# Disco: A Compact Index for LSM-trees

WENSHAO ZHONG, University of Illinois Chicago, USA and TikTok Inc., USA
CHEN CHEN, University of Illinois Chicago, USA
XINGBO WU, Microsoft Research, UK
JAKOB ERIKSSON, University of Illinois Chicago, USA

Many key-value stores and database systems use log-structured merge-trees (LSM-trees) as their storage engines because of their excellent write performance. However, the read performance of LSM-trees is suboptimal due to the overlapping sorted runs. Most existing efforts rely on filters to reduce unnecessary I/Os, but filters fundamentally do not help locate items and often become the bottleneck of the system. We identify that the lack of efficient index is the root cause of subpar read performance in LSM-trees. In this paper, we propose Disco: a compact index for LSM-trees. Disco indexes all the keys in an LSM-tree, so a query does not have to search every run of the LSM-tree. It records compact key representations to minimize the number of key comparisons so as to minimize cache misses and I/Os for both point and range queries. Disco guarantees that both point queries and seeks issue at most one I/O to the underlying runs, achieving an I/O efficiency close to a $B^+$-tree. Disco improves upon REMIX's pioneering multi-run index design with additional compact key representations to help improve read performance. The representations are compact so the cost of persisting Disco to disk is small. Moreover, while a traditional LSM-tree has to choose a more aggressive compaction policy that slows down write performance to have better read performance, a Disco-indexed LSM-tree can employ a write-efficient policy and still have good read performance. Experimental results show that Disco can save I/Os and improve point and range query performance by up to 220% over RocksDB while maintaining efficient writes.

CCS Concepts: • **Information systems → DBMS engine architectures**; *Indexed file organization*; *Data scans*.

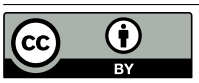
Additional Key Words and Phrases: LSM-tree; key-value store; storage system

**ACM Reference Format:**
Wenshao Zhong, Chen Chen, Xingbo Wu, and Jakob Eriksson. 2025. Disco: A Compact Index for LSM-trees. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 33 (February 2025), 27 pages. https://doi.org/10.1145/3709683

## 1 INTRODUCTION

The log-structured merge-trees (LSM-trees) [40] have been adopted in many database systems [21, 25, 33, 44] as their storage engines because it has better write and relatively acceptable read performance compared to traditional data structures such as $B^+$-trees. Unlike $B^+$-trees that perform expensive in-place writes on updates, LSM-trees enable large sequential disk writes by creating a new run of data on disk for every batch of updates. However, this out-of-place update approach has a negative impact on the read efficiency because a query has to check multiple runs to locate the requested data. To maintain a desirable read performance, LSM-trees use a compaction process to merge and rewrite existing runs, thereby reducing the number of runs that need to be accessed for

Authors' addresses: Wenshao Zhong, University of Illinois Chicago, Chicago, Illinois, USA and TikTok Inc., Bellevue, Washington, USA, wzhong20@uic.edu; Chen Chen, University of Illinois Chicago, Chicago, Illinois, USA, cchen262@uic.edu; Xingbo Wu, Microsoft Research, Cambridge, England, UK, xingbowu@microsoft.com; Jakob Eriksson, University of Illinois Chicago, Chicago, Illinois, USA, jakob@uic.edu.




ACM 2836-6573/2025/2-ART33
https://doi.org/10.1145/3709683

a query. The compaction process trades increased background writes for reduced reads in future queries. Although there exist different compaction policies [16, 17, 43] that make different trade-offs between read and write costs, queries on an LSM-tree are still much more expensive than on a $B^+$-tree. It remains challenging to design an LSM-tree structure that achieves read performance comparable to a $B^+$-tree without sacrificing write efficiency.

Memory-resident Bloom filters [7] are one commonly adopted mechanism in LSM-trees to improve read performance at a negligible write cost. Using Bloom filters allows for identifying and skipping runs that do not contain the key of a search. However, Bloom filters may not be effective in common scenarios because of two major limitations. To begin with, Bloom filters can have a negative impact on query performance with a low memory budget [18, 38, 54]. When a Bloom filter is not memory-resident, loading it from disk causes extra I/Os and caching, which can be even worse than not using Bloom filters at all. Additionally, Bloom filters cannot handle range queries. Although range filters [36, 47, 48, 49, 51] are designed to identify and skip runs that do not contain a range of keys of a query, they are still not used in production systems for two main reasons. Firstly, the state-of-the-art range filters consume as much, if not more, memory than Bloom filters [36, 47, 48, 49, 51], making it inefficient when the memory is under pressure. Secondly, the interface of range filters does not support the iterator semantics widely used in current LSM-tree implementations for traversing key-value pairs incrementally.

Bloom and range filters offer limited benefits as they only reduce the number of runs that must be accessed, without helping locate data in a run directly. After filtering, LSM-trees still need to read additional blocks from disk to locate and retrieve the required data. The inefficiency of reads in LSM-trees is caused by the lack of indexes among the runs of data. Because each sorted run in an LSM-tree is an independent structure, a query has to read the metadata including indexes and filters on each run individually before the requested data can be located. REMIX [53] has explored the idea of multi-run indexes on LSM-trees. In REMIX, sorted runs are sparsely indexed by a $B^+$-tree-like structure, so queries can efficiently locate the required data without the need to search independently on each run. However, when the system has limited caching resources, read operations still incur superfluous read I/Os [48] as a query needs to access as many as $\log(n)$ sorted runs [53], where $n$ is the total number of sorted runs, to locate an item.

As the data size in LSM-tree systems grows, existing solutions struggle to keep a desirable read performance. This calls for a data structure that works well under memory pressure, maintains low read I/O cost similar to $B^+$-trees, and provides efficient write performance for LSM-trees.

