AdCache: Adaptive Cache Management with Admission Control for LSM-tree Key-Value Stores

Jiarui Ye
Nanyang Technology University
Singapore, Singapore
jiarui005@e.ntu.edu.sg

Junfeng Liu
Nanyang Technology University
Singapore, Singapore
junfeng001@e.ntu.edu.sg

Siqiang Luo
Nanyang Technology University
Singapore, Singapore
siqiang.luo@ntu.edu.sg

Abstract

Modern key-value stores rely on LSM-tree architectures for high write throughput, but suffer from complex read patterns that challenge traditional caching strategies. Existing cache designs, such as block cache and range cache, perform well under certain workloads but sub-optimal under others, and static configurations often fail to adapt to dynamic access patterns. We present AdCache, a reinforcement learning-assisted caching system that dynamically adjusts cache partitioning and admission policies based on workload characteristics. AdCache employs an actor-critic model to learn the best memory allocation between block cache and range cache, and incorporates lightweight admission control for both point lookups and range scans. It selectively caches frequent keys and only the most beneficial portions of scans, thereby avoiding unnecessary evictions and improving memory efficiency. Our implementation of AdCache in RocksDB achieves up to 14% higher cache hit rate and reduces SST reads by up to 25% compared to the default block cache.

Keywords

Cache, Reinforcement Learning, Range Query, LSM-tree

1 Introduction

Key-value stores are an important part of many modern systems, supporting applications such as real-time analytics, recommendation engines, content delivery, and cloud services [7, 10]. To handle high write and read demands, many key-value stores, like RocksDB [17], LevelDB [20], and Cassandra [16], use a storage method called the Log-Structured Merge (LSM) tree. LSM trees help improve write performance by collecting updates in memory and then writing them to disk in batches as sorted, immutable files across several levels [35].

Caching is a foundational technique used across nearly all layers of modern computing infrastructure. It plays a critical role in improving latency, reducing backend pressure, and saving computation or I/O cost by storing frequently accessed data closer to the application. From CPU caches and operating system page caches to content delivery networks (CDNs) and database buffer pools, caching is deeply integrated into hardware and software systems. However, most of these caching systems are designed primarily for workloads composed of point queries, where each request accesses a single object, key, or memory block. In contrast, key-value stores built on Log-Structured Merge (LSM) trees frequently serve not only point lookups but also range scan queries. These range queries access sequences of adjacent key-value pairs, often touching large portions of the dataset in a short time window. Meanwhile, compaction operations in LSM trees periodically rewrite and reorganize the data layout, invalidating cached blocks associated with outdated files. These two characteristics, large-footprint queries and structural changes, create caching challenges that differ significantly from traditional systems designed primarily for isolated point accesses.

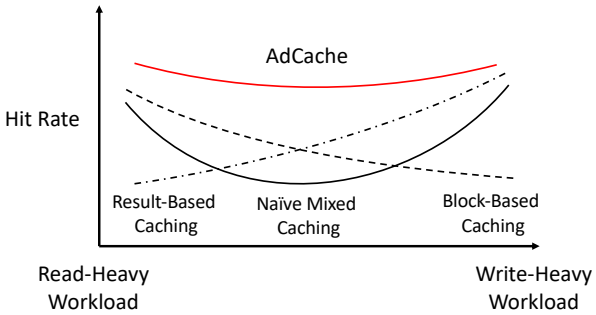

Figure 1: Performance trade-off across different workload patterns under existing LSM-tree caching strategies.

Challenges and limitations of existing caching strategies. The structural and workload differences between LSM-tree-based key-value stores and other systems present unique challenges for cache management. Two types of caching stratigies have been proposed for LSM-tree systems. Traditional **block-based caching** strategies, similar to page-based caching in non-LSM systems, are used in LSM-tree systems like RocksDB to store data blocks fetched from disk and are highly effective for point lookups. However, their performance degrades under frequent compactions, which rewrite files and invalidate cached blocks identified by file offset. To mitigate this, Leaper [47] introduces data block prefetching after compactions to reduce the impact of large-scale cache invalidation. **Result-based caching**, like Range Cache [43], on the other hand, abandons the conventional block structure and caches query results as sorted key-value sequences. While this result-based caching design is compaction-resilient and well-suited for scan-heavy workloads, it tends to achieve lower hit rates than block cache in workloads with few updates. This is mainly due to the structural mismatch between the cache layout and the block-based storage organization on disk, which can reduce reuse efficiency for point queries and short scans. As depicted in Figure 1, neither block-based nor result-based caching strategies can consistently perform well across all workload scenarios. Each is suited to a specific access pattern, and their strengths do not generalize. Consequently, a naive combination of the two, such as statically partitioning memory between block cache and range cache, fails to adapt to dynamic workloads. The optimal cache ratio in one phase, e.g., a scan-heavy workload, may become suboptimal in another, such as a point-heavy phase. Without the ability to reallocate memory dynamically, static designs suffer from either overflow or underutilization of fixed-size cache components, leading to degraded performance

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.12

and conflicting eviction behavior. Moreover, such designs lack the ability to distinguish high-value accesses from low-value noise, the infrequent queries that are unlikely to be reused, especially the long range scans. Admitting noisy items into the cache can evict more valuable entries, compounding performance issues under dynamic workloads. To support the wide variety of workload compositions encountered in real-world systems, a more flexible and adaptive caching architecture is required, one that can dynamically reconfigure its internal structure and selectively filter entries based on observed query behavior and access trends.

The Research Question: How to enable the cache components of LSM-tree systems with compaction-resistance, noise-resistance, and workload-adaptivity? Specifically, the cache should maintain high effectiveness even in the presence of frequent compactions, which are inherent to LSM-tree designs, and should avoid admitting low-frequency or large-size noise accesses that waste limited cache space. This includes selectively admitting scan results without over-committing memory to long, infrequent scans, and filtering point lookups that are unlikely to be reused. Achieving this balance requires a dynamic strategy that can adapt to workload shifts and distinguish between valuable and transient access patterns in real time.

Design 1: Adaptive caching framework for mixed cache data structures. The first key contribution of our work is a unified and adaptive caching framework that bridges two previously separate paradigms: block caching and query-result caching. In LSM-tree-based key-value stores, these two caching strategies have previously been treated as independent components, often with fixed memory partitions and no coordination. In contrast, our system dynamically adjusts the memory allocation between block cache and range cache in response to real-time workload characteristics. This allows the system to exploit the strengths of both approaches: the fine-grained, compaction-resilient behavior of result caching and the spatial efficiency of block caching for lookup-heavy workloads. By integrating both cache types under a single adaptive controller, our design eliminates the need for manual tuning and static cache boundaries, enabling more effective utilization of limited memory across diverse and shifting workloads.

Design 2: Selective admission control. In addition to adaptive cache partitioning, our system introduces fine-grained admission control mechanisms tailored to both point lookups and range scans. For point lookups, we implement a lightweight, frequency-based admission policy that filters out low-reuse keys, ensuring that only frequent items are cached. This prevents cache pollution from one-off or noisy accesses. For range scans, rather than following the traditional all-or-nothing approach, our method learns to cache only the most useful subset of scan results. This selective behavior is particularly important in LSM-tree environments, where scan operations can touch large key ranges and easily exhaust the cache if not controlled. By applying admission control to both access types, our system reduces unnecessary evictions, improves memory efficiency, and adapts to a wide variety of workload patterns.

Design 3: Reinforcement learning-based adaptive controller. At the core of our system is a reinforcement learning (RL) controller that jointly manages both cache partitioning and admission control. Instead of relying on manually tuned heuristics or static thresholds, the RL agent continuously observes workload patterns—such as access type ratios, cache hit statistics, and scan lengths—and outputs decisions that govern the block-to-range memory ratio as well as admission thresholds for point and scan

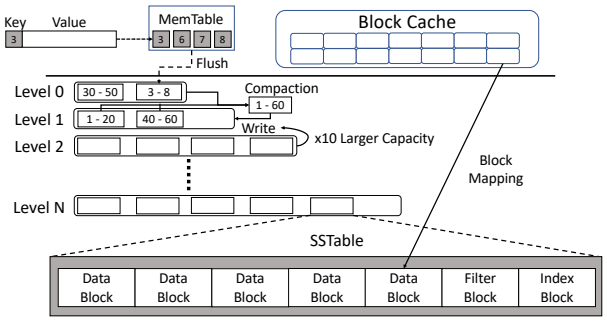

Figure 2: The structure of LSM-tree key-value stores.

queries. This unified control loop enables the system to adapt in real time to shifting workloads without requiring prior profiling or expert knowledge. Compared to rule-based or analytical methods, which often rely on strong assumptions or costly runtime statistics, RL offers a more scalable and flexible alternative. It can implicitly capture complex relationships between workload features and performance outcomes, learning effective policies from observed behavior. The decision space is continuous and low-dimensional, making the controller lightweight enough for online deployment while still expressive enough to support diverse and dynamic caching strategies. By optimizing for long-term hit rate, the RL controller ensures robust performance across heterogeneous and evolving workloads.

