Improving Range Scan Performance in LSM-trees with Group Caching

HENGRUI WANG*, Tsinghua University, China
JIAOYI ZHANG*, East China Normal University, China
JIANSHENG QIU, Tsinghua University, China
FANGZHOU YUAN, Tsinghua University, China
HUANCHEN ZHANG†, Tsinghua University, China

Log-structured merge-trees (LSM-trees) are widely used in modern key-value stores, but their multi-level structure reduces lookup efficiency, especially for range scans. Existing caching solutions, like block caches or full query caches, are memory-inefficient because they fail to exploit a critical asymmetry: eliminating an I/O from upper LSM-tree levels requires caching far fewer key-value pairs (KVs) than from lower levels. To address this, we introduce Group Cache, which uses KV Groups, the minimal set of KVs within a block for a specific query, as its fundamental caching unit. By employing a size-aware policy that prioritizes small, high-utility KV Groups, Group Cache maximizes I/O savings per unit of memory. We also address practical challenges like compaction management, intra-group hotness difference, and scalability. Our theoretical analysis and extensive experiments in RocksDB demonstrate that Group Cache significantly outperforms traditional caching methods, achieving up to 3× faster query performance with the same memory budget, or achieving similar performance while using 75% less space.

CCS Concepts: • **Information systems → Database management system engines**; **Data structures**.

Additional Key Words and Phrases: LSM-trees, Caching, Range Scan

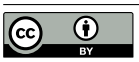
ACM Reference Format:
Hengrui Wang, Jiaoyi Zhang, Jiansheng Qiu, Fangzhou Yuan, and Huanchen Zhang. 2026. Improving Range Scan Performance in LSM-trees with Group Caching. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 47 (February 2026), 26 pages. https://doi.org/10.1145/3786661

1 Introduction

Log-structured merge-trees (LSM-trees) are a foundational data structure for modern key-value stores [2–8, 24, 26, 45] and are widely used in production systems [9, 18, 22, 27, 33, 36, 42, 43]. LSM-trees achieve high write throughput by buffering writes in memory and flushing them sequentially to disk as sorted runs, which are later sort-merged via a process called compaction. However, this multi-level structure degrades read performance, particularly for range scans, because the requested items are likely scattered across multiple sorted runs. Recent works [34, 39, 49, 59] have proposed complex compaction policies to trade write efficiency for faster scans for specific workloads. Nevertheless, [34] proves that by simply tuning the compaction policy, it is theoretically

*Both authors contributed equally to this research.
†Huanchen Zhang is also affiliated with the Shanghai Qi Zhi Institute. Corresponding author.

Authors' Contact Information: Hengrui Wang, wang-hr21@mails.tsinghua.edu.cn, Tsinghua University, China; Jiaoyi Zhang, jyzhang@dase.ecnu.edu.cn, East China Normal University, Shanghai, China; Jiansheng Qiu, qjc21@mails.tsinghua.edu.cn, Tsinghua University, China; Fangzhou Yuan, yfz23@mails.tsinghua.edu.cn, Tsinghua University, China; Huanchen Zhang, huanchen@tsinghua.edu.cn, Tsinghua University, China.



ACM 2836-6573/2026/2-ART47
https://doi.org/10.1145/3786661

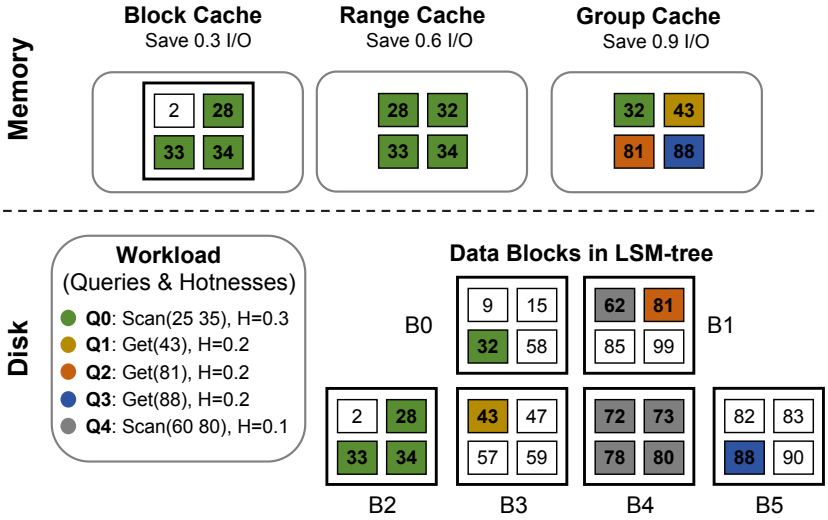


Fig. 1. An example of different cache designs. Each cache could store 4 KVs. The workload contains 5 queries Q0-Q4, and their hotnesses are 0.3, 0.2, 0.2, 0.2, 0.1, respectively. The LSM-Tree contains 6 blocks B0-B5. The accessed KVs of each query are marked in the same color as the query. The Block Cache stores the hottest data block B2, saving 0.3 I/O. The Range Cache stores the result of the hottest query Q0, saving 0.6 I/O. The Group Cache stores the small KV Groups of Q0-Q3, saving 0.9 I/O.

impossible to improve an LSM-tree's scan performance without degrading its write performance, and unfortunately, many real workloads are both write-heavy and scan-heavy, such as LinkBench [9] (31% write, 18.3% get, and 50.7% scan) and Assoc_2ry [11] (56% write and 44% scan). Consequently, efficient range scans remain a primary and unresolved challenge in LSM-tree design.

This paper introduces a simple yet effective method for improving range scan performance based on a key observation: For any given range query, the number of matching keys is highly asymmetric across the levels of an LSM-tree. Upper-level runs typically contain few relevant key-value pairs (KVs), whereas lower, larger levels often contain many. Consequently, the cost of preventing one I/O operation varies dramatically; Caching results from an upper level is inexpensive, while doing so for a lower level requires caching a long sequence of KVs.

Existing LSM-tree caching mechanisms, such as RocksDB's Block Cache [6] and Range Cache [50], are fundamentally limited by their operation-agnostic design. These mechanisms fail to exploit the above *Range-access Asymmetry*. They treat all I/O operations as equivalent and consequently allocate significant cache space to large, lower-level runs, yielding disproportionately small I/O performance gains. This inefficiency is detrimental even in workloads dominated by point queries. Because a single range scan can leave a large data footprint, a hot scan can evict numerous warm cache entries vital for frequent point queries. As shown in our benchmarks on the real-world Prefix_dist workload [11] (14% inserts, 83% point queries, and 3% scans), scanned items account for 58% of the Range Cache usage. This cache-space contention results in high p90/p99 query latencies (including both point and range queries), and the problem is exacerbated with an increasing memory pressure [31] caused by ever-growing data volume.

To leverage the Range-access Asymmetry in LSM trees, we propose the KV Group as a new caching unit. A KV Group is defined as a contiguous sequence of key-value pairs from a data block that are relevant to a range/point query. As shown in Figure 1, a query will access one

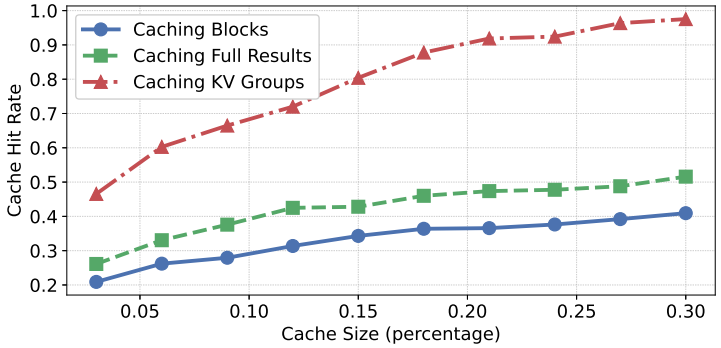


Fig. 2. Static Optimal Performance of different caching granularities. The experiment is conducted on a scan-only workload with a Zipf factor of 0.8. The LSM-tree contains 20 sorted runs (8 at L0), each equipped with a range filter. Given exact knowledge of the workload, we measure the cache hit rate by directly keeping the hottest data blocks, query results, and KV Groups with the highest $\frac{\text{frequency}}{\text{size}}$ ratio in the cache.

or two data blocks in the LSM-Tree, and their target keys are marked as the same color as the query. Colored keys in a data block form a KV Group. It represents the exact set of KVs read by a single I/O. Therefore, while KV Groups differ greatly in size—directly reflecting the Range-access Asymmetry—caching any one of them provides the uniform benefit of saving one I/O. Based on this observation, we introduce Group Cache, a system that uses the KV Group as its basic caching unit. The novelty of Group Cache is that it disaggregates a query's results into their constituent KV Groups and manages each group individually to save more I/O under the same memory budget, as shown in Figure 1. Fundamentally, refining the cache unit with the KV Group increases *static optimal performance* (defined in Section 5.1). As shown in Figure 2, KV Group caching achieves 2× higher static optimal cache hit rate than both block-based and full-result caching.

We further addressed three practical challenges: cache management during compactions, intra-group hotness differences, and scalability. We implemented Group Cache within RocksDB and applied LRU-S [30] cache policy. We evaluated Group Cache using microbenchmarks, YCSB [15], LinkBench [9], and 5 real workloads in [11]. Our results show that Group Cache substantially and consistently outperforms existing caches. It delivers up to 3× the query throughput of the default Block Cache and Range Cache while being far more memory-efficient; achieving similar performance with the Block Cache requires 4× the cache space.

We make three primary contributions in this paper. First, we identify the Range-access Asymmetry in LSM-trees and its implications for caching efficiency. Second, we propose Group Cache, a system that leverages this asymmetry by using variable-sized KV Groups as its basic caching unit. We prove that our approach yields superior static optimal performance compared to existing methods. Finally, we implement Group Cache in RocksDB with three size-aware cache policies and demonstrate experimentally its performance advantages over state-of-the-art approaches under the same memory constraints.

2 Background & Related Work

2.1 Key-Value Stores

LSM-Tree-Based Key-Value Stores. LSM-Tree stores data in multiple levels of sorted runs whose capacities grow exponentially with the size ratio T. Writes are buffered in memory and flushed to level 0; when a level exceeds its budget, background compaction moves data to the next level.

Compaction policy determines the read/write trade-off: Leveling policy keeps a single run per level, and favors reads at higher write cost, while Tiering (Universal) policy allows multiple runs per level and lowers write cost at the expense of read amplification. Many systems use hybrids such as lazy leveling [6, 19, 20]: upper levels use tiering and only the last level uses leveling; when up to $T - 1$ runs accumulate, they are merged into one and pushed to the next level, and if the target is not the final level its contents are left untouched, which reduces write overhead relative to pure leveling [44]. LSM-trees also attach probabilistic membership filters to reduce unnecessary I/O, typically one Bloom filter per SSTable with a tunable false positive rate [14, 17, 21, 63].

B-Tree-Based Key-Value Stores. B-Tree-based key-value stores organize records in a single sorted run and keep inner nodes in memory, so that a point lookup typically needs one I/O, but they suffer from high write amplification. Recent systems revisit update-in-place designs and proposed several methods to improve the write performance for B-Tree-based KV stores. TreeLine [58] combines record-level caching to keep hot tuples in memory, page grouping to turn logical scans into long physical reads, and insert forecasting to reduce page reorganizations. Bf-Tree [25] further observes that access within a page is highly skewed, so caching at page granularity wastes memory and amplifies writes; it decouples buffer-pool entries from on-disk pages and introduces variable-length mini-pages that cache only hot sub-ranges and buffer recent updates. It leverages the skewness in a read-optimized system to optimize the write performance. Differently, we focus on a write-optimized LSM system to optimize the read performance. Although we take a similar direction of leveraging skewness, our observed skewness is a structural property, which is also different from Bf-Tree.