We present Disco, a compact multi-run index that achieves efficient reads on LSM-trees. With Disco, point queries and range queries deterministically incur at most one read I/O to the underlying runs to locate the data on disk. Based on this, we build a Disco-indexed LSM-tree named DiscoDB that employs a write-efficient compaction policy, without sacrificing read performance. Experimental results show that Disco performs similarly to $B^+$-trees for read operations in terms of I/Os per query. DiscoDB achieves a significant read performance improvement while maintaining an efficient write performance compare to other LSM-tree designs.

This paper makes the following contributions. First, we identify the problem of inefficient indexes or the lack thereof in LSM-trees. Second, we propose Disco, an efficient index design for LSM-trees, that combines the compact key representations with REMIX's multi-run index. Finally, we build and evaluate Disco as well as DiscoDB. Experimental results show that Disco can efficient index items on LSM-trees so that a point and range query use minimal I/Os when the cache budget is limited. We hope this work can fill the missing piece in the research of LSM-tree indexes.

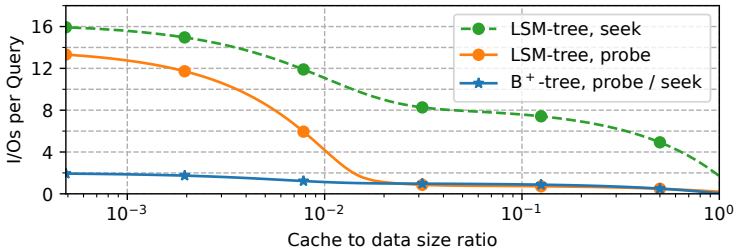



Fig. 1. Numbers of I/Os per query for LSM-trees and B$^+$-trees with different caching budgets. The LSM-tree has 8 sorted runs, each is equipped with Bloom filters that will be effective only in point queries (probe).

## 2 BACKGROUND AND MOTIVATION

Over time, the ratio of main memory to permanent storage in computer systems has been shrinking because the price drop in DRAM has always been slower than that of hard drives and SSDs [15, 37, 38, 54]. As a result, many LSM-tree deployments use a modestly-sized main memory paired with much larger hard drives or SSDs to serve more data at a reasonable hardware cost. At the time of writing, a 32 GB DRAM stick costs about the same as an 4 TB HDD or a 2 TB SSD [15, 37], resulting in a memory-to-disk ratio of 1:128 or 1:64, respectively. Thus, it is increasingly important to get maximum utility out of relatively precious main memory resources.

Bloom filters are designed to improve read using a small memory footprint, but it is facing growing difficulties to scale up with the increasing disk-to-memory ratios. The assumption that Bloom filters can be cached in main memory no longer holds. To show how expensive accessing multiple runs in LSM-trees can become, we conduct an experiment to compare the I/O costs of an LSM-tree and a B$^+$-tree under different caching budgets.[1] Figure 1 shows the average I/Os per query with varying memory budgets. As a read optimized index structure, the B$^+$-tree uses at most two I/Os to finish a query across all memory budgets. The LSM-tree has a point query cost similar to the B$^+$-tree only when the cache budget is more than 2% of the dataset size, which are sufficient to cache Bloom filters. However, when the memory budget is limited, it requires 6× more I/Os than the B$^+$-tree due to the cache misses when accessing Bloom filters. Even with Bloom filters, the point queries have about 80% of I/O cost of range queries, which directly search the requested key on each run. In conclusion, Bloom filters can incur superfluous I/Os and even degrade the read performance when the caching resource is scarce.

Despite being useful for point queries when cached in memory, Bloom filters cannot handle range queries, which are an important workload in database systems. As one of the major workloads, range queries are included in the ubiquitous Yahoo Cloud Serving Benchmark (YCSB) [13]. Many relational database systems [32, 45] are built on top of LSM-trees. They issue range queries to the underlying LSM-trees for every SQL query [39]. Hence, range queries have been used as one of the

[1]In the experiment, we issue queries in a uniformly random manner and record the I/O traces of a small partition in an LSM-tree or B$^+$-tree. The partitions have 256 MB of key-value pairs, consisting of 1.8 millions email address keys and 120-byte values. The I/O traces are later replayed on a cache simulator with different cache size ranging from 0.1% to 100% of the dataset size to measure the number of I/Os per query. The LSM-tree has eight overlapping sorted runs each of which is 32 MB, matching the size of SSTable in popular implementation of LSM-trees [21, 22]. Each run is equipped with a Bloom filter that is configured to use a typical setup of 10 bits per key [22] to maintain a false positive rate about 1%. A point query on the LSM-tree first checks the Bloom filters, while a range query searches on each run and sort-merges the results. The B$^+$-tree stores all the data in sorted order in one file, and it does not use Bloom filters so the point and range queries in B$^+$-tree share exactly the same code path.

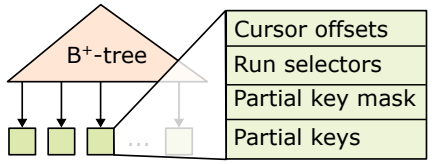



Fig. 2. An overview of Disco. Each square box represents a segment in Disco.

key performance indicators in many systems [24, 36, 46, 51, 53]. Achieving efficient range queries is always one of the desiderata in many systems.

It is challenging to improve range query performance on LSM-trees. Many systems implement range queries using iterators that support seek and next operations. A range query first performs a seek operation to find the start key of the query. Then, a number of next operations could be used to retrieve the subsequent keys until a certain condition is met. Oftentimes, the user-provided search key does not exist in the LSM-tree. In that case, a seek operation needs to find the smallest key that is greater than the search key. Because the sorted runs overlap in ranges, an LSM-tree needs to perform a seek operation on every run to find the required data, slowing down range queries.