Contributions. In summary, we make the following contributions.

- This paper explores reinforcement learning-based adaptive caching tailored specifically for LSM-tree-based key-value stores. We present a hybrid caching system that dynamically manages both block and range cache under mixed workloads with the following techniques: (1) We propose a unified, workload-aware cache architecture that integrates block-based and result-based caching under a single adaptive framework, enabling flexible memory partitioning guided by workload characteristics. (2) We design fine-grained admission control strategies for both point lookups and range scans. These mechanisms filter low-reuse items and selectively admit high-value data into the cache, significantly reducing pollution from infrequent queries. (3) We introduce a lightweight actor-critic reinforcement learning agent that operates online and adjusts cache partition ratios and admission thresholds in real time, guided by a I/O-based reward model tailored to LSM-tree workloads.
- We implement **AdCache** (**Ad**aptive **Cache** Management with **Ad**mission Control), a reinforcement learning (RL) caching system for LSM-tree storage engines on the top of RocksDB. AdCache achieves up to 14% higher cache hit rate and reduces SST reads by up to 25% compared to the default block cache in RocksDB. Under dynamic workloads, AdCache improves the average throughput by 12%.

2 Background

2.1 LSM Tree

Log-Structured Merge (LSM) trees are a write-optimized data structure used in key-value stores like RocksDB, LevelDB, and

Cassandra. The system overview of a LSM-based Key-Value Store is shown in Figure 2. They buffer writes in memory and flush them to disk in batches, offering better write throughput than traditional B-tree-based systems [27, 36].

An LSM-tree organizes data across multiple levels of sorted files, also referred to as sorted runs [34, 38]. In RocksDB, each level from Level-1 onward contains a single sorted run, while Level-0 may contain multiple overlapping sorted runs to accommodate higher write throughput. New writes go into a memory table (MemTable), which is later flushed to disk as immutable sorted files (SSTables, SSTs). These files are periodically merged by a process called compaction to remove obsolete data and maintain order.

Updates and Compactions. In LSM-tree systems, updates, including inserts and overwrites, are first written to an in-memory structure (MemTable) and then flushed to disk as immutable sorted string tables (SSTables). Over time, these files are periodically merged and reorganized through a process called compaction. Compactions eliminate obsolete entries, reclaim space, and maintain sorted order across levels, but also change the physical layout of data on disk. As a result, compactions can invalidate cache entries that are tied to specific file offsets, posing challenges for conventional block caching.

Point Lookups. A point lookup retrieves the value for a single key. The query is first checked against the MemTable and any unflushed data in the Write-ahead Log (WAL). If not found, the system searches through SSTables from newest to oldest, using Bloom filters and index blocks to quickly skip irrelevant files and blocks. Once the key is located, its containing data block is loaded and potentially cached. In read-intensive workloads, point lookups benefit significantly from effective caching.

Range Lookups. Range lookups, also called scans, retrieve all key-value pairs within a specified key range. These queries may span multiple SSTables and levels, requiring a merge of sorted streams. Even short range scans can touch many blocks, especially when data is fragmented across levels. Because each SSTable must be sequentially traversed for matching keys, range scans tend to generate more block reads and exhibit different caching behaviors compared to point lookups. Without admission control, large scans can quickly evict useful cached entries.

2.2 Caching Strategies in LSM-Tree Systems

Caching strategies in LSM-tree-based key-value stores (LSM-KVS) can generally be divided into two classes: *block-based caching*, which caches physical data blocks as they appear on disk, and *result-based caching*, which stores the results of queries directly as key-value pairs in memory. These two approaches reflect different trade-offs in terms of cache granularity, compaction sensitivity, and suitability for various access patterns.

Block-Based Caching in LSM-KVS. Since data in LSM-trees is stored on disk in the form of fixed-size data blocks, block-based caching is the most common and straightforward approach. Systems like RocksDB and LevelDB implement a **Block Cache** that stores recently accessed data blocks in memory, typically managed with LRU or CLOCK-based eviction policies. Block caching is efficient for point lookups and short-range scans, particularly when key locality is strong. However, one major drawback of this approach is its sensitivity to compaction. When compaction rewrites SSTables, the cached blocks—identified by file and offset—become invalid, even if the logical data remains unchanged. This leads to abrupt drops in hit rate and wasted memory. To alleviate this, **Leaper** [47] introduces post-compaction block prefetching to repopulate the cache. **LSbM-tree** [40] offers a buffer cache on disk for compaction solely. However, these methods still inherits the limitations of physical block granularity and compaction invalidation.

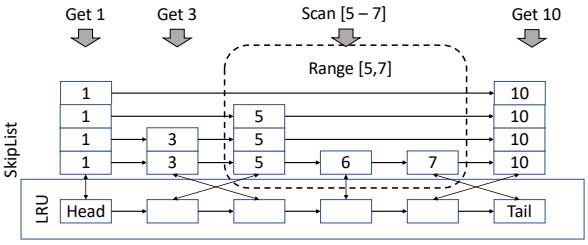

Figure 3: Structure of result-based caching strategy Range Cache. A point lookup results in a single key being cached, and a range lookup results in a range of adjacent keys being cached.

Result-Based Caching in LSM-KVS. To mitigate cache invalidation caused by compactions, result-based caching was introduced as an alternative to traditional block-based approaches. Instead of storing entire data blocks, this strategy caches query results directly as key-value pairs, which are decoupled from the physical SSTable layout and thus unaffected by compaction. RocksDB implements a simple form of result-based caching through the **Row Cache**, which stores frequently accessed key-value pairs without retaining the full data block in memory. **SpotKV** [31] takes the I/O cost of point lookups into consideration. **AC-Key** [45] extends this idea by integrating three types of caches: a key-value cache (KV Cache), a key-pointer cache (KP Cache), and a block cache. Both KV and KP caches are immune to compactions but are limited to serving point lookups. AC-Key further introduces a hierarchical caching structure that dynamically adjusts the memory allocation across these three cache types using the Adaptive Replacement Cache (ARC) policy. **Range Cache** [43] generalizes this idea to support range queries. It caches the results of both point and range lookups in a sorted structure (e.g., a skip list), as illustrated in Figure 3. Because range cache entries are based on logical key order rather than file layout, they remain valid across compactions. This makes result-based caching particularly effective in update-heavy workloads. However, in read-oriented workloads with stable data and fewer compactions, range cache often underperforms compared to block-based strategies. This is primarily due to the mismatch between the cache's logical layout and the underlying block-based storage structure, which can reduce hit locality, increase lookup cost and extra I/Os for long scans.

3 AdCache: Reinforcement Learning-Driven Cache Partitioning and Admission

This section presents the design of our reinforcement learning-assisted caching framework for LSM-tree-based key-value stores. Our system addresses two key challenges: how to efficiently cache query results, and how to dynamically balance the memory allocation between different cache types based on workload characteristics. The entire framework is driven by an online reinforcement learning agent that adapts cache behavior in real time.

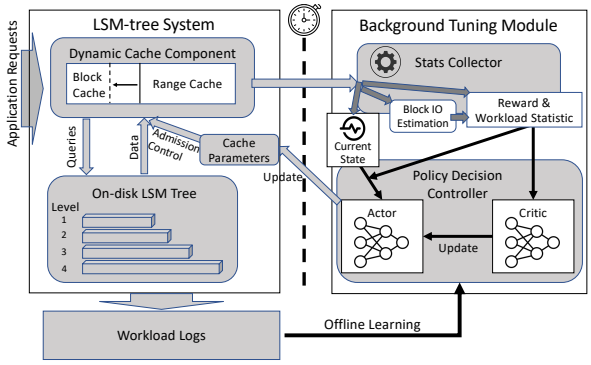

Figure 4: Components and mechanisms of AdCache.

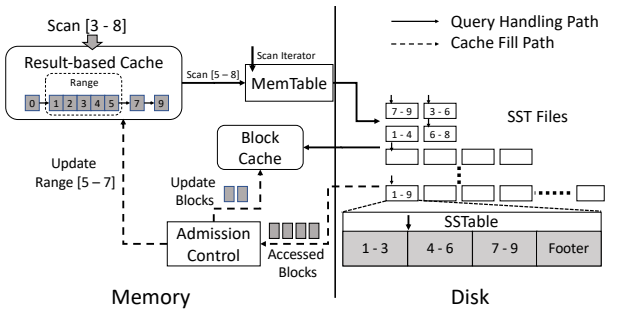

Figure 5: Workflow of queries in AdCache.

3.1 Overview of AdCache

Figure 4 shows the high-level architecture of our proposed reinforcement learning-assisted cache management system. The system consists of two major parts: the *LSM-tree system* and the *Background Tuning Module*.