2.2 Range Scan Optimizations in LSM-Trees

Range scan performance remains one of the key challenges in LSM-trees. Range filters [12, 16, 23, 32, 35, 47, 48, 52, 60, 62] have been proposed to partially mitigate this issue by avoiding unnecessary I/Os. These filters determine whether a given query range may contain keys in a sorted run and can skip probing the run if the filter returns a negative result. Modern, powerful filters can reduce the number of unnecessary I/Os to nearly zero, as shown in previous research [48]. However, they cannot help when the target range exists in most of the sorted runs. In [48], the author shows that state-of-the-art range filters can hardly improve the performance of scanning 20 consecutive KVs, even with higher bits-per-key. The reasoning is that the read amplification does not stem from unnecessary accesses, but from the block-level access granularity of SSDs. Even with *ideal* range filters (i.e., filters with zero false positives), the performance improvement remains limited, and the additional computational overhead further reduces their benefit [48].

Compaction is the main technique for improving the performance of range scans in LSM-trees. By clustering related keys into fewer blocks, compaction reduces the number of I/O operations required for range queries. However, this benefit comes at the cost of increased write amplification. Consequently, numerous studies have focused on designing compaction policies that balance read and write efficiency [19, 20, 39]. A recent work, Moose [34], analyzed the trade-off between range query cost and update cost, and proposed a theoretical Pareto frontier that characterizes the fundamental limitations of compaction-based optimization. The conclusion is that optimizing range scan performance through compaction inevitably worsens write performance.

Another optimization in recent RocksDB is asynchronous seek and prefetching [1]. These techniques overlap seek operations across sorted runs and prefetch data blocks into the Block Cache during sequential scans. However, prior work shows that, with range filters, a seek requires nearly a single I/O [49]. RocksDB also reports that prefetching can harm short scans by fetching unnecessary blocks [1]. These optimizations parallelize I/Os rather than reduce them. In our benchmarks on real workloads, asynchronous I/O can hardly improve the performance (Section 7.4).

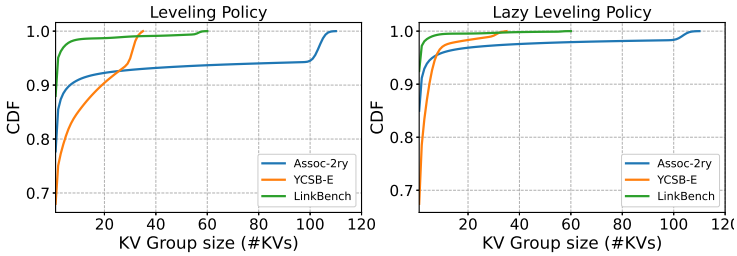


Fig. 3. The KV Group size distribution on real workloads.

2.3 Caching in LSM-Trees

Block cache is the most widely adopted caching scheme in systems like LevelDB [2] and RocksDB [6]. It stores frequently accessed data blocks in memory to accelerate both point lookups and range queries. Typically, it uses a hash table to index cached blocks and an LRU chain to manage eviction. While this approach is simple and practical, it suffers from several critical limitations. First, block cache suffers from poor memory efficiency, as it caches entire data blocks even when a query accesses only a few keys within them. Second, block cache suffers from the cache invalidation problem during compaction [56]. When SSTables are merged together, previously cached data blocks may no longer exist. Some designs [3, 53] use KV granularity instead of block granularity. These KV caches improve memory efficiency but sacrifice the ability to accelerate range queries. So we will not discuss KV caches in this paper.

In [50], the author proposed a novel Range Cache based on query granularity. It caches the entire result of recent range queries. If the same range is queried again, the result can be returned directly from memory without any disk access. This design is effective for workloads with high temporal locality. Nevertheless, the lack of spatial locality still brings Range Cache significant drawbacks under real workloads. For example, two correlated range scans (such as $scan(0, 100)$ and $scan(100, 200)$) could access the same data blocks. However, caching the result of the first query cannot reduce any I/Os for the second one. Range Cache is an intuitive solution, but it ignores the differences among I/Os within the same query.

3 Asymmetry in LSM-Tree Range Scans

Range scans in LSM-Trees suffer from significant performance degradation due to high read amplification. This amplification arises from the need to access multiple sorted runs across different levels, with each access typically incurring a disk I/O. Prior work mainly focused on identifying hot blocks or queries, but overlooked the fact that eliminating an I/O requires a significantly different amount of cache space across all levels of the LSM-Tree.

We first analyze the minimal data unit required to eliminate a single I/O for a range query. In this paper, we assume that the LSM-Tree only probes sorted runs that contain keys in the target range during a range scan[1]. When a range query retrieves key-value pairs (KVs) from a data block, it incurs exactly one I/O, regardless of how many target KVs reside within that block. Caching only a subset of these KVs does not eliminate the I/O, as the block must still be accessed to retrieve the remaining KVs. Therefore, to avoid this I/O, all relevant KVs within the block must be cached.

This all-or-nothing behavior leads us to define a ***KV Group*** as the minimal unit of data that must be cached to eliminate a single I/O for a given query. A KV Group consists of all key-value

[1]This is because most irrelevant sorted runs can be skipped by range filters.

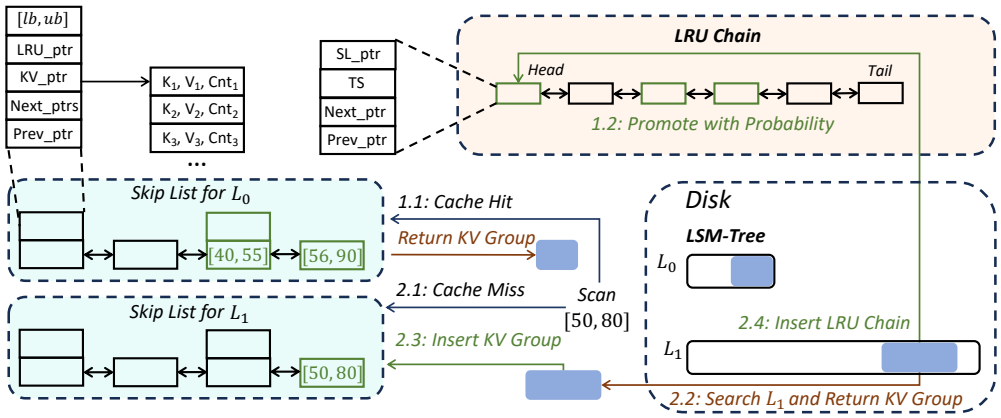


Fig. 4. An overview of Group Cache. The LRU chain and skip lists are stored in memory.

pairs within a data block that fall into the query's range. If a query's target KVs span multiple data blocks, each data block will have a KV Group. Thus, each KV Group contains up to B KVs, where B is the maximum number of KVs per block. These KV Groups are managed independently. For ease of discussion, we assume that a range query maps to at most one KV Group per sorted run. As shown in Figure 4, each blue segment corresponds to a KV Group for the query $scan(50, 80)$.

The Range-access Asymmetry is a structural property that emerges from the exponential growth in sorted run sizes across LSM-Tree levels. When keys follow similar distributions across runs, the expected number of keys within any specific range decreases exponentially from lower to upper levels. Consequently, KV Groups from upper-level sorted runs are typically very small (one or two KVs), while those from lower-level runs are substantially larger, as demonstrated in Figure 1. When distributions differ, the asymmetry usually becomes more pronounced as specific key ranges concentrate in particular sorted runs.

We validate this asymmetry using three scan-heavy real workloads: YCSB-E [15], LinkBench [9], and Assoc_2ry [11]. The KV Group size distributions for these workloads under two compaction policies are shown in Figure 3. A data block typically contains approximately 30, 60, and 100 KVs in YCSB-E, LinkBench, and Assoc_2ry, respectively. Across all settings, the KV Group size distributions are highly skewed, and the skewness becomes more pronounced with the Lazy Leveling policy. The reason is that an LSM-Tree with the Lazy Leveling policy has a single largest sorted run, together with numerous but smaller sorted runs, compared with the Leveling policy. This indicates that the structural asymmetry is amplified in modern LSM-Tree systems that adopt lazy compaction policies [19, 20, 34, 49] like Lazy Leveling.

Consequently, most of the I/Os could be eliminated by caching very few KVs, while others need to cache far more KVs. We consider a range scan retrieving K consecutive keys spanning L levels in an LSM-Tree with leveling compaction and size ratio T between adjacent levels. Given the LSM-Tree's exponential growth property, each level contains roughly T times as many target keys as the level above. So approximately $\frac{K}{T}$ keys are retrieved from the upper $L-1$ levels (requiring $L-1$ I/Os), while the remaining $\frac{(T-1)\cdot K}{T}$ keys come from the last level (requiring 1 I/O). Traditional block caching and full query result caching fail to exploit this asymmetry. They treat all I/Os equally and store $L \cdot B$ keys and K keys respectively to save all L I/Os in future queries. However, if we instead cache only the $\frac{K}{T}$ keys from upper levels—effectively caching the small KV Groups—we can

eliminate $L-1$ I/Os while consuming only $\frac{1}{T}$ of the memory. In conclusion, because KV Groups differ intrinsically, we should manage them individually—prioritizing smaller KV Groups—rather than treating all I/Os as equal, as in Block and Range caches.

4 Group Cache: Exploiting the Asymmetry

This section introduces **Group Cache**, a caching system designed to exploit the range access asymmetry in LSM-Trees. Group Cache uses the *KV Group* as its cache granularity, as KV Groups naturally reflect the range access asymmetry. Group Cache consists of two main components: (1) data structures that store KV Groups in memory, and (2) cache policy that govern their admission and eviction. An overview of Group Cache is shown in Figure 4.

4.1 Group Cache Design

As shown in Figure 4, Group Cache maintains two core data structures: per-run skip lists and a global LRU chain. We adopt **skip lists** in Group Cache to store KV Groups in order, which offers flexible memory allocation and fast search performance. To manage cache admission and eviction, a **global LRU chain** supports size-aware, LRU-based cache policies.

The skip list. Each KV Group has a skip list node, which stores: the group's key range (lb, ub), a pointer to the corresponding LRU node (*LRU_ptr*). As the KV Groups have variable sizes, we store them separately from the corresponding skip list node. The skip list node uses a pointer (*KV_ptr*) to find its KV Group. A per-KV 8-bit access counter (*Cnt*) is stored together with each KV pair for group splitting (see Section 6.1). Within each skip list, KV Groups are sorted by their lower bounds.

As each KV Group is related to a specific sorted run, we need to reconstruct the cached KV Groups during compactions. To localize and speed up this process, we built each sorted run a local skip list to cache the KV Groups in this sorted run. During compaction, we will reconstruct the KV Groups and merge multiple skip lists together. This design mirrors RocksDB's use of per-run Bloom/Range filters. This merge incurs less overhead than filter merging because the skip list is not persisted, whereas filters must be rewritten to SSD.