Range filters have been proposed to try to solve the problem similarly to Bloom filters. However, range filters do not support the iterator semantics. They handle generalized range queries in the form of: "Is there a key in $[l, h]$ in the set $S$?" [51]. To apply range filters in LSM-trees, users must input a predetermined query range, which may not be known in practical scenarios. For example, in YCSB workload E, a range query consists of a seek and 50 next operations. Because a seek operation corresponds to an open range query, range filters can help only when the search key is greater than the largest key in the run.

Although filters can rule out runs that do not contain the target key and save I/Os when the memory is sufficiently large to cache them, they may also harm query performance because they do not help locate the keys in a run. For example, a range filter is expected to return positive for long range queries because the range is likely to cover most sorted runs. When the filter returns true, LSM-trees have to load the index blocks [20, 26] from disk to locate the desired data. In that case, the range filter does not save I/Os but brings overhead to queries. The same problem exists when filtering existing keys using point filters. Had we known that the keys of interest do exist, we would not have consulted filters.

The limitations of the filtering approaches have motivated the research on auxiliary indexes on LSM-trees. In [53], the authors propose a multi-run index on LSM-trees named REMIX to help improve range query performance. It records the access path of the keys so that a range query does not need to sort-merge the keys on the fly. It is effective only when the system has plentiful caching budget or runs on fast storage devices such as 3D-XPoint (Intel's Optane SSD) where the cost of sort-merging is more prominent. However, if the system runs on commodity hardware such as an SSD with a limited caching budget, the cost of any additional I/O quickly outstrips the sort-merge savings, so REMIX becomes ineffective.

In this paper, we introduce Disco, an auxiliary index on LSM-trees, that maintains the write efficiency of LSM-trees and achieves a similar read efficiency to B$^+$-trees even under scarce memory budgets.

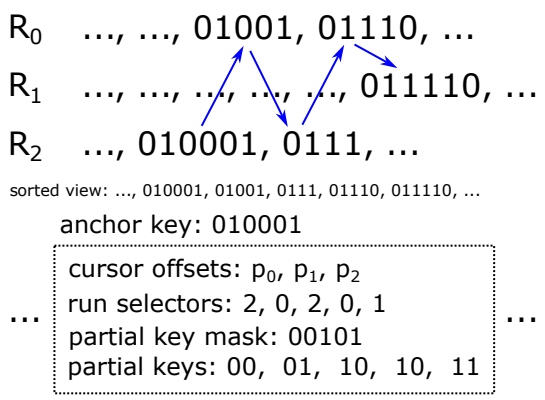



Fig. 3. An example segment in Disco. The segment contains of 5 keys: `0100 01`, `0100 1`, `0111`, `0111 0`, and `0111 10`. The arrows show the ordering of the keys.

## 3 DESIGN

At a high level, Disco is an index across all runs of an LSM-tree. As shown in Figure 2, Disco divides the ordered set of all key-value pairs contained in the tree into short *segments* indexed by a *search tree*. Here, each segment indexes a bounded number of consecutive keys. The use of segments effectively reduces the height of the search tree, ensuring that the tree has a small memory footprint.

In Section 3.1, We describe how the search tree in Disco efficiently locates a target segment that indexes the target key and explain how the metadata in a segment is used to allow for accessing any keys in the segment efficiently. The metadata of segments themselves are small contiguous data structures that may be kept in memory, or read from disk using a single I/O. In Section 3.2, we propose a novel, compact representation of a segment. The compact representation enables Disco to identify the position of a target key using at most one I/O to the runs of the LSM-tree. As a result, Disco is able to return the target key within any one of the runs, using 1–2 I/O operations. Section 3.3 explains how Disco is efficiently maintained when the LSM-tree undergoes compactions. While Disco speeds up merging in compactions, a compaction can also provide hints to help update Disco.

### 3.1 The Search Tree and Segments

The keys in a Disco-indexed LSM-tree are divided into variable length segments. Each segment has an *anchor key* that represents the lower bound of the segment and also the upper bound (exclusive) of the previous segment if it exists. The segments are indexed by a $B^+$-tree by their anchor keys, while the values encode each segment's metadata. Thus, a query finds the target segment by finding the largest key that is less than or equal to the user input key.

Figure 3 shows an example of a segment in the search tree covering a subset of data in three runs. In this example, we use the smallest key in a segment as the anchor key. We can reduce storage cost further by using only the minimal unique prefix of this key as the segment's anchor key.

The metadata of a segment contains a set of *cursor offsets*, a set of *run selectors*, and a *compact segment representation* that consists of a *partial key mask*, and *partial keys*. The cursor offsets and run selectors are borrowed from REMIX [53] and they serve similar purposes. The usage of the partial key mask and partial keys will be discussed later in Section 3.2. Cursor offsets record the positions of the smallest key in each run that belongs to the segment. In our example, $p_0$, $p_1$, and $p_2$

point to the string keys `0100 1`, `0111 10`, and `0100 01`, respectively. The iterators of each run will be initialized to point to those keys. Run selectors correspond to the keys in a segment. It records the access order of the runs when scanning the segment. In this example, the run selectors 2, 0, 2, 0, 1 indicate that the sorted view of this segment can be generated by reading the keys pointed to by the iterators of the runs in the order of $R_2, R_0, R_2, R_0, R_1$. After we finish reading a key, the iterator will move forward to track the next key on the run.

Segment metadata enables Disco to efficiently access any keys in the segment using minimal I/Os. We can access a key at any position in the segment by counting the number of occurrences of run selectors. In this example, if we want to access the third key `0111` in the segment we can first look at the third run selector, 2, the one corresponding to the same position. By looking at the run selectors preceding the third run selector in this segment, it is trivial to find that this is the second occurrence in the segment. The third key could be retrieved by reading the second key in $R_2$ after we initialize the iterator of the run with cursor offset $p_2$. With the help of the segment metadata, we can randomly access any key in a segment by its index using only one I/O.