The LSM-tree system handles incoming application requests, and forward it to the **Dynamic Cache Component**. Dynamic Cache Component consists of a block cache and a range cache with a dynamic memory boundary adjusted in runtime. The Background Tuning Module continuously monitors system performance. A **Stats Collector** gathers workload statistics and block I/O measurements. The **Policy Decision Controller**, implemented using an actor-critic reinforcement learning model, adjusts cache parameters based on the observed state and reward signals. The model outputs two main decisions: (1) the memory partitioning between block cache and range cache, and (2) the admission control parameters for query results. The workload logs can be collected for pretraining to enhance system scalability, learning stability and avoid further online learning costs.

Importantly, all model inference and training occur asynchronously in the background. Cache parameter updates are decoupled from the main query serving path, ensuring that reinforcement learning computations introduce no noticeable overhead to normal database operations. This overall architecture allows the system to adapt its caching strategy dynamically to changing workloads, improving cache hit rates and system efficiency without manual tuning.

3.2 Cache Interaction Workflow

To better illustrate how our caching framework integrates into the query processing and data access pipeline, we separate the system behavior into two conceptual workflows: the *query handling path* and the *cache fill path*, as shown in Figure 5.

Query Handling Path. When a query (either a point lookup or a range scan) arrives, it first checks the range cache. If the query result is found, it is returned immediately without further I/O. Otherwise, the system proceeds to probe the MemTable, which contains recently written data. If still unresolved, the system searches for the corresponding data blocks in the block cache. On a block cache miss, the query ultimately triggers a disk read. This top-down process prioritizes memory-resident data and minimizes disk access whenever possible.

Cache Fill Path. When data is retrieved from disk due to a cache miss, it flows through the cache fill path. The accessed blocks are inserted into the block cache, enabling potential reuse in future queries. Additionally, depending on the workload and learned policy, a portion of the query result may also be admitted into the range cache. This bottom-up path captures how queries populate the cache and how data flows into memory over time.

3.3 Adaptive Cache Partitioning

Our system employs two distinct types of caches: the **block cache** and the **range cache**. Each cache targets different access patterns and exhibits different sensitivities to compaction and workload characteristics.

By combining both cache types and adjusting their relative sizes dynamically, our system can adapt to a wide variety of workloads. For example, in a point-lookup-oriented workload mixed with updates and scans, maintaining a large range cache with a small block cache can maximize memory reuse and hit rate. Even in this setting, having a small block cache remains important: when a range query is not fully cached in the range cache, the system must scan the underlying LSM-tree. Since LSM-trees organize data in levels with exponentially increasing size, blocks from lower levels are accessed more frequently during scans. A small block cache can effectively capture these hot blocks from lower levels, significantly improving scan performance and overall cache hit rate. This phenomenon has also been observed by [43].

To manage the memory allocation between block cache and range cache, we introduce a dynamic *cache boundary*. The boundary determines how much memory is allocated to each cache type at runtime. Rather than using a static partition, the boundary is continuously adjusted by the RL agent based on real-time workload statistics, such as point-to-scan ratios, cache hit/miss rates, scan lengths, and the observed impact of compactions.

In workloads with little update activity and stable access patterns, the RL agent shifts the boundary toward a larger block cache. In workloads with frequent compactions operations, it reallocates more memory to the range cache while preserving a small but effective block cache for important blocks at lower levels and other hot data blocks. This adaptive partitioning ensures that memory is always focused on the cache structure that provides the highest benefit for the current workload.

3.4 Admission Control for Lookup Queries

Admission control was originally introduced to prevent cache pollution, where infrequently accessed items evict more valuable data. Early cache systems admitted all misses by default, but research such as TinyLFU [15] demonstrated that this can significantly reduce cache efficiency in point lookup-only workloads with skewed or bursty access patterns. TinyLFU introduced a

lightweight frequency-based admission policy, showing that selectively admitting only high-frequency items improves hit rate and overall performance. In LSM-tree-based systems, this issue is further amplified: frequent compactions, writes, and mixed workloads cause fluctuations in access locality, making naive admission highly inefficient. Caching every key after a miss, even once, can lead to evicting hot items that would have otherwise served multiple accesses. Therefore, selective admission is especially beneficial in this context.

Frequency-Based Admission for Point Lookups. Our system integrates a frequency-aware admission policy specifically for point lookups. When a miss occurs, we increment the frequency counter of the key in a compact data structure (e.g., Count-Min Sketch). Instead of immediately admitting the key into the cache, we calculate its normalized importance by comparing its frequency to the global sum of frequencies across missed keys. A key is only admitted if this normalized score exceeds a tunable threshold. To ensure adaptability across workloads, the admission threshold is not fixed but dynamically adjusted by our reinforcement learning (RL) agent. A static threshold can be either too strict or too permissive depending on the workload characteristics, particularly the access skew, query type, and cache size. For instance, consider two workloads following Zipfian distributions with different skewness values: one highly skewed and one closer to uniform. To achieve the best cache utilization for the same cache size, we would need to admit only the top-N hottest keys. However, the frequency ratios corresponding to those top-N keys would differ significantly between the two workloads due to their distinct distributions. This means a fixed threshold might over-admit in one case and under-admit in another. In theory, if we had precise knowledge of the key distribution, we could analytically determine the ideal admission threshold. But collecting and maintaining accurate distributional statistics in real-time incurs significant computational overhead. Instead, our design embraces a lightweight, learning-based approach that adjusts the admission threshold on the fly. This follows a similar philosophy seen in recent learning-based cache eviction strategies [41, 46], where models adapt decisions based on observed access patterns without relying on manually defined heuristics. Our RL module effectively learns the appropriate threshold under different conditions, enabling robust, low-cost, and high-performance cache management across dynamic and heterogeneous workloads. To keep frequency counts bounded and responsive to changing patterns, we adopt a decay mechanism. Once a key's count reaches a saturation point (e.g., 8), all frequencies and the global sum are halved. This ensures that only consistently hot keys are retained while old or bursty keys naturally fade from the cache decision process.

Overall, this lightweight admission scheme acts as a first line of defense against cache pollution. By combining principled frequency tracking with adaptive thresholds, we ensure efficient use of cache space even in highly dynamic, compaction-prone environments.

Partial Admission for Range Lookups. Range lookups are a frequent access pattern in key-value store workloads, especially in analytical and batched queries. However, caching the full result of a scan can severely degrade cache performance. Scans typically touch a large sequence of keys in a short time window, and admitting all of them into the cache can quickly evict high-value entries, particularly those serving point lookups.

This issue becomes even more problematic under mixed workloads. As shown in Figure 6, a short scan of length 16 may evict

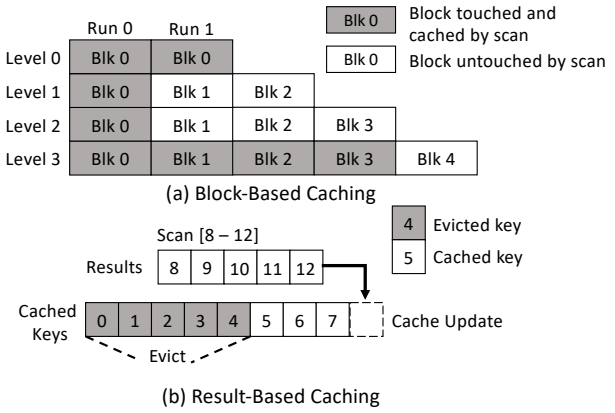

Figure 6: Caching behavior of block-based and result-based strategies.

around 8 data blocks from the block cache, which is more than the ideal 4 blocks (assuming 4 entries per block). This is because the scanned key range overlaps with each sorted run in the LSM-tree, and each run contributes at least one accessed block. For the range cache, a long scan of length 64 can evict 64 key-value entries. If these entries were previously serving point lookups, their eviction would result in up to 64 additional I/O operations. Such scan-related evictions reduce the overall hit rate by displacing frequently accessed items with low-reuse scan results. Since most scan keys are not accessed again, this leads to wasted cache space and more future misses. Consequently, the cache becomes less effective at absorbing point lookup traffic, degrading performance system-wide.

To address this, we propose a scan-aware partial caching policy that selectively caches only a portion of each scan, based on two adaptive parameters: a and b. These parameters are learned and adjusted by the RL agent in response to workload behavior. If a scan's length l is less than a threshold a, we admit the entire scan into the cache. The rationale is that short scans are unlikely to harm cache efficiency and may benefit from full reuse. If the scan length $l > a$, we compute the number of items to cache as $b \cdot (l - a)$. The parameter b controls the aggressiveness of partial caching, effectively determining how many repeated accesses are needed before the full scan range becomes cached. Overlapping scans naturally accelerate this process. By adjusting a and b, the system learns to balance between preserving memory for point lookups and accelerating hot scan ranges. The default value of a is initialized to match the average length of short scans observed in the workload. This strategy ensures that frequent or overlapping scans gradually populate the cache without overwhelming it, while infrequent or long scans have a limited and controlled memory footprint. Note that this strategy can also be applied to the block cache, where the number of blocks instead of the number of keys is controlled.