The LRU Chain. Each LRU node represents a unique KV Group and stores: (1) a pointer to the group's skip list node (*SL_ptr*), and (2) a timestamp (*TS*) marking the most recent promotion to the head of the chain. To evict a KV Group, the system uses the pointer to locate its skip list node and removes both the LRU node and the skip list node. To improve scalability under concurrency, the LRU chain is physically partitioned into shards, but logically managed as a unified global chain based on timestamps (see Section 6.2).

Query Process. In LSM-Trees, each sorted run is queried independently. Given a query range $[lb, ub]$ and a sorted run sr, the system searches sr's skip list to locate relevant KV Groups. The search begins by locating the last node whose lower bound is below lb, and proceeds until the first node whose lower bound exceeds ub. The KV Groups between these nodes potentially overlap with the query. A **cache hit** occurs only when the union of these KV Groups fully covers $[lb, ub]$ (**Step 1.1** in Figure 4), in which case the query proceeds to the next run. Based on the cache policy, the associated LRU node may be promoted with a certain probability (**Step 1.2**). If no matching KV Group is found, this is a **cache miss** (**Step 2.1**). The system then fetches the target data block from disk (**Step 2.2**) and inserts the KV Group into the cache with a certain probability (**Steps 2.3–2.4**). When admitted, we assign the KV Group's range to $[lb, ub]$ to maximize its future coverage, even if no key in the group exactly equals lb. If only part of the query is cached, the uncovered sub-ranges are split and handled as new queries. For example, if $[lb_0, ub_0]$ is partially covered by cached ranges $[lb_0, ub_1]$ and $[lb_1, ub_0]$ (with $ub_1 < lb_1$), the intermediate range $[ub_1 + 1, lb_1 - 1]$ is treated as a separate cache miss.

Cache Policies. KV Groups vary in size, so our cache policy must account for both hotness and memory cost. In our implementation, we use the simplest size-aware LRU variant, LRU-S [30] (see Section 5.2). We will demonstrate, both theoretically and experimentally, that such a policy can closely approach the static-optimal performance when applied at the KV Group granularity. It is worth noting that there are many advanced hotness-modeling algorithms [10, 13, 38, 40, 41, 51, 57, 61]. However, such improvements are orthogonal to our primary contribution, and we also show experimentally that two advanced policies—AdaptSize [10] and Cliffhanger [13]—do not improve performance over LRU-S.

Node Access and Promotion. The above cache policies prioritize smaller KV Groups by promoting or admitting them with a higher probability of access. Each promotion updates the node's timestamp. However, range queries may only access a subset of KVs in a group, causing uneven hotness. To handle this, we define a KV Group as "accessed" only when all of its KVs have been read at least once. Each KV tracks access frequency with a counter. When all counters exceed 1, the group is eligible for promotion. If promotion succeeds, all counters reset to 0; otherwise, they are decremented by 1. This method may evict hot KVs alongside cold ones. To mitigate this, we introduce a **group splitting** mechanism that partitions KV Groups based on access patterns (see Section 6.1).

Overhead Analysis. Group Cache's main time overhead comes from skip-list operations, each taking less than 1 μs, which is negligible compared with a disk I/O of about 80 μs. For space, each LRU node in Group Cache and Range Cache uses 24 bytes of pointers, an 8 byte timestamp[2]. Each Range Cache skip-list node uses 40 bytes of pointers. Besides, each node also needs 4 byte reference and 1 byte flag for concurrent control. In total, the metadata needed to maintain these structures is 77 bytes. For Group Cache, an additional $2 \cdot |key|$ bytes are used to store the lower and upper bounds (lb and ub). Assuming a KV Group contains G KVs, the per-KV metadata is 77 bytes for Range Cache, and $\frac{77+2\cdot|key|}{G} + 1$ bytes for Group Cache, where the +1 accounts for the per-KV counter. When keys are large, Group Cache can omit storing lb and ub in the skip-list node and instead reference the start and end keys already present in the KV Group. When KVs are variable-sized, both Range Cache and Group Cache require an additional 3 bytes per KV (1 byte for the key length and 2 bytes for the value length) to record lengths.

In contrast, Block Cache needs 92 bytes of metadata per cached data block. In our benchmarks with 8 byte keys and 256 byte values, and an average Group size of 3.3 KVs, metadata consumes approximately 2%, 22%, and 10% of total cache space in Block Cache, Range Cache, and Group Cache, respectively. Although Block Cache appears to have smaller metadata, a fixed-size data block can contain many non-target bytes, which reduces space efficiency for range scans. When the KV size becomes smaller, the metadata overheads of Group Cache and Range Cache increase; however, in such cases, a cached data block also tends to include more unrelated data.

4.2 Managing Group Cache During Compaction

Compaction changes the LSM-Tree structure by merging multiple sorted runs into a new one. In this process, the KVs in the affected runs are reorganized; therefore, KV Groups in the old runs are invalidated, since caching them individually no longer saves I/O. As shown in case 2 of Figure 5, the blue segment {10, 31} represents a KV Group related to the range query `scan(5, 40)`. After compaction, caching {10, 31} no longer saves I/O for `scan(5, 40)`. To address this cache-invalidation issue, Group Cache extracts all cached KV Groups from the runs involved in the compaction. If some cached KV Groups are related to the same query[3], they are merged into a new

[2]Although the original Range Cache does not include a timestamp in LRU nodes, we find it necessary for concurrency control.

[3]In practice, KV Groups that overlap in range should be merged even if they originate from different queries. We assume all queries are disjoint for ease of understanding.

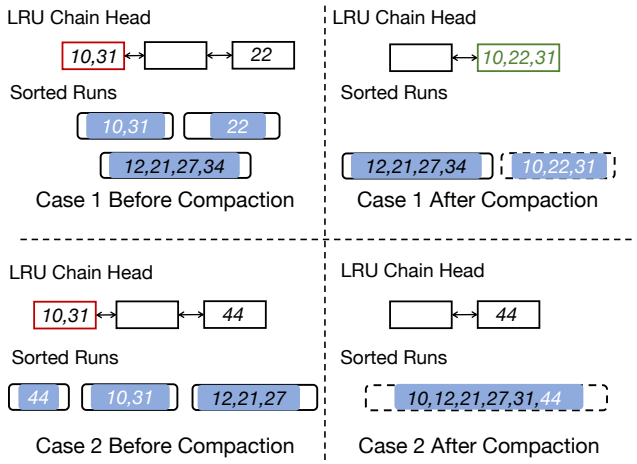


Fig. 5. Merging KV Groups during compaction. Dashed rectangles are sorted runs created by compaction. White numbers in the sorted runs represent cached keys. Red LRU nodes are to be evicted. The green LRU node is newly created.

KV Group. The Group Cache then decides whether to retain the new KV Group. Specifically, there are two different cases as proposed below.

Case 1. The new KV Group fully covers all target KVs related to the query in the newly created sorted run. In this case, we retain the group in the Group Cache, as it can still eliminate I/Os for future queries. If the group remains unchanged by compaction (i.e., it is not merged with others), its LRU node remains in its original position. However, if the group results from merging multiple overlapping KV Groups, we insert its LRU node at the position previously held by the overlapping group's LRU node closest to the tail of the LRU chain. This reflects its reduced priority, as the new group is larger and more costly to cache. The other overlapping groups are removed from both the skip list and the LRU chain.

Case 2. The new KV Group does *not* fully cover all target keys in the newly created sorted run. In this case, caching it is not beneficial because an I/O is still required to fetch the missing KVs. Thus, we discard both the group and its associated LRU node(s). We should notice that KV Groups related to point queries will never be invalidated [4]. While it might be beneficial to proactively fetch and cache the missing KVs to complete the group, enabling future I/O savings, our experiments show that the current simple approach is already effective in practice.

Example. Figure 5 illustrates both cases. KV Groups are shown in blue. The KV Group {44} is created by a point query, whereas the others are created by the range query `scan(5, 40)`. In the upper half of the figure, two sorted runs are merged, and two KV Groups {10, 31} and {22} correspond to the same query. They are merged into a new group covering keys {10, 22, 31}, which fully covers the range [5, 40] in the new sorted run after compaction; therefore, the group remains cached. We insert its LRU node at the position formerly occupied by the KV Group {22} (which was closer to the tail), and then remove the obsolete red LRU node. In the lower half, the cached group {10, 31} does not need to be merged. However, it no longer fully covers [5, 40] in the new

[4]A special case is that the cached KV for a point query is an old version, while the compaction involves a new version. In this case, we will directly update the cached KV to maintain its validity.

sorted run, because keys 12, 21, and 27 were not cached before compaction. So, {10, 31} is no longer cached, and we remove it and its LRU node. The point KV Group {44} is preserved.

Cache Invalidation. Range Cache fully stores the results and is immune to compaction. Block Cache, in contrast, invalidates all cached blocks upon each compaction. Group Cache avoids cache invalidation for point queries. For range scans, cache invalidation may occur in Case 2. In our benchmarks, we find that cache invalidation usually becomes frequent when the LSM-Tree uses the aggressive Leveling compaction policy. However, we still find that our Group Cache outperforms Range Cache in all cases. An intuitive method to warm up the cache after compaction is to fetch the new KV Group if it is not too large. For example, if the uncovered portion of the new KV Group is less than 50%, we could fetch the uncovered part instead of evicting the covered part.

5 Theoretical Properties of Group Cache

We analyze Group Cache in two steps. As baselines, we use competitive analysis and the *static optimal* policy—fixing the cache to the most frequently accessed objects (or highest $\frac{frequency}{size}$ when sizes differ)—which is asymptotically optimal relative to Belady even under dependent requests [29]. First, we show that using KV Groups as the caching unit outperforms blocks or full-result caching by exploiting asymmetric I/O savings. Second, we prove that a simple size-aware policy (LRU-S) achieves a constant-factor miss rate relative to the static optimal, even with correlated accesses.

5.1 Static Optimal Performance Analysis

In this section, we show that the KV Group granularity can enhance the static optimal performance of the cache. We compare the static-optimal performance of three different cache units: block, full query result, and KV Group. We assume that the query workload follows a Zipf distribution with exponent $\alpha \in (0, 1)$. The j-th hottest query has probability $\lambda_j = c \cdot j^{-\alpha}, \quad j = 1, 2, \ldots$ where $c = 1/\sum_{j=1}^{\infty} j^{-\alpha}$ is the normalization constant. The cache supports multiple object size classes, indexed by k. Caching an object of class k consumes w_k space and saves v_k I/Os on each cache hit. For example, caching a data block for a point query requires B space to save 1 I/O. Fully caching the result of a range query requires K space to save L I/Os. The probability that a query accesses an object in size class k is π_k.