### 3.2 The Compact Segment Representation

Central to the compact segment representation is the idea of discriminative bits. Below, we first provide an intuition, using string keys and discriminative characters for illustrative purposes. We then describe the actual design in detail, which is based on individual discriminative bits, rather than characters.

*3.2.1 Intuition: discriminative characters.* The discriminative characters of a segment of keys are the characters that make a difference to the ordering of the keys within the segment. For example, if a segment contains three string keys "disc**o**n**n**ected", "disc**o**n**t**inuous", and "disc**r**e**p**ancy", only the character after "disc" and the seventh character, both bold-faced, are discriminative. The other characters can be elided, without altering the internal order.

Conceptually, it is sufficient to represent only the discriminative characters in the index, as opposed to the entire keys, which helps keep the index compact. In our running example above, we may use the string "disc" as the anchor key for our segment. We may then represent the individual keys using only the fifth and seventh characters "on", "ot" and "rp".

Given a query for a key that falls within the segment key range, say "disc**o**u**n**t", we can extract the discriminative characters from the key, and quickly find its position within the sorted index. In this case, the compact search key "on" would match with the compact representation of "disc**o**n**n**ected", yielding the on-disk location where the key "disc**o**u**n**t" may (or may not, in this example) be found.

The search key "discovery" presents a problem, however. Here, the 'v' is in fact discriminative for the search key, but not represented in the segment. After extracting the two discriminative characters of the segment, which were selected without knowledge of the search key, the compact representation of "disc**o**v**e**ry" is "oe". Using "oe" to search the compact key would lead us to the location of "disc**o**n**n**ected", namely "on" in compact keys, which is not the segment key result that should have been found using the original search key.

Should we need to find the correct segment key, we can achieve this goal with a single additional search, after a small update to the compact search key. First, we compare the full search key "discovery" and the result key "disconnected" and find their shared prefix: "disco". Intuitively, the key we are looking for is the largest key following the shared prefix, because of the 'v' in "disco**v**ery" being greater than the 'n' in "disco**n**nected". To find this key, we may simply set all the discriminative characters in the search key, after this shared prefix, to the maximum value (say, 'z'). Thus, we search the index a second time, but this time for "oz" instead of "on", which finally yields the search result "disc**r**e**p**ancy", the correct result key in the segment.

**Algorithm 1** The process of building a partial key mask and partial keys

```
1: function BUILD(string_keys)
2:     pkmask ← 000...0
3:     pkeys ← ∅
4:     for i ∈ [0, string_keys.len − 2] do
5:         lcp ← BITWISE_LCP(string_keys[i], string_keys[i+1])
6:         MARK_BIT_AT(pkmask, lcp)
7:     for i ∈ [0, string_keys.len − 1] do
8:         pkeys[i] ← BIT_EXTRACT(string_keys[i], pkmask)
9:     return pkmask, pkeys
```

As noted, Disco does not use discriminative characters, but rather discriminative bits in key strings. A detailed description follows below.

*3.2.2 Design details: discriminative bits.* The segment representation encodes a group of sorted variable-length string keys into an integer array, shown as *partial keys* in Figure 3, and a bit vector, *partial key mask*, where 1s mark the positions of discriminative bits. The partial key mask will be used to extract bits from a search key from user input and generate a partial key of the search key. The new partial key is then used for search on the sorted integer array.

*Construction.* The compact segment representation is constructed in two steps. The first step is to compute the partial key mask of a group of keys. Then we will use the partial key mask to extract the partial keys from them. The time complexity of this step is linear to the number of keys in the group. The pseudocode of this process is shown in Algorithm 1.

We derive the partial key mask by iterating through each pair of neighboring keys in sorted order. The first loop in Algorithm 1 shows the steps. The partial key mask is a bit vector that marks the position of discriminative bits. The discriminative bit of a pair of keys is defined as the first mismatching bit between the two. The location of the discriminative bit is captured by the length of the bitwise common prefix of the two strings. We set the corresponding position to one in the partial key mask to collect all the discriminative bit positions. For a group of $n$ keys, we need to go through all $n-1$ pairs of neighboring keys. It follows that $n$ keys can produce $n-1$ discriminative bits, out of which at most $n-1$ are distinct. As a result, at most $n-1$ bits can be set to 1 in the partial key mask.

A partial key is generated by extracting the bits from a string key at the positions where the partial key mask is set to one. The extraction uses parallel bit extract operations [11] to select non-contiguous bits from strings. The partial key mask is used as the bit mask for the parallel bit extract operations. For one string key, it outputs a partial key whose length equals to the total number of bits set to 1 in the partial key mask. Because the partial key mask marks where each pair of keys start to differ, the generated partial keys are unique. Since the original string keys are sorted and the partial key mask captures where the bit changes from 0 to 1, the partial keys are inherently sorted.