3.5 Reinforcement Learning-Based Control

At the core of our system is a lightweight reinforcement learning (RL) agent responsible for dynamically adjusting cache management parameters. We choose actor-critic due to its ability to efficiently handle continuous control, such as tuning cache ratios and thresholds, while maintaining low overhead and stable learning—making it well-suited for online deployment in LSM-tree

systems [11, 21, 28]. The actor-critic architecture learns to optimize two key decisions: the allocation ratio between different cache structures, and the admission thresholds for point lookups and range scans. These parameters are difficult to set statically, as optimal values vary significantly with workload composition. For instance, a cache ratio suitable for read-heavy workloads may perform poorly under write-intensive or scan-heavy patterns. Similarly, fixed admission thresholds can either be too strict, missing frequent keys, or too lenient, admitting noise that pollutes the cache.

To achieve online adjustment of cache parameters, several alternative strategies exist, but each comes with limitations in the context of LSM-tree key-value stores. Rule-based approaches and manually tuned heuristics depend on prior knowledge of workload characteristics, which is often unavailable or unreliable in real-world deployments. Analytical models, though effective in controlled environments, tend to break down under the complex and evolving workload patterns typical in LSM-KVS, such as shifting ratios of reads, writes, and scans. Additionally, real-time tracking of workload distributions or computing optimal parameters often introduces significant computational overhead.

In contrast, Reinforcement Learning (RL) offers an adaptive and generalizable solution by learning directly from runtime feedback such as cache hit rates. (1) The relationship between cache hit rate and control parameters (e.g., admission thresholds or cache ratios) is complex and non-linear, making it difficult to model analytically or tune manually. (2) RL can implicitly capture key access distributions and workload dynamics without explicitly collecting or maintaining detailed workload statistics, reducing overhead. (3) RL requires minimal prior knowledge and supports continuous online adaptation, enabling the system to self-optimize in real time and maintain high performance under diverse and evolving workloads. **Model Design.** The actor network receives the current system state as input, which includes cache statistics (e.g., occupancy, hit/miss ratios) and workload patterns (e.g., access frequencies, scan-to-point ratios, and scan length distributions). It outputs cache control actions, specifically: (1) the memory partitioning between block cache and range cache, and (2) the caching policy parameters for partial scan admission. The critic network estimates the value of the current policy to guide learning. Training is performed online with optional offline pretraining. The reward signal is based on the cache hit rate, but because the range cache stores query results rather than physical data blocks, conventional cache hit measurements are not directly applicable. Instead, we estimate the total number of block I/Os that would occur without any caching and derive a normalized reward.

Reward Calculation. Let the notations be defined as in Table 1. For block cache, the hit rate can be conveniently calculated by

$$h = 1 - \frac{\mathrm{IO}_{\mathrm{miss}}}{\mathrm{IO}},$$

where IO represents the total number of data block I/Os without caching. However, in query-result-caching systems, IO is not directly observable. To address this, we estimate the total number of block I/Os without caching, denoted as $\mathrm{IO}_{\mathrm{estimate}}$, given by

$$\mathrm{IO}_{\mathrm{estimate}} = p \times \mathrm{IO}_{\mathrm{point}} + s \times \mathrm{IO}_{\mathrm{scan}},$$

where the total number of I/Os is computed as the sum of I/Os caused by point lookups and those caused by range scans.

For point lookups, bloom filters are typically deployed in LSM-tree systems to reduce the number of disk accesses during the queries. The average number of I/Os per point lookup can be estimated as:

$$\mathrm{IO}_{\mathrm{point}} = 1 + \mathrm{FPR},$$

where FPR denotes the false positive rate of the Bloom filter. Bloom filters are highly memory-efficient, and in our setting, we assume the FPR is negligible. For instance, a Bloom filter with 10 bits per key is sufficient to reduce the FPR close to zero. Given a key size of 24 bytes and a value size of 1000 bytes, the memory cost of such a Bloom filter accounts for only around 1.2% of the total database size. While a memory trade-off exists between the Bloom filter and cache (particularly when using a very limited Bloom filter budget of 1–5 bits per key), this scenario typically arises only under highly constrained memory and point-lookup-heavy workloads, which are uncommon in practical deployments. Therefore, we omit this trade-off and focus our optimization efforts solely on the use of cache memory. For range scans, the I/O cost consists of two parts. First, a seek phase initializes iterators over all sorted runs that overlap with the requested key range. In RocksDB's leveled compaction scheme, this results in an average of $(L-1)+r$ iterators. To estimate the number of sorted runs, we adopt a model based on 1-leveling structures commonly used in systems such as RocksDB. The estimated number of runs, r, is given by

$$r = L - 1 + \frac{r_0^{\max}}{2},$$

where L denotes the number of levels and $r_0^{\max}$ represents the maximum number of runs in Level 0, usually defined by write stall trigger factor. Second, each iterator traverses data blocks until the desired scan length l is satisfied. Assuming each block contains B entries, the total number of I/Os per scan is:

$$\mathrm{IO}_{\mathrm{scan}} = \frac{l}{B} + \left(L + \frac{r_0^{\max}}{2} - 1\right).$$

By substituting the above into the total I/O expression, we obtain:

$$\mathrm{IO}_{\mathrm{estimate}} = p \times (1 + \mathrm{FPR}) + s \times \frac{l}{B} + s \times \left(L + \frac{r_0^{\max}}{2} - 1\right).$$

The estimated cache hit rate h_{estimate} is calculated by comparing the number of avoided block I/Os against the no-cache baseline:

$$h_{\mathrm{estimate}} = 1 - \frac{\mathrm{IO}_{\mathrm{miss}}}{\mathrm{IO}_{\mathrm{estimate}}},$$

Symbol	Definition
p	Number of point lookup queries
s	Number of scan queries
l	Average scan length (number of keys)
B	Number of entries per block
L	Number of levels in the LSM-tree
r	total number of sorted runs in the LSM-tree
r_0	Average number of sorted runs in Level 0
$r_0^{\max}$	Maximum number of sorted runs in Level 0
IO	Total number of block I/Os without caching
$\mathrm{IO}_{\mathrm{estimate}}$	Estimated block I/Os without caching
$\mathrm{IO}_{\mathrm{miss}}$	Actual measured block I/Os after cache misses
$\mathrm{IO}_{\mathrm{point}}$	Average I/Os caused by a point lookup operation
$\mathrm{IO}_{\mathrm{scan}}$	Average I/Os caused by a range lookup operation
h	Cache hit rate
h_{estimate}	Estimated cache hit rate

Table 1: Summary of symbols used in cache control and reward calculation.

where IO_{miss} is the measured number of block I/Os after cache misses and $IO_{estimate}$ is the estimated number of block I/Os without caching. This estimation method can be used to calculate the hit rate for both block cache and range cache without requiring knowledge of the actual number of I/Os. Its accuracy has been validated in the context of block cache. The hit rate h can be expressed as:

$$h = 1 - \frac{IO_{miss}}{IO} = 1 - \frac{IO_{miss}}{IO_{estimate}} = h_{estimate}.$$

By using $h_{estimate}$ as the reward, our RL agent is able to continuously adapt cache partitioning and admission parameters to maximize real-time cache efficiency without any offline profiling or manual workload tuning. To ensure stability during learning, we do not feed the raw hit rate $h_{estimate}$ directly as the reinforcement learning reward. Instead, we apply an exponential smoothing mechanism to gradually adjust the reward signal over time. Specifically, the reward is updated at each step as

$$h_{smoothed} \leftarrow \alpha \times h_{smoothed} + (1 - \alpha) \times h_{estimate},$$

$$reward \leftarrow \frac{\Delta h_{smoothed}}{h_{smoothed}},$$

where $\alpha \in [0, 1]$ is the smoothing factor. A larger α places more weight on past rewards, leading to slower adjustments, while a smaller α makes the reward more responsive to recent hit rate changes. Even under static workloads, the reuse distance of keys can vary slightly over time due to natural randomness in key access patterns. These small shifts can cause temporary spikes or drops in the measured cache hit rate. Frequent fluctuations in hit rate can lead to frequent cache boundary adjustments, resulting in increased evictions and degraded system performance. To avoid the reinforcement learning agent overreacting to such short-term fluctuations, we apply smoothing to the hit rate signal before using it as the reward. By stabilizing the reward input, we ensure that cache tuning decisions are based on long-term trends rather than transient noise.