Under static optimality, the cache chooses to store the hottest objects in each size class. As shown in [13], the additional saved I/Os for caching one more object in any class should be equal. Assuming this additional saved I/Os is $\mu > 0$, we have:

$$\frac{\lambda_j v_k}{w_k} = \mu \quad \Longrightarrow \quad c \cdot m_k^{-\alpha} \frac{v_k}{w_k} = \mu \quad \Longrightarrow \quad m_k(\mu) = \left(\frac{c v_k}{\mu w_k}\right)^{1/\alpha}.$$

For each class k, the top $m_k(\mu)$ hottest objects will be cached. Assuming the total cache size is C, we have:

$$C = \sum_k \pi_k \sum_{j=1}^{m_k(\mu)} w_k = \sum_k \pi_k w_k m_k(\mu) = \left(\frac{c}{\mu}\right)^{1/\alpha} \sum_k \pi_k w_k^{1-1/\alpha} v_k^{1/\alpha}.$$

We denote S_g and C as follow:

$$S_g := \sum_k \pi_k \, w_k^{1-1/\alpha} \, v_k^{1/\alpha}, C = \left(\frac{c}{\mu}\right)^{1/\alpha} S_g.$$

Similarly, the total expected I/O savings (benefit) is the sum of access probabilities multiplied by corresponding savings:

$$U = \sum_k \pi_k v_k \sum_{j=1}^{m_k(\mu)} \lambda_j.$$

$$\sum_{j=1}^{m_k} \lambda_j = c \sum_{j=1}^{m_k} j^{-\alpha} \approx c \int_1^{m_k} j^{-\alpha} dj = \frac{c}{1-\alpha}(m_k^{1-\alpha} - 1) \approx \frac{c}{1-\alpha} m_k^{1-\alpha}.$$

$$U = \sum_k \pi_k v_k \frac{c}{1-\alpha} \left(\frac{c v_k}{\mu w_k}\right)^{(1-\alpha)/\alpha} = \frac{c}{1-\alpha} \left(\frac{c}{\mu}\right)^{1-1/\alpha} S_g.$$

$$U = \frac{c}{1-\alpha} S_g^{\alpha} C^{1-\alpha}.$$

This is a significant finding. As C, c, α are fixed by the cache size and workload, we only need to calculate the S_g for different cache granularities. We assume that each query is a point query with probability p and a range scan with probability $1-p$. The range scan length is K, and each scan probes L sorted runs. The size ratio of the LSM-tree is T. For ease of analysis, we assume that there are only two kinds of KV Groups for range scans: the upper group with $\frac{K}{T}$ space and $L-1$ I/O savings, and the lower group with $K(1-\frac{1}{T})$ space and one I/O saving. In practice, the upper group may be further divided into $L-1$ smaller groups. Both Group Cache and Range Cache use 1 unit of space to save 1 I/O for point queries. For Block Cache, one block of size B saves 1 I/O for a point query, and LB space is required to save L I/Os for a range query. Then:

$$S_{\text{Group}} = p + (1-p)\left[\left(\frac{K}{T}\right)^{1-1/\alpha} (L-1)^{1/\alpha} + \left(K\left(1-\frac{1}{T}\right)\right)^{1-1/\alpha}\right],$$

$$S_{\text{Range}} = p + (1-p)\,(K)^{1-1/\alpha} L^{1/\alpha},$$

$$S_{\text{Block}} = p \cdot B^{1-1/\alpha} + (1-p) \cdot (LB)^{1-1/\alpha} \cdot L^{1/\alpha}.$$

With 4 sorted runs (Leveling policy) and $T = 10, K = 20, B = 20, \alpha = 0.8, p = 0.2$, Group Cache is 29% better than range cache and 75% better than block cache. With Lazy Leveling policy, assuming there are 10 sorted runs, the advantage becomes 45% and 131%.

5.2 Optimality of LRU-S

We have shown that caching KV Groups promotes the static optimal performance of LSM-Tree's cache. We are also interested in how close a size-aware cache policy could approach the static optimal performance. In this section, we prove that in our setting, a simple LRU-S [46] algorithm is asymptotically a c-competitive cache algorithm and its cache miss rate is within a constant factor from that of the static optimal algorithm and Belady's algorithm. LRU-S is the simplest size-aware cache policy. When the system accesses an object with size s, the object will be admitted (uncached objects) or promoted (cached objects) to the head of the LRU chain with probability $\frac{1}{s}$. In [30], the authors proved that the LRU-S algorithm's cache miss rate is within a constant factor from that of the static optimal performance. As illustrated above, LRU-S is asymptotically c-competitive with Belady's optimal algorithm.

The proof sketch of the above theorem in [30] is to track the total size of objects that were promoted to the front of the cache more recently than the currently requested one. If this total size exceeds the cache capacity, a cache miss occurs. This total size is modeled as $S(t) = \sum s_i B_i(t)$, where $B_i(t) = 1$ if object i was promoted at least once in the last t time units. It is a Bernoulli variable with parameter $b_i = 1 - e^{-q_i p_i t}$, where q_i is the request rate of object i. In the proof, a

crucial step is to establish a concentration property for $S(t)$ (Lemma A3 in [30]). This concentration property shows that $S(t)$ is tightly clustered around its expected value, which simplifies the analysis and allows for a direct comparison between the miss rates of LRU-S and the static optimal policy.

However, the original derivation of this property relies on the assumption that all objects are *independently accessed*. Unfortunately, in our Group Cache setting, KV Groups are not always accessed independently. In particular, range scans typically access L different KV Groups simultaneously, introducing correlation among these accesses. We find that we could re-derive such a concentration property. We observe that each KV Group is correlated with at most $L-1$ other KV Groups. Combining such observation with Theorem 2.3 in [28], we could prove the concentration property for $S(t)$ in our setting. The other parts of the proof in [30] remain valid in our setting. **As a result, we conclude that LRU-S maintains its optimality in our Group Cache setting.**

6 Optimizations

6.1 Group Splitting

Under workload shifting, we hope to reuse previously cached results as much as possible. Because shifts are typically gradual [39], it is common that within a cached KV Group, some KVs cool down while others remain hot. In this scenario, evicting the entire KV Group becomes too coarse-grained. We introduce a Group Splitting mechanism to dynamically partition large KV Groups into smaller, more flexible sub-groups based on actual accesses. Group Splitting is triggered when a KV Group reaches the tail of the LRU chain and is about to be evicted. At this point, we examine the access counters for each KV within the group. KVs that have not been accessed since the last promotion will have a zero counter. These zero-counter positions serve as natural boundaries to partition the KV Group into several hot sub-groups, each containing consecutive KV pairs that have all been accessed at least once since the last promotion. KVs with zero counters are directly evicted.

For each sub-group, we define its access frequency f as the minimum counter value among its KVs. It represents the number of times the sub-group had the opportunity to be promoted in the cache since its last full promotion. Based on this frequency, we attempt to re-insert the sub-group into the cache for f times, where each attempt follows the underlying cache policy's probabilistic admission mechanism (such as the $\frac{1}{s}$ probability of LRU-S).

Once a sub-group is admitted (meaning the probabilistic admission test passes), we reset the counters of its KVs to zero. If the sub-group fails to be admitted (the probabilistic admission tests fail for all f attempts), we recursively apply the splitting process until all KVs are either successfully admitted or completely evicted, as shown in Algorithm 1. Specifically, we subtract f from the counters of all KVs in the sub-group, as we have already used their provided f chances. This normalization step prevents the recorded hotness of the current sub-group from affecting our evaluation of the hotness of the smaller sub-groups. After normalization, new zero-counter positions will emerge and trigger further partitioning. In the worst-case, each recursive step only reduces the group size by one, and the algorithm performs $O(s)$ work at each step. So the time complexity is $O(s^2)$. Since the maximum size of a KV Group is B, the worst-case time complexity of the Group Splitting mechanism is $O(B^2)$.

We show the process to split a KV Group consisting of keys k_1 to k_{10} with access counters $[5, 5, 5, 8, 9, 0, 9, 9, 6, 6]$. We observe that k_6 has a counter value of 0 after normalization, which serves as a natural split point. The group is partitioned into two consecutive sub-groups: $[k_1, k_5]$ and $[k_7, k_{10}]$, each containing only KVs with positive counters. For the first sub-group $[k_1, k_5]$, we define its frequency $f = \min([5, 5, 5, 8, 9]) = 5$ and try to admit this sub-group for 5 times. If it fails all 5 admission attempts, we subtract 5 from its counters, resulting in $[0, 0, 0, 3, 4]$. This reveals new zero-counter positions at k_1 to k_3, splitting the group again into two sub-groups: $[k_4, k_5]$ and

Algorithm 1: Group Splitting Algorithm

```
1 Procedure GroupSplit(G, policy)
2     subgroups = PartitionByZeroCounters(G)
3     foreach sg in subgroups do
4         f = min(kv.counter for kv in sg)
5         success = TryReinsert(sg, f, policy)
6         if not success then
7             foreach kv in sg do
8                 kv.counter = kv.counter - f
9             GroupSplit(sg, policy)
                // Recursive
```

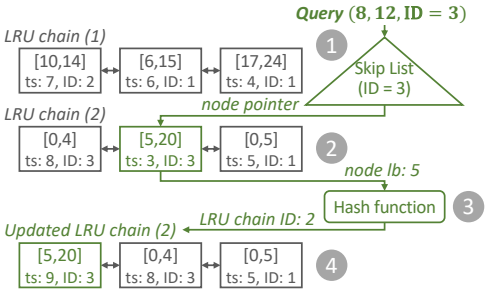


Fig. 6. The promotion of a partitioned LRU chain.

$[k_1, k_3]$. The latter is evicted. The former has $f = \min([3, 4]) = 3$ and is retried for admission 3 times. If it still fails, we subtract 3 again to get $[0, 1]$, and eventually retain only k_5. We try to admit k_5 again, and if it fails, k_5 will be evicted.

6.2 Concurrency Control via Partial Sharding

Managing concurrent access is a primary challenge for LRU caches [54, 55]. A conventional solution, adopted by RocksDB [6], partitions the cache into multiple independent shards. In this design, a data block is assigned to a specific shard via a hash of its ID, and each shard maintains its own independent storage structure (hash table) and LRU chain. While this method reduces contention, it introduces two limitations. First, the full hash partitioning is incompatible with Group Cache, which requires KV Groups to be sorted. Second, this mechanism results in shard-local eviction: the chosen eviction victim may not be the true globally least-recently used (LRU) item.

To address these limitations, we propose a Partial Sharding mechanism. Our key observation is that lock contention arises primarily from modifications to the LRU chain (which occur on every hit or admission), not from the storage structure (which is typically read-only during probes). Therefore, our solution partitions *only* the LRU chain into multiple concurrent lists while maintaining a single, global storage structure. This design preserves the global sorted order, while assigning LRU nodes to shards by hashing their KV Group's lower bound. Furthermore, to ensure a true global LRU eviction policy, we assign a timestamp to each node upon its last promotion. During eviction, we examine the tail node of every LRU shard and evict the node with the oldest timestamp. This technique achieves the same eviction accuracy as a single global LRU chain while retaining the concurrency benefits of sharding.

As shown in Figure 6, for a range query $scan(8, 12)$ in sorted run 3 (Step ①), Group Cache locates the target node covering the range $[5, 20]$ in the skip list. Then, using the LRU node pointer stored in the skip list node, the corresponding LRU node can be retrieved. Next, the hash of the node's lower bound (5) is used to find the head of the corresponding LRU chain (Steps ②–③). After acquiring the chain's lock, the node is promoted to the head and its timestamp is updated (Step ④). This design enables efficient updates by locking only one LRU shard per promotion.

7 Evaluation

In this section, we present a comprehensive evaluation of Group Cache. Since Group Cache is designed to optimize read performance, we first run read-only microbenchmarks to examine its behavior across workloads and configurations (Section 7.1). We then evaluate mixed read–write workloads to study potential trade-offs and the effects of different compaction policies (Section 7.2).

We further assess robustness and scalability (Section 7.3). Finally, we evaluate Group Cache on six real workloads to demonstrate its value in production environments (Section 7.4).