Figure 4 shows an example of generating a partial key mask and partial keys. We iterate through each pair of adjacent keys to find the discriminative bits. In this example, the first pair key$_1$ `0100`**`0`**`1` and key$_2$ `0100`**`1`** differ at the fifth bit indicated by the dashed box, so the discriminative bit is at bit position 4 in zero-based index. The second pair key$_2$ `01`**`0`**`0`1 and key$_3$ `01`**`1`**1 give us another discriminative bit at bit position 2. If one key is the prefix of another and the longer one ends at `0`, such as the third pair key$_3$ `0111` and key$_4$ `0111 0` in this example, then there is not a discriminative bit for them. The final pair key$_4$ `0111`**`0`** and key$_5$ `0111`**`1`**`0` happen to have the discriminative bit at

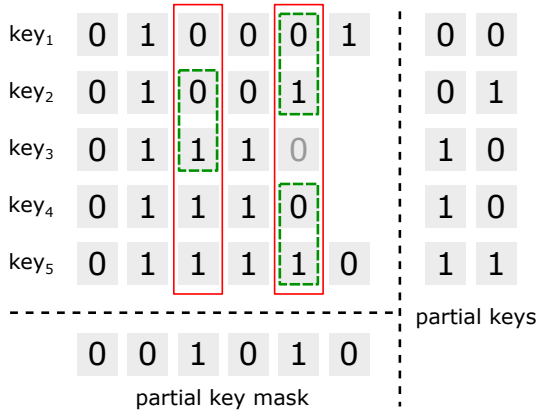



Fig. 4. An example of generating partial key mask and partial keys for binary string keys: {0100 01, 0100 1, 0111, 0111 0, 0111 10}. These keys generate a partial key mask: 0010 10 and partial keys {00, 01, 10, 10, 11}.

**Algorithm 2** The algorithm of point queries on a compact segment representation

```
1:  function POINT_QUERY(search_key, pkeys, pkmask)
2:      pkey ← BIT_EXTRACT(search_key, pkmask)
3:      pos ← FIND_EXACT_MATCH(pkeys, pkey)
4:      if pos = Not found then
5:          return Not found
6:      result_key ← STRING_KEY_AT(pos)
7:      cmp ← STR_CMP(probe_key, search_key)
8:      if cmp = 0 then
9:          return pos
10:     return Not found
```

bit position 4. In this case, three discriminative bits result in two bits set to one in the partial key mask 0010 1.

To extract the partial keys, we iterate through the string keys and perform parallel bit extract operations. As shown in Figure 4, the partial keys selected from the bits at bit positions 2 and 4. When a bit position is out of bound for a key, such as the case for $key_3$, we will logically pad 0s to it. Because $key_3$ and $key_4$ do not generate a discriminative bit, their partial keys are the same. We treat them as the same key during queries.

*Point queries.* Performing point queries on the compact segment representation is very efficient. Similarly to a point filter, an empty point query often returns key-does-not-exist without using any I/Os to the LSM-tree, and it sometimes needs to issue I/Os to verify the existence of the key when it has false positives. The compact segment representation guarantees that only one I/O is needed in that case. Point queries of existing keys take at most one I/O to locate the key.

Algorithm 2 shows the process of performing point queries. Given a search key, the first step is to extract the partial key of the search key using the partial key mask. We extract the partial key from the search key in the same way how partial keys are extracted. If the search key is shorter than the discriminative bit vector, we logically pad 0s at the end of the search key. That will give us a partial key of the same length as the stored partial keys.

**Algorithm 3** The algorithm of range queries on a compact segment representation

```
1:  function RANGE_QUERY(search_key, pkeys, pkmask)
2:      pkey ← BIT_EXTRACT(search_key, pkmask)
3:      pos ← LONGEST_PREFIX_MATCH(pkeys, pkey)
4:      probe_key ← STRING_KEY_AT(pos)
5:      cmp ← STR_CMP(probe_key, search_key)
6:      if cmp = 0 then
7:          return pos
8:      n_mismatch ← MISMATCH_COUNT(probe_key, search_key, pkmask)
9:      if mismatch_count = 0 then
10:         if cmp < 0 then
11:             return pos
12:         else
13:             return pos + 1
14:     if cmp < 0 then
15:         new_pkey ← PKEY_LOWEST(pkey, n_mismatch)
16:     else
17:         new_pkey ← PKEY_HIGHEST(pkey, n_mismatch)
18:     pos ← PKEY_SEARCH(pkeys, new_pkey)
19:     return pos
```

We then search in the partial key array to find an entry that exactly matches the partial key extracted from the search key. If an exact match is not found, we can conclude that the search key does not exist without accessing any keys in the segment. Otherwise, there exists a partial key that matches the partial key of the search key. We then issue an I/O to access the string key at the corresponding position as shown at line 6 in Algorithm 2. We return the index of the result key only if it matches the search key.

*Range queries.* It takes additional steps to perform a range query on the compact segment representation. Similar to performing a point query, the first step of doing a range query is to extract a partial key of the search key from the user. The partial key is then used as the search key to perform a longest prefix match (LPM) in the array of partial keys. It returns the position of a partial key in the partial key array that best matches the prefix of the partial key extracted from the user. Similarly to point queries, we will issue an I/O to access the string key at the position and compare it to the search key. It directly returns the result position if the search key happens to match the accessed string key. Otherwise, it evaluates how many bits in the partial key mask that these two keys have matched. Analogous to the intuition in Section 3.2.1, we can derive an updated partial key based on the knowledge of the number of matched bits and the ordering between the accessed string key and the search key. Finally, the result position will be found by performing a regular search on the partial keys with the updated partial key of the search key. The detailed process of performing a query on the compact segment representation is shown in Algorithms 3 and 4.

The following paragraphs discuss how we use the partial key mask and partial keys to locate a search key in detail. For the sake of simplicity, we define a *seek* operation as finding the smallest key that is greater than or equal to the search key. We assume each range query starts with a seek operation. To perform a range query, the first step is similar to point queries. What is different is that after extracting the partial key, we might not find a partial key that exactly matches the partial key of the search key. Should we indeed find a matching partial key, the string key at the result

**Algorithm 4** Auxiliary functions of range queries

```
function MISMATCH_COUNT(probe_key, search_key, pkmask)
    lcp_bit ← BITEWISE_LCP(probe_key, search_key)
    matched_bits ← POP_COUNT(pkmask[: lcp_bit])
    tot_ones ← POP_COUNT(pkmask)
    return tot_ones - matched_bits
function PKEY_HIGHEST(pkey, mismatch_count)
    bit ← SHIFT_LEFT(1, mismatch_count)
    mask ← bit - 1
    highest ← BITWISE_OR(highest, mask)
    highest ← highest + 1
    return highest
function PKEY_LOWEST(pkey, mismatch_count)
    bit ← SHIFT_LEFT(1, mismatch_count)
    mask ← REVERSE_BIT(bit - 1)
    if pkey < bit then
        return 0
    lowest ← BITWISE_AND(pkey, mask)
    return lowest
```

position might not match the search key: a case of "false positive". In both cases, we still want to finish the range query using the least I/Os. To do that, searching in the partial keys needs to mimic the matching process on a radix tree (trie) where we find the LPM partial key.