Adaptive Learning Rate. In order to accelerate model converge, as well as increase learning stability, an adaptive learning rate for the actor is deployed. The learning rate lr is updated by the following equation at the beginning of every window.

$$lr = lr \times (1 - reward).$$

This formula is designed to adapt the learning rate based on workload dynamics: increasing it when a workload change is detected and decreasing it when the workload remains stable. When a workload shift occurs, the hit rate drops, resulting in a negative reward and a higher learning rate. This encourages the model to explore more aggressively and escape local optima in search of a new global optimum. Conversely, when the workload remains stable, the reward becomes positive, and the learning rate gradually decreases over time, promoting faster convergence.

3.6 Pretraining

We incorporate a lightweight pretraining phase to initialize the actor model before deployment. This pretraining can be conducted in either a supervised or unsupervised manner. In the supervised setting, the model is trained using a collection of representative workload vectors paired with target configurations, where the target values can be obtained through controlled experiments. In the unsupervised setting, the model follows the same reinforcement learning process as in the online phase, learning directly from reward signals without explicit labels. The representative workloads can be gathered from deployed databases or manually crafted to simulate edge cases such as rapidly changing access patterns. Once trained, the model is saved and used at runtime, reducing startup overhead and improving early-stage learning stability.

Pretraining offers several advantages: (1) Portability and scalability: A pretrained model can be deployed across machines, avoiding per-machine retraining. (2) Efficiency: It reduces the computational cost during deployment, especially on resource-constrained systems. (3) Stability: It enables faster adaptation to dynamic workloads by providing the model with a well-informed initialization, thus avoiding long warm-up periods during online learning.

4 Implementation

In this section, we explain the design and implementation details of AdCache, focusing on the model location, computational overhead, memory, and concurrency support.

4.1 Model Location

The reinforcement learning model used in AdCache resides on the CPU, meaning that training, inference, and model storage all consume CPU resources. While GPUs are commonly used to accelerate deep learning tasks, we argue that placing our model on the CPU is a more appropriate choice in our context.

Running learning computations on CPUs is a standard practice in learning-based caching systems, as seen in prior works such as LeCaR [41] and Cacheus [37]. Our model is relatively small and lightweight, making it unlikely to benefit from the parallelism offered by modern GPUs. Moreover, transferring data between CPU and GPU would introduce additional latency, which could negatively impact system performance. Importantly, most production environments running databases like RocksDB do not have GPUs available. Therefore, deploying the model on CPU is a more practical and efficient design decision.

4.2 Computational Overhead

Integrating reinforcement learning into a cache management system presents a key challenge: model inference and training must not interfere with the performance of normal query processing. Cache systems must serve point lookups and scans with minimal latency, and any noticeable computational overhead from online learning could degrade system responsiveness.

To address this, we adopt a window-based control mechanism that amortizes reinforcement learning computation over time. Incoming queries are grouped into fixed-size windows, typically 1000 operations each. At the end of each window, we collect workload statistics such as the ratio of point lookups to scans, average scan lengths, and cache hit rates. These statistics serve as the input state for the actor-critic model. The cache control actions used during each window are based on the predictions made at the end of the *previous* window. Specifically, the model output from the previous window is retrieved and applied throughout the current window, while the newly collected statistics are used to update the model asynchronously. In other words, cache parameter updates are always one window behind the latest observed workload.

In scenarios with limited computational resources, like multi-client environments, the window size can be manually configured to a larger value. This reduces training invocation frequency and ensures minimal training impact on client threads.

This design ensures that model inference and training are fully decoupled from the main serving path. Reinforcement learning computation runs in the background, without blocking query execution. As a result, the computational overhead of online learning becomes negligible relative to the cost of serving normal database operations. Detailed evaluations on the computational overhead can be found in Section 5.4.

4.3 Memory Overhead.

Besides computational overhead, reinforcement learning-based cache control introduces some additional memory consumption.

In our system, both the actor and critic networks are lightweight fully connected neural networks with a hidden dimension of 256. Each network consists of an input layer, two hidden layers, and an output layer, using 32-bit floating-point precision for all parameters. The total number of parameters across both models is roughly 140,000, resulting in a combined memory usage of approximately 550 KB for storing network weights. This overhead is negligible compared to typical cache sizes, which are often on the order of tens to hundreds of megabytes. When online training is enabled, additional memory is required to store gradients and optimizer states. For each parameter, the Adam optimizer maintains two auxiliary tensors (first and second moment estimates), and backpropagation temporarily stores gradients. As a result, the total memory overhead during online training is approximately four times the model parameter size, amounting to around 2 MB in total, as shown in Table 2.

Compared to the size of typical cache allocations, often several gigabytes, this memory footprint is negligible. Even under continuous online training, the memory consumption remains small and does not impact system scalability or efficiency.

Component	Estimated Memory Usage
Actor + Critic model parameters	550 KB
Gradients (Actor + Critic)	550 KB
Optimizer states (Adam)	1.1 MB
Total memory overhead	**2.2 MB**

Table 2: Memory overhead of reinforcement learning model and online training.

4.4 Concurrency Control

There is no concurrency issue in single-client scenarios, as operations on the range cache and block cache are executed sequentially without conflict. However, in multi-client environments, concurrency control becomes necessary. To address this, we implemented a sharded range cache architecture, similar to RocksDB's block cache, since the original Range Cache design does not support multi-threaded access by default. The database key space is partitioned into multiple shards, each guarded by its own lock to manage concurrent access. Operations on different shards are independent, enabling safe and efficient parallelism. Furthermore, since the system is highly I/O-bound under multi-client workloads, the latency introduced by lock contention is negligible.

5 Evaluation

We first describe our experimental setup, including the workloads and baselines used for comparison. We then present detailed results and analysis across both static and dynamic workloads.

5.1 Experimental Setup

AdCache is implemented on Rocksdb, and our code is available at Github. We conduct our experiments on a machine running Ubuntu 22.04, equipped with an Intel Core i9-13900K CPU (36 MB L3 cache), 128 GB of RAM, and a 2 TB NVMe SSD. The LSM-tree storage engine is configured to use a 1-leveling compaction policy with a size ratio of 10 between levels. A Bloom filter with 10 bits-per-key is enabled to optimize point lookups. The key size is set to 24 bytes and the value size to 1000 bytes, and the total database size is 100GB. Each SSTable file is configured to be 4 MB in size, and each data block within an SSTable is 4 KB. Write slowdown is triggered at 4 level 0 files, and write stop is triggered at 8 level 0 files. Online training of AdCache is triggered at the end of every window containing 10^3 operations. The actor-critic reinforcement learning module is configured with an initial actor learning rate of 1×10^{-3} and a critic learning rate of 1×10^{-3}. By default, α is set to 0.9 to emphasize long-term cache hit rate improvements over immediate gains, the access pattern follows Zipfian distribution with 0.9 skewness, the queries are done by a single client and AdCache uses 25% cache with no pretraining unless specified in the experiment. For evaluations involving latency measurements, direct I/O is enabled for SST file reads to bypass the operating system page cache. This ensures that all measured latencies reflect the storage engine's internal caching behavior without external memory effects.

To highlight the necessity of a unified and workload-aware design, we include *Range Cache with LeCaR* and *Range Cache with Cacheus* as representative baselines that naively combines a learning-based eviction strategy with an LSM-tree cache structure. LeCaR [41] and Cacheus [37]are well-known reinforcement learning-based policies designed for general-purpose caches, and is often cited as lightweight yet effective alternatives to traditional eviction strategies. Further description of them can be seen in Section 6. Despite that LeCaR and Cacheus perform well generally, they are not designed with the characteristics of LSM-tree systems in mind, such as compactions and invalidation. These baselines serve as an example of a straightforward application of RL to caching in LSM-KVS, without deeper integration or adaptation to LSM-specific behaviors. By comparing with this baseline, we demonstrate that simply augmenting an existing LSM cache with an RL-based eviction policy is insufficient. In contrast, our design integrates RL holistically—jointly tuning structural choices, admission policies, and parameter thresholds based on workload dynamics—resulting in significantly better performance across diverse workloads.

We evaluate the following cache management strategies:

- **RocksDB (Block Cache)**: The default block caching mechanism used in RocksDB, which caches raw data blocks directly based on their disk layout.
- **KV Cache**: A key-value cache that stores the results of point lookups, allowing fast retrieval without disk access. Only point queries benefit from this cache, while scans still access underlying data blocks.
- **Range Cache**: The design proposed in [43], which caches the results of range queries in a logical key-value structure (skip list) to eliminate cache invalidations caused by

compactions. Since Range Cache is not open-source, we reimplement it following the description in the original paper.

- **Range Cache with LeCaR**: A variant of Range Cache where the traditional LRU eviction policy is replaced by LeCaR, an online learning-based caching policy. This baseline evaluates the effectiveness of a naive combination of ML-based eviction strategies and LSM-tree workloads.
- **Range Cache with Cacheus**: A variant of the Range Cache in which the default LRU eviction policy is replaced with Cacheus. Cacheus is a successor to LeCaR, enhanced with scan and churn workload supports.
- **AdCache**: Our proposed system, which uses reinforcement learning to dynamically adjust cache partitioning and admission policies based on observed workload patterns.