Baselines: We compare our Group Cache with two baselines: Block Cache (RocksDB's default cache) and Range Cache [50]. We implement both Group Cache and Range Cache in Dostoevsky [19], an optimized version of RocksDB with a better read-write trade-off [19, 34, 49]. The experiments for Block Cache with asynchronous I/O are conducted on traditional RocksDB.

Setup: Our machine equipped with an AMD EPYC 7742 64-core processor, 512MB of L3 cache, and a 3TB D7-P5620 NVMe SSD. The memory write buffer is set to 64 MB. A SNARF [47] range filter is built for every file with a default memory allocation of 14 bpk. Our LSM-Tree contains 4 levels with 200M KVs, using 8B integer keys and 256B values. We enable direct I/O for both reads and writes. The size ratio is set to 10. By default, the cache size is configured as 5% of the total LSM-Tree size (approximately 2.5 *GB*). The cache is only used to store KVs[5]. Our default compaction policy is Lazy Leveling, and the default cache policy is LRU-S. In all multi-thread experiments, we set the number of shards to 32 as RocksDB default.

Metrics: Except for Table 1, all throughputs are measured for overall workloads, and latencies consider both point and range queries, as the cache designs have negligible impact on writes.

7.1 Experiments on Read-Only Workloads

By default, we run experiments under a Zipfian workload with a skew factor of 0.8. Range scans account for 20% of the queries, and the remaining 80% are point queries. Each range scan retrieves between 10 and 100 key-value pairs, following the characters of real workloads [9, 11, 15]. Given a size ratio of 10, approximately 90% of the target KVs are located in the last level. Each experiment is preceded by 10M warm-up queries drawn from the same distribution. Our default dataset is the books dataset in SOSD[37]. In Figure 7 and Figure 8, we show the latency and cache hit rate of the three caches. We also use different cache policies (AdaptSize [10] and Cliffhanger [13]) for Group Cache and find that they cannot improve the performance over LRU-S (Figure 7).

Impact of Range Scan Percentages. We vary the range scan percentage from 0 to 0.8, with the remaining workload consisting of positive point queries. When the workload contains only point queries, Range Cache and Group Cache have similar performance as they both act as a normal KV Cache. As the scan percentage increases, the latency of all policies increases. Among all methods, the three Group Cache variants consistently outperform both Block Cache and Range Cache. Besides, these three size-aware cache policies show similar performance, while the simplest LRU-S is slightly better. This supports our argument that improving the static optimal performance is more important than designing sophisticated cache policies. Thus, we report only LRU-S in the remaining experiments. When the range scan percentage is relatively low, Group Cache offers a 2× to 3× advantage. This advantage becomes less significant at higher scan percentages. The reason is that as the scan percentage increases, the number of I/Os exceeds what can be effectively cached, even by Group Cache. For example, at a scan percentage of 0.8, although Group Cache has a 1.5× higher cache hit rate compared to Block Cache, the latency improvement is small.

Impact of Zipf Factors. In Figure 7 and Figure 8, we also vary the Zipf factor between 0 and 1.2. When the Zipf factor is 0, the workload is uniform and the Block Cache has the best performance due to its least metadata overhead. Group Cache consistently outperforms Block Cache and Range Cache when the Zipf factor is larger than 0. The advantage is maximized when the Zipf factor is 0.8. As the workload becomes more skewed, the performance of different cache policies tends to converge because all caches have a high cache hit rate and can fit most of the hot KVs. In contrast,

[5] `cache_index_and_filter_blocks` is set to false as RocksDB default.

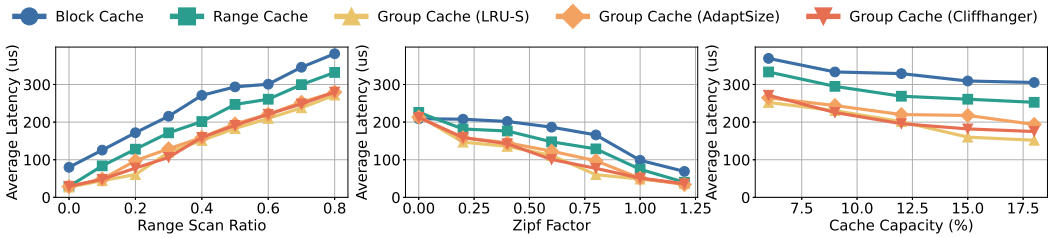


Fig. 7. Average latency under different range scan ratios, Zipf Factors, and Cache Capacities – The experiments are conducted on the books dataset in SOSD. The size ratio of the LSM-Tree is 10.

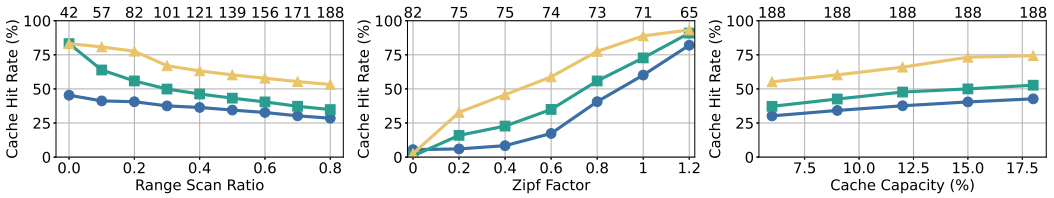


Fig. 8. The cache hit rate under different range scan ratios, Zipf factors, and cache capacities. –The total number of required I/Os (M) for each experiment, when no cache is available, is shown at the top of the figures. For any data point, the number of saved I/Os is calculated by multiplying the cache hit rate by the total number of I/Os.

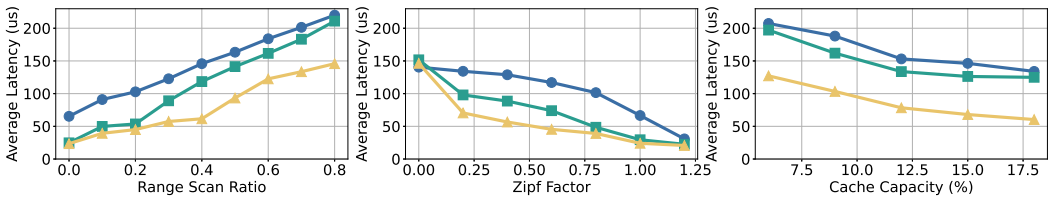


Fig. 9. Performance on tall LSM-Tree – The size ratio is set to 3.

when the distribution is flatter, only Group Cache maintains strong performance by leveraging the underlying access asymmetry.

Impact of Cache Sizes. We evaluate the impact of cache size. The cache size for all cache policies is varied from 6% to 18% of the total LSM-Tree size. In previous experiments, we observed that with 20% range scans, the cache hit rates of Group Caches were already very high at 5% cache size. To better show the effect of cache size, we configure the workload to contain 80% range scans and 20% positive point queries. These scans generate a large number of I/Os and are highly sensitive to cache size. As shown in Figure 7 and Figure 8, we find that the advantage of Group Caches increases as cache size increases. This indicates that Group Caches can better utilize the available cache space by prioritizing high-utility, small KV Groups. A common concern with Group Cache is its higher metadata overhead compared to block-based caching. For instance, each KV Group requires a dedicated LRU node. However, our results show that to achieve comparable performance, Range Cache requires more than 2× the space, and Block Cache requires over 4×. Although Group Cache introduces additional metadata, its impact on space efficiency is relatively small.

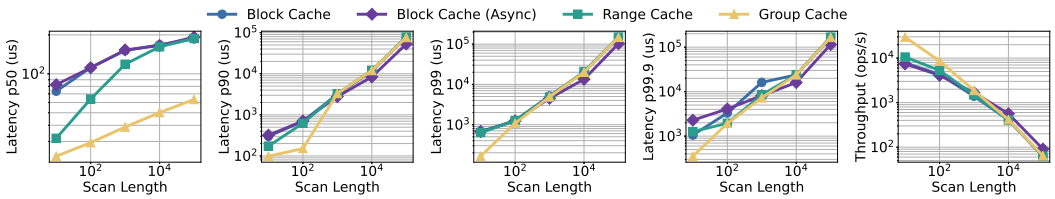


Fig. 10. The impact of range scan length.

Impact of Tall LSM-Tree. The performance of the Group Cache depends on the skewness of the KV Group size distribution. When the LSM-Tree is tall with a small size ratio, the distribution will be less skewed. So in Figure 9, we test the performance of Group Cache for an LSM-Tree with size ratio 3, which theoretically provides a better read performance compared with using a size ratio of 10 [59]. As a result, the advantage of Group Cache over Range Cache becomes less significant. Moreover, the asymmetry is also less significant. However, with a higher range scan ratio, the advantage of Group Cache becomes more significant. The reduced asymmetry could still save a lot of I/Os with increasing query overhead.

Impact of Scan Length. In real workloads [9, 11], range scans may involve more than 100 keys. In Figure 10, we evaluate the impact of scan length. The workload comprises 80% point queries and 20% range scans. The x-axis reports the maximum scan length, and for each range query, the scan length is drawn uniformly from 10 to this upper bound. We use the OSM dataset from SOSD, which degrades the accuracy of range filters [12, 23, 47, 52] and thus approximates a worst-case scenario for caching. RocksDB has recently optimized range-scan performance with asynchronous I/O [1] to utilize SSD bandwidth better. Our current Group Cache and Range Cache do not yet support asynchronous I/O. In this experiment, we therefore also compare Block Cache with asynchronous I/O against Group Cache. The experiments of Block Cache are conducted on the latest RocksDB.

As shown in Figure 10, Group Cache consistently achieves lower P50 latency than all baselines. P50 primarily reflects point queries and short scans, for which Group Cache offers better cache efficiency and prevents their KV Groups from being evicted by long scans. The tail latencies of all methods converge as scan length increases, because long scans touch many data blocks filled entirely with target keys and create many large KV Groups (most contain B KVs). Consequently, each cache handles these long scans similarly. With asynchronous I/O, Block Cache performs worse on short scans due to over-prefetching [1]; however, once the scan length exceeds 1000, Block Cache with asynchronous I/O achieves the best performance. Accordingly, in Section 7.4, we enable asynchronous I/O for Block Cache when the scan length exceeds 1000.

7.2 Experiments on Read-Write Workloads

In this section, we show Group Cache's performance on mixed read-write workloads with different compaction policies. We vary the write ratio from 0.2 to 1.0. 50% of the write operations are insert, and the others are update operations. 20% of the query operations are range scans, and the others are point queries. The update and query operations follow a Zipf distribution with a 0.8 Zipf Factor. Other settings remain the same as in our previous experiments. We find that under each write ratio and compaction policy, our Group Cache always outperforms other baselines.

Impact of Write Operations. On the write-only workload, all three caches show similar performance on three different compaction policies, as the cache designs do not incur significant write overhead. Such an argument is also supported by Table 1. We measure the p99, p100, and average compaction times of three caches on LSM-Trees with different size ratios. As shown in Table 1, the

Table 1. Compaction CPU time (*s*). –We measure the CPU time spent on compaction (`rocksdb.compaction.times.cpu_micros`) under a 50/50 read–write workload across different size ratios. Under each size ratio, all caches incur similar compaction overhead.