When the position of the LPM partial key is determined, we will read the actual string key at the same position in the segment, denoted as the *probe key* in the following. This is the only place where we issue an I/O and do a string comparison. Having the actual string key, we can determine if it is an exact match. If it is, we return the position immediately. Otherwise, we learn that the probe key is either greater or smaller than the search key.

Having accessed the actual string key infers not just the ordering of the search key and the probe key, but also the number of matched characters or bits between the two keys. The length of the longest common prefix (LCP) exactly captures this metric. If we think of all the keys in a segment are organized in a trie, then each internal node represents a common prefix of all its leaves, and all the keys in the subtree rooted in the node share the common prefix of that node. A seek operation of the search key is equivalent to finding either *the leftmost leaf* or *the next key after the rightmost leaf* in the subtree. If the probe key is greater than the search tree, we need to find the leftmost leaf. Otherwise, the probe key is smaller than the search tree, and we look for the next key after the rightmost leaf.

Directly doing range queries on the real keys triggers excessive I/Os. Instead, we construct a new partial key that would return the same position when it is used to perform a search in the partial keys. To do that, the first step is to find the number of bits that the LCP matches the partial key mask. For easier of handling in the next step, we compute the number of mismatching bits in the implementation as shown in function `mismatch_count` in Algorithm 4. If the number of mismatching bits is zero, that implies that either the search key or the probe key is the prefix of the other. The function returns the current position or the next position depending on the key comparison result.

In the second step, we will use the number of mismatching bits to construct a new partial key that could lead to the leftmost leaf or the key after the rightmost leaf of it. If the search key is greater than the probe key, we will construct a new partial key that fills all the trailing mismatching

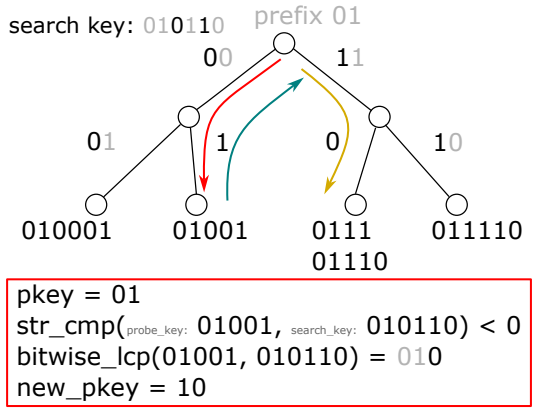



Fig. 5. An example of range query using Disco illustrated in a radix tree (trie).

bits in the partial key of the search key with all `0`s, as shown in function `pkey_lowest`. Otherwise, we fill them with all `1`s then arithmetically plus one. The new partial key is then used to do a seek operation in the partial keys. The position we get from this search is guaranteed to be the position of a seek operation of search key in the segment.

*A range query example.* To better illustrate how range queries mimic the search on a trie, we construct a Patricia trie in Figure 5 using the example keys in Figure 4. With the search key `0101 10` and the partial key mask `0010 10`, we get the partial key of the search key `01`. Next, we try to find the LPM partial key, which is equivalent to following the Patricia trie to match the tokens. It goes to the partial key `01`, which has the corresponding string key `0100 1`. This key is denoted as the probe key in Algorithm 3. By comparing the probe key to the search key, we learn that the search key is greater than the probe key hence we need to find the key after the rightmost leave of the LPM subtree. We then call the auxiliary function `mismatch_count` in Algorithm 4 to find the number of mismatching bits in the partial keys. It shows that the probe key and the search key share 3 bits of common prefix, which translates to only one bit `0` matched in the partial key mask. Correspondingly, there is one mismatching bit in the partial key mask. Knowing the probe key is smaller than the search key, we construct a new partial key using the auxiliary function `pkey_highest` in Algorithm 4. It fills the trailing mismatching bit with a `1` and then adds one to it, resulting in a new partial key `10`. When we use this new key to search in the array again, it returns the position of the third partial key in the array. This position is indeed the result position of the finding the search key `0101 10` in this group of string keys.

### 3.3 Rebuilding Disco

Because a Disco is an index structure built for a set of runs, whenever a new run is generated or some existing runs get replaced, the Disco needs to be updated accordingly. One straightforward way to update a Disco is to build a new Disco. Given a new set of sorted runs, we can build a Disco by scanning and sort-merging all the keys in those runs. Although the write I/O could be kept small because of the compact design, scanning all data in those runs can incur excessive read I/O that might slow down the building process and the overall write performance. In this section, we analyze the rebuilding cost, discuss the trade-off we make in Disco, and show the techniques we employ to reduce the rebuilding cost. We will evaluate how much read and write I/O Disco triggers during write intensive workloads and how Disco affects the write performance in Section 4.2.