5.2 Evaluation on Static Workloads

To investigate how different caching schemes perform under varying cache sizes and workload types, we evaluate all methods on four representative static workloads. These workloads are designed to isolate specific access patterns:

- **Point Lookup**: Consists solely of point queries.
- **Short Scan**: Performs range scans of fixed length 16.
- **Balanced**: Contains an even mix of 33% point lookups, 33% short scans, and 33% writes.
- **Long Scan**: Performs range scans of fixed length 64.

This setup allows us to analyze how each caching strategy responds to specific access patterns and how their effectiveness scales with available cache capacity. The evaluation results are presented in Figure 7, where subfigures (a), (b), (c), and (d) correspond to the Point Lookup, Short Scan, Balanced, and Long Scan workloads, respectively.

Point Lookup. In the point lookup workload, both Range Cache and KV Cache behave identically, functioning as pure key-value caches with an LRU eviction policy. Block Cache, by contrast, stores data in blocks rather than individual key-value entries. This can lead to inefficient memory usage when infrequently accessed keys occupy space within cached blocks, resulting in a lower overall hit rate. When the LRU policy in Range Cache is replaced by LeCaR and Cacheus, the hit rate improves by approximately 3% and 8% respectively under small cache sizes, demonstrating the potential benefit of machine learning-based (ML-based) eviction strategies in general pattern workloads. AdCache consistently achieves the best or equally-best hit rate across all cache sizes, highlighting the value of filtering out infrequently accessed keys through its admission control mechanism. Compared to Range Cache, AdCache provides up to a 9% improvement in hit rate, and up to 14% compared to Block Cache, reducing the number of SST reads by up to 25%.

Short Scan. In short-scan workloads, KV Cache fails to cache results due to the nature of the access pattern. Surprisingly, Range Cache—whether using LeCaR or Cacheus—underperforms compared to Block Cache. Although scans are involved, the access pattern resembles point lookups, with each item spanning multiple blocks. Range-based caching offers little benefit here, as partial hits still incur the full cost of an LSM-tree seek. AdCache adapts effectively by converting the entire range cache into a block cache, recognizing the superior performance of block-based strategies in this setting. Its partial admission policy allows full scan results (e.g., 16+ entries) to be cached per query, making its hit behavior and performance nearly identical to that of the block cache.

Balanced Workload. In the balanced workload, KV Cache only serves point lookups, yielding the lowest hit rate. Range Cache performs worse under small cache sizes due to the high cost of partial scan misses, but its hit rate approaches that of Block Cache as the cache grows. LeCaR and Cacheus often underperform compared to Range Cache, despite using learning-based eviction. AdCache starts with block cache at small sizes and shifts to range cache as capacity increases. Its admission control helps filter infrequent keys, leading to higher hit rates at larger sizes. Specifically, it achieves a 6% hit rate gain over Block Cache, reducing SST file reads by 16.2%.

Long Scan. In the long-scan workload, fully caching infrequent scans leads to high eviction and low hit rates. Range Cache with LRU or LeCaR underperforms Block Cache across all sizes. However, Cacheus and LeCaR improve hit rates under small caches, demonstrating the advantage of learning-based eviction. AdCache applies partial admission at all cache sizes, limiting memory usage from infrequent queries. As a result, AdCache reduces SST file reads by about 17.2% compared to RocksDB block cache.

5.3 Evaluation on Dynamic Workloads

To evaluate the adaptability of caching strategies under evolving access patterns, we construct a dynamic workload consisting of six workload phases executed sequentially: A → B → C → D → E → F. The operation ratios for each phase are shown in Table 3. These workload phases are designed to emulate realistic usage patterns observed in LSM-tree-based systems, which are widely deployed in write-heavy applications such as time-series databases, log processing platforms, and cloud-native storage systems. We begin with a read-dominant phase to simulate analytical workloads with long scans, then gradually transition into more balanced or write-heavy phases that reflect common ingestion-heavy scenarios. The inclusion of mixed read and write patterns captures the complexity of real-world applications where point lookups, range scans, and write operations often coexist. This progression also aligns with the design motivation of LSM-trees, to handle high write throughput efficiently, while stressing the cache system's ability to adapt to evolving access behaviors over time.

Workload	Get (%)	Short Scan (%)	Long Scan (%)	Put (%)
A	1	1	97	1
B	1	49	49	1
C	49	49	1	1
D	25	25	1	49
E	1	49	1	49
F	1	12	12	75

Table 3: Dynamic workload phases used in evaluation.

Each phase runs for a fixed number of 50 million operations. We monitor both cache hit rate and end-to-end throughput of the workload sequences, as shown in Figure 8. Throughput is measured in the form of query per second (QPS). The rankings of throughput and hit rate are shown in Table 4.

Across most workload phases, AdCache ranks as the best or among the top-performing schemes. In read-heavy workloads (A, B, and C), it favors block cache, while in write-heavy workloads (D, E, and F), it switches to range cache, mirroring the best cache

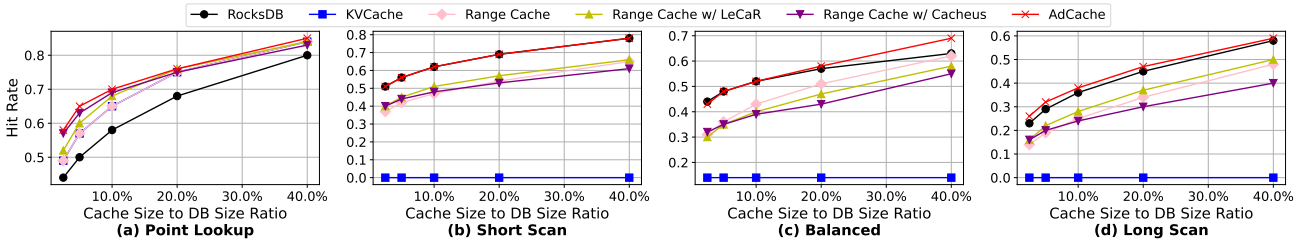

Figure 7: Hit rate of different caching strategies with varying cache sizes under static workloads.

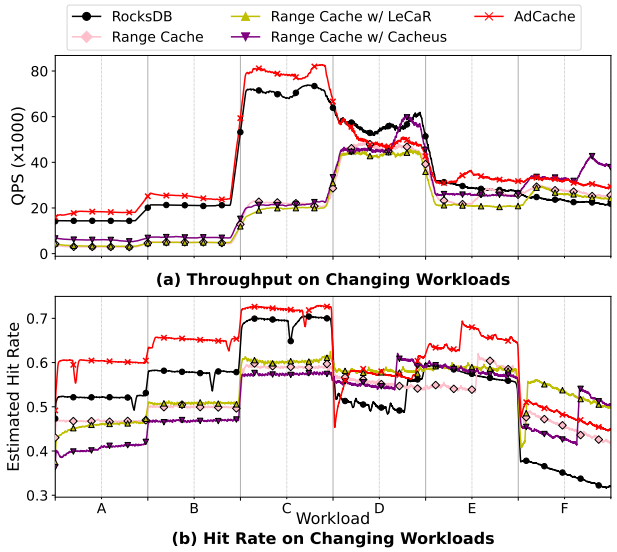

Figure 8: Throughput and hit rate of different caching strategies under dynamic workloads.

Workload	RocksDB	Range Cache	LeCaR	Cacheus	AdCache
A	2/2	5/3	4/4	3/5	1/1
B	2/2	5/4	4/3	3/5	1/1
C	2/2	3/4	5/3	4/5	1/1
D	1/5	4/4	5/1	3/3	2/2
E	2/4	4/5	5/2	3/3	1/1
F	5/5	3/4	4/1	1/3	2/2
Average	2.3/3.3	4.0/4.0	4.5/2.3	2.8/4.0	1.3/1.3

Table 4: Rankings of caching schemes (throughput/hit rate) across dynamic workload phases. Lower is better.

choice for each setting. A temporary drop in hit rate is observed during the transition from workload C to D, reflecting the structural switch. Further analysis of this transition is provided in Section 5.4.