	Block Cache			Range Cache			Group Cache		
T	P99	P100	Avg	P99	P100	Avg	P99	P100	Avg
3	15.83	142.98	1.16	16.11	143.41	1.14	16.06	142.23	1.15
6	39.61	175.73	2.16	39.18	174.65	2.12	39.52	178.92	2.11
9	43.64	141.44	2.79	42.65	142.12	2.84	42.84	144.84	2.80
12	75.62	94.14	3.68	77.85	96.09	3.72	77.45	97.08	3.72

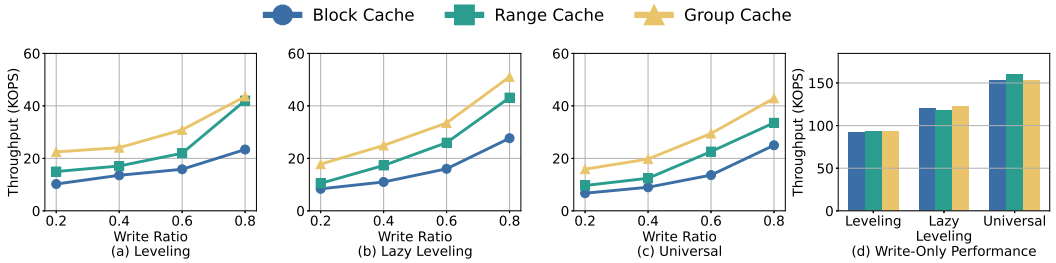


Fig. 11. Performance on mixed read-write workloads with different compaction policies.

compaction times across caches are similar. This is because maintaining a small cache incurs little overhead and does not incur additional I/O. As the write ratio increases from 0.2 to 0.8, all caches have increasing throughputs, because the write speed is much faster than the query speed. Our Group Cache consistently shows the best performance due to its better query performance. We find that Range Cache is gradually approaching Group Cache's performance with a higher write ratio. The reasoning is: (1) If a key range is cached in the Range Cache, newly inserted (updated) KVs in this range will be directly inserted into the Range Cache [50] when it is inserted into the mem-table. On the other hand, Group Cache will not bring it into the cache before it has been queried. (2) With more writes and updates, there will be more compactions and cache invalidation, which degrades the performance of Group Cache. As the Block Cache has a more serious cache invalidation problem, the performance gap between Block Cache and Range Cache becomes larger under a higher write ratio.

Impact of Compaction Policies. We evaluate Group Cache under three representative compaction policies: *Leveling*, *Lazy Leveling*, and *Universal*. We find that our Group Cache provides the best performance on all compaction policies, and the Lazy Leveling policy could provide a better overall performance on the mixed workload, which is consistent with previous works [19, 34, 49]. *An aggressive compaction policy, like the Leveling policy, not only reduces the write speed, but also increases the cache invalidation frequency.* The advantage of Group Cache under the Universal policy is less significant as it has more equal-sized large sorted runs, which reduces the skewness of KV Group size distribution.

7.3 Experiments on Robustness and Scalability

Impact of Data Block Size. The data block size impacts the read performance in several ways. It impacts the latency of each I/O, the granularity of the Block Cache, and also the KV Group size upper bound. In Figure 12(a), we vary the block size from 4KB to 256KB. The query latencies of all

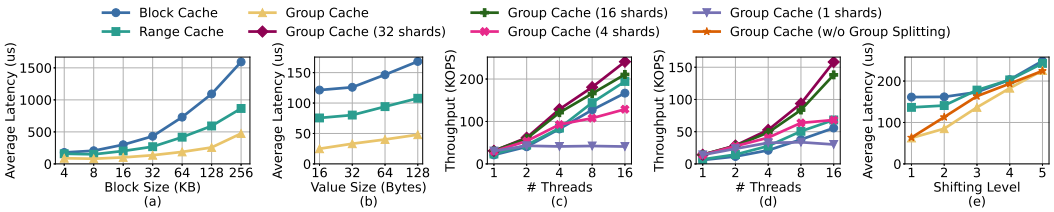


Fig. 12. Experiments on Robustness and Scalability. – (a)/(b) show the impact of data block size/value size. (c)/(d) show the scalability on YCSB-A/YCSB-E. (e) shows the impact of workload shifting. Assuming all keys are sorted in array KEY, Shifting level i indicates that in the new workload, the hotness of $KEY[j]$ is equal to the hotness of $KEY[j - 10 \times (i - 1)]$ in previous workload.

caches increase as each I/O becomes more expensive. The advantage of Group Cache and Range Cache over Block Cache becomes more significant with larger data blocks (more unnecessary data).

Impact of Value Size. In Figure 12(b), we vary the size of the value. We insert more KVs compared with previous experiments so that the total data size remains 50GB and the shape of the LSM-Tree remains the same. With a smaller value, the Range Cache and Group Cache could cache more KVs, and the Block Cache could store the same number of data blocks. Although the metadata overhead for Range Cache and Group Cache becomes heavier with a smaller value, the unnecessary KVs in a data block also increase. So we find that Range Cache and Group Cache could outperform Block Cache with all value sizes. Besides, the advantage of Group Cache over Range Cache also becomes more significant under smaller value sizes, as the Group Cache uses KV Groups to amortize the increasing metadata overhead.

Multi-Threaded Performance. We evaluate the throughput of the caching algorithms under dynamic read-write workloads using the YCSB benchmark, as shown in Figure 12(c) and (d). We use the standard YCSB workloads A and E. Since Group Cache primarily targets range scans, we modify Workload A to include 50% updates, 40% point queries, and 10% range scans. Workload E consists of 95% range scans and 5% insertions. We run experiments with the number of client threads varying from 1 to 16. Compactions are triggered when L0 accumulates 10 sorted runs, and writes are stalled if L0 exceeds 20 runs. We also optimize Range Cache by integrating our partial sharding mechanism to reduce contention on the LRU chain. We show the impact of our partial sharding mechanism by varying the number of shards of the Group Cache from 1 to 32.

In Workload A, the performance gap between Group Cache and the baselines narrows, as frequent updates reduce the caching effectiveness of Group Cache as discussed above. Nevertheless, Group Cache still maintains a consistent throughput advantage. In contrast, Workload E demonstrates the full strength of Group Cache and improves throughput by more than 3× compared to the baselines. In both workloads, Group Cache's scalability comes from the partial sharding methods. With only 1 shard, the performance of Group Cache declined to only 20%. Besides, we have not found significant victim selection overhead when there are multiple shards, as the number of shards is small.

Performance under Workload Shifting. For a cache system, a crucial property is whether it can quickly adapt to workload shifting. We create a shifting workload by shifting the initial hot key set several positions. In Figure 12(e), we firstly warm the cache with 10M queries. Then, we conduct queries with workload shifting. The test workload only contains 1M queries to avoid the cache being totally warmed again. We also test the impact of group splitting, which is designed to reuse the cached data as much as possible. As the shifting level increases, all caches have increasing query latency. The impact of group splitting is maximized when the shifting level is moderate. When the shifting level is high, the group splitting can no longer be beneficial, as most of the KVs

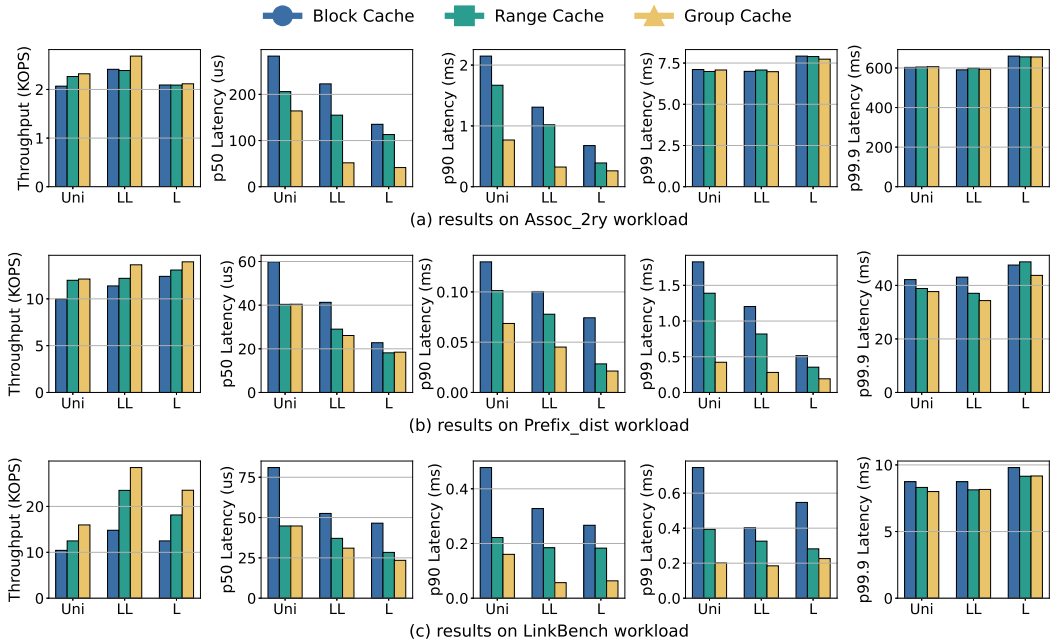


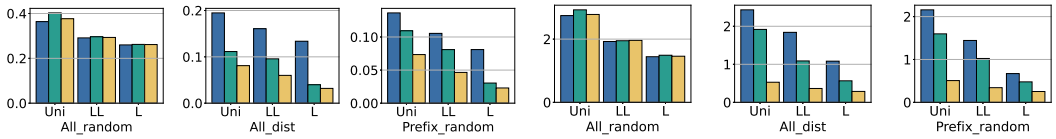
Fig. 13. Results on three real workloads. –Uni = Universal compaction, LL = Lazy Leveling, L = Leveling.

in cached KV Groups become cold. The Group Cache is always better than Range Cache and Block Cache by quickly recognizing those high-utility KV Groups.

7.4 Experiments on Real Workloads

We evaluate our caching schemes on six real workloads: Assoc_2ry [11], four default workloads in MixGraph [11], and LinkBench [9]. We evaluate the caches across multiple workloads under three compaction policies—Universal (Uni), Lazy Leveling (LL), and Leveling (L). Following our earlier findings, we enable asynchronous I/O in the Block Cache for scans longer than 1,000 keys. Because keys in these workloads are not integers, we build a SuRF filter [60] for each sorted run to avoid unnecessary I/Os during scans. The data block size is set to 8 KB. All operations are conducted with 8 threads. The other parameters use the default configuration. We first load 100 M KVs, followed by 10 M warm-up operations, and then measure performance over 40 M operations.

We summarize our key findings as follows: (1) Except for a completely random workload, Group Cache consistently outperforms all baselines, as it accelerates short scans and prevents hot point queries or short scans from being affected by long scans, making Group Cache valuable for both scan-heavy and get-heavy workloads. (2) In some workloads (Assoc_2ry and Prefix_dist), the throughput is dominated by long scans, even though they are rare in number. As discussed earlier, all caches handle these long scans similarly, resulting in similar throughput. In fact, long scan's throughput can only be improved by using more memory or higher SSD bandwidth. However, Group Cache shows significant advantages in latency metrics. (3) When the workload is both scan-heavy and write-heavy (Assoc_2ry and LinkBench), the Lazy Leveling policy achieves the best performance due to its superior read–write trade-off and reduced bandwidth contention between queries and compactions. Under the Lazy Leveling policy, the advantage of Group Cache is usually maximized due to the highest Range-access Asymmetry.



(a) P90 latencies (b) P99 latencies

Fig. 14. P90 and P99 latencies (ms) on MixGraph benchmarks.