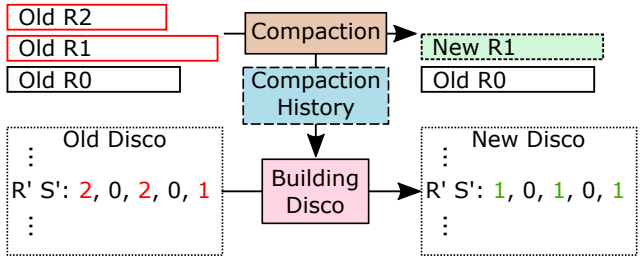



Fig. 6. A diagram illustrating how to use compaction history for building Disco. In this diagram two old runs (R1 and R2) get merged and become new R1. R' S' denotes Run Selectors.

*Disco reduces compaction sort-merging overhead.* LSM-trees maintain background compaction processes that periodically organize sorted runs to maintain its structures. In a Disco-indexed LSM-tree, such as DiscoDB, the existing runs are already indexed by an existing Disco. When doing compactions, the existing Disco can help accelerate sort-merging without excessive in-memory comparisons.

*Building new Disco makes use of compaction history.* After a compaction, the old Disco is obsolete and needs to be replaced. The cost for building the new Disco would be prohibitively high if we naively scan all the runs to build the new Disco. However, we observe that compactions only rewrite existing keys in the old runs onto the new runs and the metadata in the new Disco mostly stay the same as the old one except that the run selectors and the cursor offsets for the merged runs are updated. Based on this, we implement a mechanism for building the new Disco from the old Disco with updated run selectors and cursor offsets.

Figure 6 shows a diagram of the mechanism. The compaction process generates and ships a compaction history to the Disco building process. The data structure of compaction history is lightweight. It describes what keys get merged in the new run and the corresponding cursor offset changes. The Disco building process uses the compaction history to update the run selectors and cursor offsets without reading the new set of runs. Because compactions in LSM-trees inevitably need to load the to-be-merged runs to memory to perform merges, the I/O cost of rebuilding Disco only includes reading the old Disco and writing the new one.

*Unaffected segments do not trigger I/O.* LSM-trees generate a new run for a batch of updates. After a new run is generated, the old Disco needs to be updated to incorporate the new run into the index. Directly scanning the new set of runs to build the Disco can cause excessive read I/O. The problem is prominent especially when a small run is generated and the Disco has to be rebuilt on a large amount of existing data. However, we find in this situation that most of the segments in the new Disco only need to update the cursor offsets for the new run and only a small percentage of the segments have to be rebuilt by scanning. To save I/O to rebuild the new Disco, we skip accessing the segments that are not affected by the updates.

When there is a new run generated, rebuilding a new Disco can be thought of as inserting the new keys to the existing Disco. To avoid scanning the existing runs, we try to maintain the existing search tree in Disco as much as possible. We use the search tree to find which segment each new key belongs to. If a segment does not receive any new keys, all the segment metadata stay the same except for cursor offsets. In this case, we do not need to issue any I/Os to the underlying runs. The cursor offset of the new run will be appended to the new Disco. When a segment is affected by the new keys, we try to incorporate them in the segment if the lengths of partial keys are still within

threshold; otherwise, the segment is split to incorporate the new keys. In this way, the read I/O to rebuild the Disco only depends on the update size and not the existing data size.

*Batch I/O reduces rebuilding cost.* If scanning existing runs is unavoidable to rebuild Disco, the I/O cost can still stay low. Because using scans to build Disco has a sequential access pattern, the read I/O is less expensive by doing I/O in batches. This can be easily implemented by calling `posix_fadvise` and `posix_madvise` to change the read-ahead window before and after building Disco. Moreover, implementations can choose to rebuild Disco asynchronously so that Disco does not block front-end write and provides efficient read subsequently.

### 3.4 Storage Cost of Disco

The storage cost of Disco reflects both read and write efficiency. A smaller storage cost means a lower cost to rebuild the Disco and a high cache utilization when serving queries. The metadata of Disco can be broken down into five parts: anchor keys, cursor offsets, run selectors, partial key mask, and partial keys. We use $D$ to denote the number of keys in a segment which is also referred as the segment size. The anchor key takes about $1/D$ of the total key size on overage. If we assume the size of a cursor offset is $S$ bytes and the number of runs is $H$, the cursor offsets in a segment in Disco takes $S \times H$ bytes while a run selector requires $\lceil \log_2(H) \rceil$ bits. Because the partial key mask record the bit positions where the neighboring keys differ, without any encoding optimizations a partial key mask is expected to require the average key size $\bar{L}$ of space in a segment. We assume each partial key takes $P$ bytes. Adding all the five parts together, Disco is expected to use $\left((2\bar{L} + SH)/D + \lceil \log_2(H) \rceil / 8 + P\right)$ bytes/key.

We estimate the actual storage cost by plugging practical numbers into the equation. We use cursor offset of 4 bytes ($S = 4$), enabling it to address 4 GB of space in a run. We evaluate the situation where the Disco indexes 8 sorted runs ($H = 8$). A practical implementation would use 1 byte for a run selector which can index 256 runs at most. Combining all the configurations, we have the storage cost of $\left((2\bar{L} + 32)/D + 1 + P\right)$ bytes per key. A practical configuration chooses the partial key size from 16-bit, 32-bit, and 64-bit integers, corresponding to $P = 2$, $P = 4$, and $P = 8$ respectively. The choice of partial key size affects the average segment size. We will use 32-bit partial keys ($P = 8$) as the typical configuration. Section 3.5 takes a more in-depth discussion on the segment sizes and the partial key sizes.

Table 1 presents a summary of the storage costs of Disco in comparisons with REMIX and SSTable. We use the average key and value sizes collected in production environments [2, 9, 19]. For SSTable, we compare to Bloom filters configured to 10 bits per key [22]. The block index [20, 26] is estimated as the average key-value size plus a block handle size (4 B) divide the number of key-value pairs in a 4 KB block. The storage cost of REMIX is calculated according to the configurations in [53]. The last column shows the size ratio of Disco to all the key-value data it indexes. The storage cost of Disco is up to 29% of the key-value data when the value size is small such as in USR store. For most of the key-value stores, Disco takes less than 5% of the key-value data.