RocksDB's block cache performs best in read-heavy workloads, benefiting from its alignment with the on-disk data structure. In contrast, Range Cache outperforms block cache in write-heavy workloads due to its resilience to cache invalidation during compaction. Range Cache with LeCaR and Cacheus generally matches vanilla Range Cache in hit rate, but shows improved throughput in some cases, like workload F, highlighting the potential of RL-based caching strategies. AdCache, with its adaptive cache partitioning and admission control, consistently achieves the highest overall throughput and hit rate. In write-heavy and long-scan workloads, it delivers a 25%–37% improvement in throughput

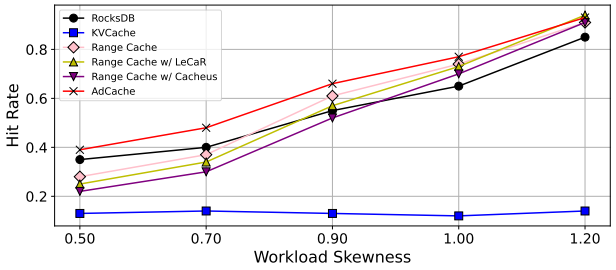

Figure 9: Hit rate of different caching strategies under varying workload skewness.

over RocksDB. However, its performance drops in workload D, mainly due to (1) incomplete cache adaptation following a workload shift, and (2) overhead from frequent insertions into the range cache's skip list, combined with sequential access to both range and block caches.

5.4 Understanding AdCache Behavior

To further investigate the caching behavior in LSM-tree systems and the effectiveness of AdCache, we conduct a series of micro-benchmark experiments.

Workload Skewness. We vary the point lookup and range scan skewness of the Zipfian workloads, and the experiment results are shown in Figure 9. We use the same workload ratio with 50% update and equal amount of point lookups and short scans.

As expected, most caching strategies benefit from higher skewness due to stronger access locality. KVCache, however, remains largely insensitive to skewness, achieving low and flat hit rates across all settings due to its inability to capture range scans under mixed workloads. Range Cache with LRU, LeCaR, and Cacheus achieve lower hit rates than block cache under low skewness, but outperforms it under high skewness. This is because block cache stores infrequent keys in the same blocks along with frequent keys. While this characteristic benefits performance under random access patterns (low skewness), it becomes a limitation when the workload exhibits a clear separation between hot and cold keys, as block cache may waste memory on less useful data.

AdCache consistently outperforms all baselines across the entire skewness spectrum. It achieves a 77% hit rate at skewness 1.0 and up to 93% at skewness 1.2, showing strong adaptivity to highly localized workloads. Compared to RocksDB's block cache, AdCache achieves up to a 12% improvement in cache hit rate and reduces SST file reads by as much as 34.3%.

Training Parameters. To evaluate the impact of learning parameters on the model's convergence and stability, we varied the window size and α in the reward function and observed the model's behavior during a workload shift. Specifically, the

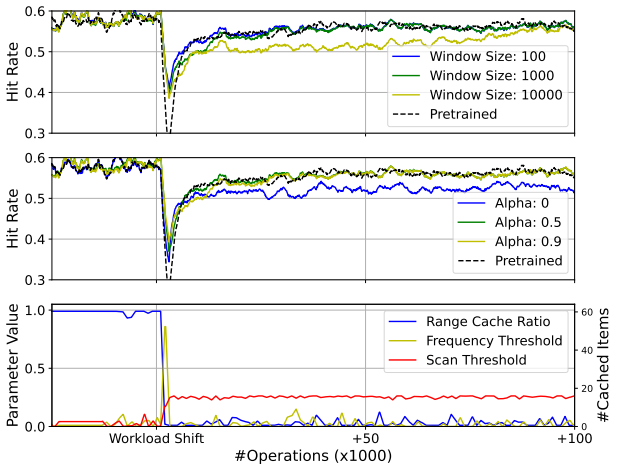

Figure 10: Impact of model parameters on model convergence.

system is warmed up under a read-heavy workload, and then transits into a short-scan-heavy workload. The results are shown in Figure 10.

The first figure shows the impact of window size, which controls how often the model updates. All experiments use a fixed α of 0.9. The "pretrained" curve represents a pretrained model with no online learning, updating cache settings every 1000 queries. AdCache adapts to the workload shift under all window sizes, but convergence is slower with a window size of 10,000 due to infrequent updates. The pretrained model experiences a sharper hit rate drop, as it lacks both online adaptation and reward smoothing. Window sizes of 100 and 1000 perform similarly in terms of hit rate, but we adopt 1000 as the default for its slightly lower overhead.

The second figure shows the impact of the smoothing factor α on reward calculation, with all experiments using a window size of 1000. The "pretrained" model, as before, updates cache settings periodically without online learning. All settings eventually converge after the workload shift, except $\alpha = 0$, which settles into a suboptimal configuration due to overreacting to short-term hit rate fluctuations. The pretrained model shows the sharpest drop, followed by $\alpha = 0$, which lacks smoothing but still learns gradually. Models with $\alpha = 0.5$ and 0.9 behave similarly, achieving stable convergence, indicating the system is not highly sensitive to the exact α value as long as smoothing is present.

The third figure shows how cache parameters evolve over time, using a window size of 1000 and $\alpha = 0.9$. The range cache ratio (blue) starts near 100% for point-lookups and drops to around 0% after transitioning to a short-scan-heavy workload, consistent with prior findings that block cache performs better in short scans. The frequency threshold for point lookups (yellow) remains near 0, admitting most non-one-off keys, with a brief spike during the transition as the model tries to probe its impact. The scan threshold (red), derived from a and b, stabilizes around 16–18 in the new workload, matching the scan length of 16.

Training Overhead. To analyze the computational overhead introduced by background training, we gradually increased the number of client threads and measured the system throughput. The results are presented in Figure 11(a). As the number of clients increases from 1 to 32, the per-client QPS remains largely unaffected. This indicates that the background training process does

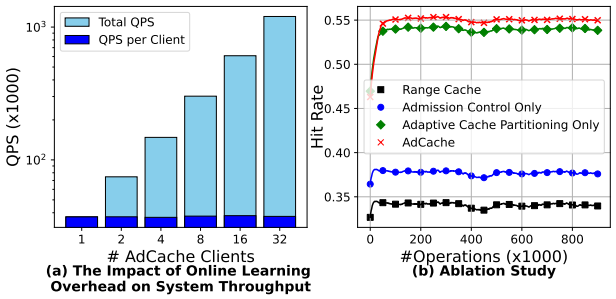

Figure 11: Learning overhead and ablation study of AdCache.

not introduce significant interference. Furthermore, the per-client QPS is not notably constrained by the number of CPU cores, as the system's primary bottleneck lies in I/O throughput rather than computation.

Ablation Study. To demonstrate the effectiveness of both admission control and adaptive cache partitioning, we evaluated AdCache with only one of the two components enabled at a time. The results are presented in Figure 11(b).

The black line at the bottom represents Range Cache, which yields the lowest hit rate among all configurations. The blue line corresponds to AdCache with only admission control enabled. In this setting, partial admission is applied to scan operations, allowing a portion of each long scan (approximately 16-20 entries) to be cached upon access, thereby improving effectiveness without overwhelming the cache. This approach results in an 11% increase in hit rate over Range Cache. The green line illustrates the effect of enabling only adaptive cache partitioning. Under long-scan workloads, block cache has proven more effective, as shown in earlier experiments. As a result, AdCache reallocates memory in favor of block cache, effectively converting range cache into block cache. This adjustment yields a 55% improvement in hit rate. Finally, the red line at the top shows the full AdCache configuration with both components enabled, achieving a 61% hit rate improvement over Range Cache.

6 Related Work

Analytical Models and Theoretical Cache Studies. The study of caching has a long history, with many efforts focused on improving cache efficiency through analytical modeling and smarter eviction strategies. Most analytical approaches model cache behavior using statistical properties such as access frequency, recency, reuse distance, and object size.

Early works [1, 12, 39] introduced stack distance distributions to analytically model cache hit rates under various replacement policies. Reuse distance [13, 22, 26] became a central concept for understanding temporal locality in workloads, and was later extended into absolute reuse distance [3] to better model policies beyond LRU. Eviction policies have also evolved from traditional heuristics toward more adaptive techniques. LRFU [25] and subsequent policies [14, 15, 33] combine recency and frequency signals to improve eviction decisions. Other works incorporate object size into eviction priority [6, 9, 23, 29], addressing the imbalance caused by large entries consuming disproportionate cache space.

With the rise of machine learning, several studies [4, 5, 8, 18, 19] explore training models on individual cache objects to predict reuse distance or future access probability. GL-Cache [46] generalizes this idea by grouping objects to reduce computational

overhead. Another line of research learns from policy experts: for example, LeCaR [41] maintains adaptive weights over LRU and LFU, selecting eviction candidates based on learned confidence in each policy and updating weights using hits in eviction history. Cacheus [37] is a successor to LeCaR, designed to extend LeCaR's capabilities of LRU and LFU policies by adding support for scan and churn workloads. Other approaches such as LHD [2] aim to learn directly from access distributions to estimate object utility and rank items accordingly.