Assoc_2ry. In Facebook's UDB schema, Assoc_2ry is the secondary index for associations. We tune the parameters of the MixGraph benchmark to match the characteristics reported in [11]. The workload consists of 56% puts and 44% scans. About 60% of scans return one key, and 90% return fewer than 100 keys. A scan involves up to 100,000 keys. The key and value sizes are 64 B and 16 B.

As shown in Figure 13(a), Group Cache delivers higher throughput (about 15%) under the Universal and Lazy Leveling compaction policies. Lazy Leveling achieves the best overall performance because it offers a more favorable read–write trade-off. Under the Leveling policy, p99 and p99.9 latencies exceed those of Lazy Leveling and Universal because Assoc_2ry's tail is dominated by very long scans whose required I/Os are almost unaffected by data layout, while Leveling's aggressive compactions compete with queries for I/O bandwidth. Group Cache achieves substantially lower p50 and p90 latencies, which are dominated by short scans. This advantage is most pronounced with Lazy Leveling, which yields the most skewed KV-group size distribution, as discussed above. In contrast, tail latencies (p99 and p99.9) are unaffected by the cache design, as the scan length can reach up to 100,000 keys. We also find that when queries run with 8 threads concurrently with writes and background compactions, asynchronous I/O no longer allows Block Cache to outperform Group Cache, as the storage bandwidth is nearly saturated and async I/O merely parallelizes I/Os rather than reducing the total number of I/Os.

Prefix_dist. The Prefix_dist workload is one of the four default workloads provided by the MixGraph benchmark [11] and best approximates real workloads. The MixGraph benchmark uses two sets of parameters—key_dist and keyrange_dist—to generate distributions for keys and prefixes. Prefix_dist combines hot prefixes with hot keys within those prefixes. We generate it using the provided MixGraph benchmark with the original parameters, yielding 14% puts, 83% gets, and 3% scans. Following the original settings, we set the key and value sizes to 48 B and 43 B, respectively.

As shown in Figure 13(b), Group Cache achieves higher throughput (about 10%) even when the workload is get-heavy. Because the write ratio is small, the Leveling compaction policy delivers the best throughput and the lowest p50, p90, and p99 latencies. The p50 latency is dominated by hot point queries, so Range Cache and Group Cache perform similarly; both outperform Block Cache due to finer-grained cache units. At p90 latency, which is dominated by warm point queries and short hot scans, Group Cache clearly outperforms Range Cache. Although the workload contains only 3% scans, Group Cache prevents warm keys and small hot KV Groups from being evicted by large KV Groups. The p99 latency corresponds to short range scans, where Group Cache shows the greatest advantage. The p99.9 latency also improves with Group Cache. Since the workload contains only 3% scans, the p99.9 latency corresponds to medium-length queries, which can also benefit from Group Cache. The experiment on Prefix_dist highlights the value of Group Cache in get-heavy workloads.

To further investigate the impact of query distributions in get-heavy workloads, we also evaluate Group Cache on the other three MixGraph workloads: All_random, All_dist, and Prefix_random. In All_random, both key_dist and keyrange_dist parameters are set to zero, so all queries have uniform hotness. In All_dist, keyrange_dist parameters are set to zero, resulting in hot queries that

are uniformly scattered across the entire key space. In Prefix_random, key_dist parameters are set to zero, resulting in hot ranges where all queries within each hot range have the same hotness. We show the p90 and p99 latencies of these three workloads in Figure 14a and Figure 14b. We find that in All_random, all caches perform similarly, and Block Cache is slightly better due to lower metadata overhead. On the skewed workloads (All_dist and Prefix_random), each cache exhibits trends similar to Prefix_dist. The latencies of Block Cache increase due to the lack of spatial locality compared with Prefix_dist, whereas Range Cache and Group Cache are more robust since their performance does not rely on spatial locality.

LinkBench. LinkBench models Facebook's social-graph storage. The workload consists of 31% puts (including `assoc_insert`, `assoc_update`, `assoc_delete`, `obj_insert`, `obj_update`, and `obj_delete`), 18.3% gets (including `obj_get`, `assoc_count`, and `assoc_multiget`), and 50.7% scans (i.e., `getLinkList`). For scan operations, about 70% of queries return at most one key, 90% return fewer than 10 keys, and 95% return fewer than 100 keys. The maximum scan limit is 10,000 keys. The key and value sizes are 25 B and 128 B, respectively.

As shown in Figure 13(c), Group Cache exhibits a larger throughput advantage than in the previous workloads, driven by the higher fraction of short scans in LinkBench. Range Cache and Group Cache provide similar p50 latency because most range scans return at most one key, and this fraction is higher than in Assoc_2ry. The advantage of Group Cache becomes significant at p90 and p99 for scans that involve fewer than 100 keys. The p99 latencies for Block Cache and Group Cache increase under the Leveling policy. As the LinkBench workload contains more writes, the cache invalidation problem of both Block Cache and Group Cache becomes more pronounced.

8 Conclusion

This paper tackles inefficient range scans in LSM-Trees by exploiting a range access asymmetry in their structure. We introduce Group Cache, a novel system that uses variable-sized KV Groups as its caching unit. By employing a size-aware policy to prioritize small, high-utility groups, Group Cache maximizes I/O savings for a given memory budget. Our evaluation shows it delivers up to 3 $\times$ faster query performance compared to traditional block and range caches. This allows for more write-efficient LSM-Tree configurations without compromising read speed.

Acknowledgement

This work was supported (in part) by the Shanghai Qi Zhi Institute.

References

[1] [n. d.]. Asynchronous IO in RocksDB. https://github.com/facebook/rocksdb/wiki/Asynchronous-IO.
[2] 2012. Sanjay Ghemawat and Jeff Dean. LevelDB. https://github.com/google/leveldb.
[3] 2024. Apache. Cassandra. http://cassandra.apache.org.
[4] 2024. Apache. HBase. http://hbase.apache.org/.
[5] 2024. CockroachDB. https://github.com/cockroachdb/cockroach..
[6] 2024. Facebook. RocksDB. https://github.com/facebook/rocksdb..
[7] 2024. LinkedIn. Voldemort. http://www.project-voldemort.com.
[8] 2024. WiredTiger. https://github.com/wiredtiger/wiredtiger..
[9] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark D. Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *ACM SIGMOD Conference*. https://api.semanticscholar.org/CorpusID:11759711
[10] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. 2017. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 483–498. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/berger

[11] Zhichao Cao, Siying Dong, Sagar Vemuri, and David Hung-Chang Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *USENIX Conference on File and Storage Technologies*. https://api.semanticscholar.org/CorpusID:211137004

[12] Guanduo Chen, Zhenying He, Meng Li, and Siqiang Luo. 2024. Oasis: An Optimal Disjoint Segmented Learned Range Filter. *Proc. VLDB Endow.* 17, 8 (may 2024), 1911–1924. https://doi.org/10.14778/3659437.3659447

[13] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: scaling performance cliffs in web memory caches. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI'16)*. USENIX Association, USA, 379–392.

[14] Alex Conway, Martín Farach-Colton, and Rob Johnson. 2023. SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy. *Proc. ACM Manag. Data* 1, 1, Article 46 (may 2023), 27 pages. https://doi.org/10.1145/3588726

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[16] Marco Costa, Paolo Ferragina, and Giorgio Vinciguerra. 2024. Grafite: Taming Adversarial Queries with Optimal Range Filters. *Proc. ACM Manag. Data* 2, 1, Article 3 (mar 2024), 23 pages. https://doi.org/10.1145/3639258

[17] 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* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3035918.3064054

[18] Niv Dayan, Philippe Bonnet, and Stratos Idreos. 2016. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. *Proceedings of the 2016 International Conference on Management of Data* (2016).

[19] 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. *Proceedings of the 2018 International Conference on Management of Data* (2018).

[20] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. *Proceedings of the 2019 International Conference on Management of Data* (2019).

[21] 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* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 365–378. https://doi.org/10.1145/3448016.3457273

[22] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017).

[23] Navid Eslami and Niv Dayan. 2024. Memento Filter: A Fast, Dynamic, and Robust Range Filter. *Proc. ACM Manag. Data* 2, 6, Article 244 (Dec. 2024), 27 pages. https://doi.org/10.1145/3698820

[24] 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* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 32, 14 pages. https://doi.org/10.1145/2741948.2741973

[25] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index. *Proc. VLDB Endow.* 17 (2024), 3442–3455. https://api.semanticscholar.org/CorpusID:272280401

[26] 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* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 651–665. https://doi.org/10.1145/3299869.3314041

[27] 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. *Proceedings of the 2019 International Conference on Management of Data* (2019).

[28] Svante Janson. 2004. Large deviations for sums of partly dependent random variables. *Random Struct. Algorithms* 24, 3 (May 2004), 234–248.

[29] Predrag Jelenkovic and Ana Radovanovic. 2009. Asymptotic Optimality of the Static Frequency Caching in the Presence of Correlated Requests. *Operations Research Letters* 37 (2009), 307–311.

[30] Predrag R. Jelenković and Ana Radovanović. 2004. Optimizing LRU Caching for Variable Document Sizes. *Comb. Probab. Comput.* 13, 4–5 (July 2004), 627–643. https://doi.org/10.1017/S096354830400625X

[31] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 821–837. https://www.usenix.org/conference/atc21/presentation/kassa

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

[33] Haridimos Kondylakis, Niv Dayan, Konstantinos Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. *Proc. VLDB Endow.* 11 (2018), 677–690. https://api.semanticscholar.org/CorpusID:3569962

[34] Junfeng Liu, Fan Wang, Dingheng Mo, and Siqiang Luo. 2024. Structural Designs Meet Optimality: Exploring Optimized LSM-tree Structures in a Colossal Configuration Space. *Proc. ACM Manag. Data* 2, 3, Article 175 (may 2024), 26 pages. https://doi.org/10.1145/3654978

[35] 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2071–2086. https://doi.org/10.1145/3318464.3389731

[36] Siqiang Luo, Ben Kao, Guoliang Li, Jiafeng Hu, Reynold Cheng, and Yudian Zheng. 2018. TOAIN: A Throughput Optimizing Adaptive Index for Answering Dynamic kNN Queries on Road Networks. *Proc. VLDB Endow.* 11 (2018), 594–606. https://api.semanticscholar.org/CorpusID:5595327

[37] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. 2020. Benchmarking learned indexes. *Proceedings of the VLDB Endowment* 14 (2020), 1 – 13.

[38] Prashanth Menon, Thamir M. Qadah, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2022. LogStore: A Workload-Aware, Adaptable Key-Value Store on Hybrid Storage Systems. *IEEE Transactions on Knowledge and Data Engineering* 34, 8 (2022), 3867–3882. https://doi.org/10.1109/TKDE.2020.3027191

[39] Dingheng Mo, Fanchao Chen, Siqiang Luo, and Caihua Shan. 2023. Learning to Optimize LSM-trees: Towards A Reinforcement Learning based Key-Value Store for Dynamic Workloads. *Proc. ACM Manag. Data* 1, 3, Article 213 (nov 2023), 25 pages. https://doi.org/10.1145/3617333

[40] Jiansheng Qiu, Fangzhou Yuan, Mingyu Gao, and Huanchen Zhang. 2024. HotRAP: Hot Record Retention and Promotion for LSM-trees with Tiered Storage. arXiv:2402.02070 [cs.DB] https://arxiv.org/abs/2402.02070

[41] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J. Freedman. 2023. Efficient Compactions between Storage Tiers with PrismDB. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 179–193. https://doi.org/10.1145/3582016.3582052

[42] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. 2018. mLSM: Making Authenticated Storage Faster in Ethereum. In *USENIX Workshop on Hot Topics in Storage and File Systems*. https://api.semanticscholar.org/CorpusID:46996220

[43] Sean C. Rhea, Eric Wang, Edmund Wong, Ethan Atkins, and Nat Storer. 2017. LittleTable: A Time-Series Database and Its Uses. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017). https://api.semanticscholar.org/CorpusID:28501514

[44] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proc. VLDB Endow.* 14 (2021), 2216–2229.

[45] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a general purpose log structured merge tree. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012). https://api.semanticscholar.org/CorpusID:207194816

[46] David Starobinski and David Tse. 2001. Probabilistic methods for web caching. *Perform. Eval.* 46, 2–3 (Oct. 2001), 125–137. https://doi.org/10.1016/S0166-5316(01)00045-1

[47] Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. 2022. SNARF: A Learning-Enhanced Range Filter. *Proc. VLDB Endow.* 15, 8 (jun 2022), 1632–1644. https://doi.org/10.14778/3529337.3529347

[48] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. 2024. GRF: A Global Range Filter for LSM-Trees with Shape Encoding. *Proc. ACM Manag. Data* 2, 3, Article 141 (may 2024), 27 pages. https://doi.org/10.1145/3654944

[49] Hengrui Wang, Jiansheng Qiu, Fangzhou Yuan, and Huanchen Zhang. 2025. Rethinking The Compaction Policies in LSM-trees. *Proc. ACM Manag. Data* 3, 3, Article 207 (June 2025), 26 pages. https://doi.org/10.1145/3725344

[50] 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 488–500. https://doi.org/10.1109/ICDE60146.2024.00044

[51] Zhiqi Wang and Zili Shao. 2023. MirrorKV: An Efficient Key-Value Store on Hybrid Cloud Storage with Balanced Performance of Compaction and Querying. *Proc. ACM Manag. Data* 1, 4, Article 249 (Dec. 2023), 27 pages. https://doi.org/10.1145/3626736

[52] Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Huanchen Zhang, and Bin Cui. 2023. REncoder: A Space-Time Efficient Range Filter with Local Encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2036–2049. https://doi.org/10.1109/ICDE55515.2023.00158

[53] 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

[54] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K. V. Rashmi. 2023. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 70–79. https://doi.org/10.1145/3593856.3595887

[55] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 130–149. https://doi.org/10.1145/3600006.3613147

[56] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rong yao Chen, Jianying Wang, and Gui Huang. 2020. Leaper. *Proceedings of the VLDB Endowment* 13 (2020), 1976 – 1989. https://api.semanticscholar.org/CorpusID:221082134

[57] Hobin Yoon, Juncheng Yang, Sveinn Fannar Kristjansson, Steinn E. Sigurdarson, Ymir Vigfusson, and Ada Gavrilovska. 2018. Mutant: Balancing Storage Cost and Latency in LSM-Tree Data Stores. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 162–173. https://doi.org/10.1145/3267809.3267846

[58] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-In-Place Key-Value Store for Modern Storage. *Proc. VLDB Endow.* 16 (2022), 99–112. https://api.semanticscholar.org/CorpusID:252924530

[59] Weiping Yu, Siqiang Luo, Zihao Yu, and Gao Cong. 2024. CAMAL: Optimizing LSM-trees via Active Learning. *Proc. ACM Manag. Data* 2, 4, Article 202 (Sept. 2024), 26 pages. https://doi.org/10.1145/3677138

[60] 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* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 323–336. https://doi.org/10.1145/3183713.3196931

[61] Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, and Jianling Sun. 2022. SA-LSM : Optimize Data Layout for LSM-tree Based Storage using Survival Analysis. *Proc. VLDB Endow.* 15 (2022), 2161–2174.

[62] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. 2021. REMIX: Efficient Range Query for LSM-trees. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 51–64. https://www.usenix.org/conference/fast21/presentation/zhong

[63] Zichen Zhu, Yanpeng Wei, Ju Hyoung Mun, and Manos Athanassoulis. 2025. Mnemosyne: Dynamic Workload-Aware BF Tuning via Accurate Statistics in LSM trees. *Proceedings of the ACM on Management of Data (PACMMOD)* 3, 3 (2025). https://doi.org/10.1145/3725327

A Detailed Proofs for Section 5.2

In Section 5.2 we outlined a high-level proof sketch showing that, in the LSM-Tree setting with Group Cache, the simple size-aware policy LRU-S is asymptotically c-competitive and achieves a miss rate within a constant factor of both the static optimal policy and Belady's algorithm. In this appendix we provide the complete details. We begin by recalling the standard setup used in [30, 46], adapt it to the Group Cache model in which range scans introduce limited dependencies among KV Groups, and then establish the required concentration bound for

$$S(t) = \sum_{i \in A} s_i B_i(t),$$

the total size of objects (KV Groups) promoted in the last t time units. Throughout, $B_i(t) \in \{0, 1\}$ indicates whether object i was promoted at least once in the last t, with parameter $b_i(t) = \mathbb{P}[B_i(t) = 1]$; under a Poisson request process with rate q_i and LRU-S promotion probability p_i (for an object of size s_i, typically $p_i = 1/s_i$), one has $b_i(t) = 1 - e^{-q_i p_i t}$. Let $s_{\max} = \max_i s_i$.

A.1 Dependency graph and a concentration tool

In Group Cache, a range scan touches at most L distinct KV Groups across the involved runs, so each $B_i(t)$ can be dependent on at most $L - 1$ other indicators. Representing the family $\{B_i(t)\}_{i \in A}$ by a dependency graph, the maximum degree is $\Delta \le L - 1$. We will use the following concentration inequality (Theorem 2.3 in [28]), stated here for completeness.

Theorem A.1 (Janson (2004)). *Let $\{Y_\alpha\}_{\alpha \in A}$ be random variables with finite second moments, indexed by a finite or countable set A. Suppose the dependency graph on A has maximum degree Δ, and there exists $b > 0$ such that $Y_\alpha - \mathbb{E}Y_\alpha \le b$ for all $\alpha \in A$. Define*

$$Z = \sum_{\alpha \in A} Y_\alpha, \quad \mu = \mathbb{E}Z, \quad V = \sum_{\alpha \in A} \mathrm{Var}(Y_\alpha), \quad \varphi(x) = (1 + x)\ln(1 + x) - x.$$

Then for every $t \ge 0$,

$$\mathbb{P}(Z \ge \mu + t) \le \exp\left(-\frac{V}{b^2(\Delta + 1)}\, \varphi\left(\frac{4bt}{5V}\right)\right).$$

Moreover, since $\varphi(x) \ge \dfrac{x^2}{2(1 + x/3)}$ for $x \ge 0$,

$$\mathbb{P}(Z \ge \mu + t) \le \exp\left(-\frac{t^2}{2(\Delta + 1)(V + bt/3)}\right),$$

the same bound holds for the lower tail by applying the inequality to $-Z$.

A.2 Concentration of the promoted size $S(t)$

We now specialize Theorem A.1 to $S(t)$ and obtain the concentration property required by the LRU-S analysis.

Lemma A.2 (Variance–mean relation). *With $S(t) = \sum_{i \in A} s_i B_i(t)$, we have*

$$V \triangleq \sum_{i \in A} \mathrm{Var}\big(s_i B_i(t)\big) = \sum_{i \in A} s_i^2 b_i(t)\big(1 - b_i(t)\big) \le s_{\max}\, \mathbb{E}S(t).$$

Proof. Since $0 \le b_i(t) \le 1$, we have $b_i(t)(1 - b_i(t)) \le b_i(t)$. Thus

$$V \le \sum_{i \in A} s_i^2 b_i(t) \le s_{\max} \sum_{i \in A} s_i b_i(t) = s_{\max}\, \mathbb{E}S(t). \qquad \square$$

Theorem A.3 (Concentration of $S(t)$). *Let $S(t) = \sum_{i \in A} s_i B_i(t)$ be the total size of promoted KV Groups, and suppose the dependency graph of $\{B_i(t)\}$ has maximum degree at most $L - 1$. Then for any $\varepsilon > 0$, there exists a constant*

$$\theta = \frac{\varepsilon^2}{2L\, s_{\max}\,(1 + \varepsilon/3)}$$

such that

$$\mathbb{P}\big(|S(t) - \mathbb{E}S(t)| \geq \varepsilon\, \mathbb{E}S(t)\big) \leq 2\exp\big(-\theta\, \mathbb{E}S(t)\big).$$

Proof. We prove the upper tail; the lower tail follows by applying the same argument to $-S(t)$ and then using a union bound. Map Theorem A.1 with $Z = S(t)$ and $Y_i = s_i B_i(t)$. By construction, the dependency graph degree satisfies $\Delta \leq L - 1$. Moreover,

$$Y_i - \mathbb{E}Y_i = s_i\big(B_i(t) - b_i(t)\big) \leq s_i \leq s_{\max},$$

so we can set $b = s_{\max}$. Using the quadratic form in Theorem A.1 with deviation $t = \varepsilon\, \mathbb{E}S(t)$ and writing $\mu = \mathbb{E}S(t)$ yields

$$\mathbb{P}\big(S(t) \geq (1 + \varepsilon)\mu\big) \leq \exp\left(-\frac{\varepsilon^2\mu^2}{2(\Delta + 1)\big(V + s_{\max}\varepsilon\mu/3\big)}\right).$$

By Lemma A.2 and $\Delta + 1 \leq L$,

$$2(\Delta + 1)\big(V + s_{\max}\varepsilon\mu/3\big) \leq 2L\, s_{\max}\, \mu\, (1 + \varepsilon/3).$$

Substituting gives

$$\mathbb{P}\big(S(t) \geq (1 + \varepsilon)\mu\big) \leq \exp\left(-\frac{\varepsilon^2}{2L\, s_{\max}\,(1 + \varepsilon/3)}\,\mu\right).$$

The claimed upper-tail bound follows with $\theta = \frac{\varepsilon^2}{2L\, s_{\max}\,(1+\varepsilon/3)}$. The lower tail is identical, and combining both tails with a factor of 2 completes the proof. □

A.3 From concentration to competitiveness

Following [30], the miss event at time t occurs iff the total size of more-recently promoted objects exceeds the cache capacity C. The concentration in Theorem A.3 implies that $S(t)$ is tightly concentrated around $\mathbb{E}S(t)$, uniformly over time scales relevant to the renewal structure of LRU-S. Consequently, the miss probability under LRU-S can be related to the threshold event $\mathbb{E}S(t) \gtrsim C$, up to exponentially small errors in $\mathbb{E}S(t)$. The remainder of the argument (unchanged from [30]) compares this threshold to that of the static optimal policy—selecting the fixed set of objects with the largest frequency-per-size—and concludes a constant-factor gap. Since the static optimal policy is itself asymptotically optimal relative to Belady's algorithm (even with dependent queries) [29], we obtain the following.

Corollary A.3.1. *In the LSM-Tree Group Cache setting where each range scan touches at most L KV Groups and $\max_i s_i = s_{\max} < \infty$, LRU-S is an asymptotically c-competitive cache replacement algorithm, and its miss rate is within a constant factor of both the static optimal policy and Belady's algorithm.*

Received July 2025; revised October 2025; accepted November 2025