Caching in Real Systems. Key-value stores (KVS) are a foundational storage abstraction widely used in databases, distributed systems, and cloud services. Recent research has explored the deployment of KVS architectures across diverse hardware platforms, including traditional spinning disks, flash-based SSDs, and persistent memory. While caching strategies across these systems share the common goal of improving hit rate, they often optimize for distinct performance objectives. For instance, in-network and memory-disaggregated caching systems [24, 30, 44] are designed to minimize round-trip latency and reduce computational overhead. In-memory multi-core KVS designs [32] emphasize reducing DRAM contention and improving parallelism by employing techniques such as shared-prefix concurrent trees. To address the hardware bottleneck of limited CPU cache growth, [42] propose an application-level mechanism that indirectly controls CPU cache behavior without hardware modification.

7 Conclusion

In this paper, we presented AdCache, an adaptive caching system for LSM-tree-based key-value stores that combines reinforcement learning with lightweight admission control and dynamically choose between block cache and range cache layouts. By learning workload characteristics online, AdCache effectively balances caching strategies for point lookups and scans, while avoiding the overhead of unnecessary data eviction.

Through extensive evaluation, AdCache consistently outperforms existing caching schemes across a wide range of workloads. It achieves up to 14% higher cache hit rate compared to traditional block cache, and reduces the number of SST file reads by up to 25%. Our results demonstrate that intelligent, learning-based cache partitioning and selective admission provide a promising direction for improving storage system performance under dynamic and mixed workloads.

8 Artifacts

To facilitate reproducibility and further exploration, we provide the full implementation of AdCache, including source code, workload generators, and experiment scripts, in our public GitHub repository: https://github.com/qingshanlanshan/AdCache-LSM. The repository includes detailed instructions on how to configure, build, and run the experiments described in this paper. Please refer to the `README.md` file in the repository for setup instructions, system requirements, and usage examples.

Acknowledgment

This research is supported by NTU-NAP startup grant (022029-00001) and Singapore MOE Tier-2 Funding (MOE-T2EP20224-0005).

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. 1989. An analytical cache model. *ACM Trans. Comput. Syst.* 7, 2 (May 1989), 184–215. doi:10.1145/63404.63407

[2] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. {LHD}: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 389–403.

[3] Nathan Beckmann and Daniel Sanchez. 2016. Modeling cache performance beyond LRU. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 225–236. doi:10.1109/HPCA.2016.7446067

[4] Daniel S. Berger. 2018. Towards Lightweight and Robust Machine Learning for CDN Caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks* (Redmond, WA, USA) *(HotNets '18)*. Association for Computing Machinery, New York, NY, USA, 134–140. doi:10.1145/3286062.3286082

[5] Adit Bhardwaj and Vaishnav Janardhan. 2018. Pecc: Prediction-error correcting cache. In *Workshop on ML for Systems at NeurIPS*, Vol. 32. 9–1.

[6] Pei Cao and Sandy Irani. 1997. {Cost-Aware}{WWW} Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems (USITS 97)*.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008), 26 pages. doi:10.1145/1365815.1365816

[8] Zheng Chang, Lei Lei, Zhenyu Zhou, Shiwen Mao, and Tapani Ristaniemi. 2018. Learn to cache: Machine learning for network edge caching in the big data era. *IEEE Wireless Communications* 25, 3 (2018), 28–35.

[9] Ludmila Cherkasova. 1998. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy.* Hewlett-Packard Laboratories Palo Alto, CA, USA.

[10] Dong Dai, Xi Li, Chao Wang, Mingming Sun, and Xuehai Zhou. 2012. Sedna: A Memory Based Key-Value Storage System for Realtime Processing in Cloud. In *2012 IEEE International Conference on Cluster Computing Workshops*. 48–56. doi:10.1109/ClusterW.2012.28

[11] Thomas Degris, Martha White, and Richard S Sutton. 2012. Off-policy actor-critic. *arXiv preprint arXiv:1205.4839* (2012).

[12] Chen Ding and Yutao Zhong. 2003. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.* 38, 5 (May 2003), 245–257. doi:10.1145/780822.781159

[13] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (Vancouver, B.C., CANADA) *(MICRO-45)*. IEEE Computer Society, USA, 389–400. doi:10.1109/MICRO.2012.43

[14] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. 2018. Adaptive Software Cache Management. In *Proceedings of the 19th International Middleware Conference* (Rennes, France) *(Middleware '18)*. Association for Computing Machinery, New York, NY, USA, 94–106. doi:10.1145/3274808.3274816

[15] Gil Einziger, Roy Friedman, and Ben Manes. 2017. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)* 13, 4 (2017), 1–31.

[16] Facebook. 2025. Cassandra. https://cassandra.apache.org/_/index.html.

[17] Facebook. 2025. RocksDB. https://github.com/facebook/rocksdb.

[18] Qilin Fan, Xiuhua Li, Jian Li, Qiang He, Kai Wang, and Junhao Wen. 2021. PA-cache: Evolving learning-based popularity-aware content caching in edge networks. *IEEE Transactions on Network and Service Management* 18, 2 (2021), 1746–1757.

[19] Vladyslav Fedchenko, Giovanni Neglia, and Bruno Ribeiro. 2019. Feedforward neural networks for caching: N enough or too much? *acm sigmetrics performance evaluation review* 46, 3 (2019), 139–142.

[20] Google. 2025. LevelDB. https://github.com/google/leveldb/.

[21] Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. 2012. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, part C (applications and reviews)* 42, 6 (2012), 1291–1307.

[22] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 60–71. doi:10.1145/1816038.1815971

[23] Shudong Jin and Azer Bestavros. 2000. Popularity-aware greedy dual-size web proxy caching algorithms. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*. IEEE, 254–261.

[24] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th symposium on operating systems principles*. 121–136.

[25] George Karakostas and Dimitrios Serpanos. 2000. Practical LFU Implementation for Web Caching.

[26] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. 2007. Cache replacement based on reuse-distance prediction. In *2007 25th International Conference on Computer Design*. 245–250. doi:10.1109/ICCD.2007.4601909

[27] Donguk Kim, Jongsung Lee, Keun Soo Lim, Jun Heo, Tae Jun Ham, and Jae W Lee. 2024. An LSM Tree Augmented with B+ Tree on Nonvolatile Memory. *ACM Transactions on Storage* 20, 1 (2024), 1–24.

[28] Vijay Konda and John Tsitsiklis. 1999. Actor-critic algorithms. *Advances in neural information processing systems* 12 (1999).

[29] Conglong Li and Alan L. Cox. 2015. GD-Wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on*

Computer Systems (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 15 pages. doi:10.1145/2741948.2741956

[30] Yi Liu, Minghao Xie, Shouqian Shi, Yuanchao Xu, Heiner Litz, and Chen Qian. 2025. Outback: Fast and Communication-efficient Index for Key-Value Store on Disaggregated Memory. *arXiv preprint arXiv:2502.08982* (2025).

[31] Yi Liu, Ruilin Zhou, Yuhang Gan, and Chen Qian. 2024. SpotKV: Improving Read Throughput of KVS by I/O-Aware Cache and Adaptive Cuckoo Filters. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*. 344–354. doi:10.1109/CLOUD62652.2024.00046

[32] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. 183–196.

[33] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (San Francisco, CA) *(FAST '03)*. USENIX Association, USA, 115–130.

[34] Supriya Mishra. 2024. A survey of LSM-Tree based Indexes, Data Systems and KV-stores. arXiv:2402.10460 [cs.DB] https://arxiv.org/abs/2402.10460

[35] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. doi:10.1007/s002360050048

[36] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. 2022. Closing the B+-tree vs.{LSM-tree} Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 69–82.

[37] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 341–354. https://www.usenix.org/conference/fast21/presentation/rodriguez

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

[39] Rathijit Sen and David A. Wood. 2013. Reuse-based online models for caches. *SIGMETRICS Perform. Eval. Rev.* 41, 1 (June 2013), 279–292. doi:10.1145/2494232.2465756

[40] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. LSbM-tree: Re-Enabling Buffer Caching in Data Management for Mixed Reads and Writes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 68–79. doi:10.1109/ICDCS.2017.70

[41] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving Cache Replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/hotstorage18/presentation/vietri

[42] Kefei Wang, Jian Liu, and Feng Chen. 2020. Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores. *Proceedings of the VLDB Endowment* 13, 9 (2020).

[43] Xiaoliang Wang, Peiquan Jin, Yongping Luo, and Zhaole Chu. 2024. Range Cache: An Efficient Cache Component for Accelerating Range Queries on LSM - Based Key-Value Stores. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 488–500. doi:10.1109/ICDE60146.2024.00044

[44] Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian Tan. 2014. C-hint: An effective and reliable cache management for rdma-accelerated key-value stores. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–13.

[45] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David H.C. Du. 2020. AC-Key: Adaptive Caching for LSM-based Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 603–615. https://www.usenix.org/conference/atc20/presentation/wu-fenggang

[46] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. 2023. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 115–134. https://www.usenix.org/conference/fast23/presentation/yang-juncheng

[47] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. 2020. Leaper: a learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proc. VLDB Endow.* 13, 12 (July 2020), 1976–1989. doi:10.14778/3407790.3407803

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009