### 3.5 Implementations

The algorithms and examples presented in previous sections give a high-level idea of our design. We did not mention how the operations on Disco tries are performed and how we apply them to build an LSM-tree based database. In this section, we will fill those blanks by providing some implementation details that could help practitioners implement efficient key-value store systems using Disco.

*Partial key lengths.* The partial key lengths directly determine the storage cost of Disco because we need to store one partial key for every key. It also indirectly affects the storage cost because it

Table 1. Comparison of Disco storage cost to SSTable and REMIX with real-world key-value sizes. BI stands for Block Index. BF stands for Bloom Filter. The last column shows the size ratio of Disco to the key-value data. For Disco when $P = 2, D = 24$; $P = 4, D = 64$; $P = 8, D = 256$.

| Workload [2, 9, 19] | Average Key Size | Average Value Size | Bytes/Key | | | | | | | | $\frac{\text{Disco}}{\text{data}}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | SSTable | | REMIX ($H = 8$) | | | Disco ($H = 8$) | | | |
| | | | BI | BI+BF | $D = 16$ | $D = 32$ | $D = 64$ | $P = 2$ | $P = 4$ | $P = 8$ | ($P$=4) |
| UDB | 27.1 | 126.7 | 1.2 | 2.4 | 4.1 | 2.2 | 1.3 | 6.6 | 6.3 | 9.3 | 4.13% |
| Zippy | 47.9 | 42.9 | 1.2 | 2.4 | 5.4 | 2.9 | 1.6 | 8.3 | 7.0 | 9.5 | 7.71% |
| UP2X | 10.45 | 46.8 | 0.2 | 1.5 | 3.0 | 1.7 | 1.0 | 5.2 | 5.8 | 9.2 | 10.18% |
| USR | 19 | 2 | 0.1 | 1.4 | 3.6 | 2.0 | 1.2 | 5.9 | 6.1 | 9.3 | 29.02% |
| APP | 38 | 245 | 2.9 | 4.2 | 4.8 | 2.6 | 1.5 | 7.5 | 6.7 | 9.4 | 2.36% |
| ETC | 41 | 358 | 4.4 | 5.6 | 4.9 | 2.7 | 1.5 | 7.8 | 6.8 | 9.4 | 1.70% |
| VAR | 35 | 115 | 1.4 | 2.7 | 4.6 | 2.5 | 1.4 | 7.3 | 6.6 | 9.4 | 4.40% |
| SYS | 28 | 396 | 3.3 | 4.6 | 4.1 | 2.3 | 1.3 | 6.7 | 6.4 | 9.3 | 1.50% |

determines the maximum segment size. By the number of combinations of fixed length bit strings, a $n$ bit partial key could have at most $2^n$ unique partial keys. It is obvious that $n$ bit partial key mask could generate at least $n + 1$ unique partial keys. Depending on the datasets, $n$ bit partial keys can encode from $n + 1$ to $2^n$ keys.

To see how partial key lengths affect the maximum segment sizes we use some real-world datasets [2] and build Disco by incorporating keys into a segment until the partial keys cannot differentiate new keys. We use the partial key lengths of 16 bits, 32 bits, and 64 bits. We then measure the segment sizes for some real-world datasets. We find that 32-bit partial keys could distinguish about 50 to 400 string keys on average. 16-bit partial keys have similar segment size distributions with the mean values ranging from 21 to 50. The mean values for 64-bit partial keys are much larger, reaching the magnitude of 1,000.

Based on that, we find that it is a better trade-off to use 32-bit partial keys in a practical implementation. With 32-bit, at least 33 string keys could be distinguished. It guarantees that the search tree picks an anchor key for at least every 33 keys, so the search tree could be kept compact. With 32-bit partial keys, it introduces a modest storage cost of 4 bytes per key. In addition, because 32-bit integers are ubiquitous in varies of hardware platforms, bitwise operations could be performed most efficiently.

*Bit extractions.* As shown in Algorithms 1, 2, and 3, both the build and query processeses involve bit extraction operations. The efficiency of bit extraction operations are the key to achieve efficient operations on Disco. In our implementation, we divide the strings into 8 bytes units to cut down the number of iterations for long strings. We also use the *PEXT* instruction [11] in BMI2 instruction set available on modern CPUs to perform efficient extractions.

*The optimized file layout.* The design goal of Disco is to achieve I/O efficient queries. It is critical for a Disco file itself to be I/O efficient. We store Disco in a $B^+$-tree like layout. The node sizes are aligned to 4 KB blocks, the most commonly used block I/O unit. Each anchor key points to a position that stores all the metadata of a segment. Metadata blocks are also aligned to 4 KB blocks to prevent the segment metadata from being fragmented into two different I/O blocks.

*The LSM-tree structure.* We build an LSM-tree based key-value store, DiscoDB, to showcase the effectiveness of Disco. DiscoDB divides the keys into partitions. Each partition consists of a set of overlapping runs that are indexed by a Disco. Similar to other traditional LSM-tree based indexes,

[2]Including email addresses, amazon reviews, DBLP titles, DBLP URLs, and words in English dictionary.

it buffers updates in memory and periodically flushes them to respective partitions. When trying to flush data from memory to partitions on disk, DiscoDB will reject flushing to a partition and send the data back to memory if the updates are too small to maintain the cost of rebuilding Disco low. The Disco of a partition will be rebuilt when a new run is flushed to a partition from MemTable. DiscoDB also maintains a background compaction process that merges selected runs based on the estimated compaction cost.

## 4 EVALUATION

In this section, we evaluate the performance of Disco and DiscoDB and compare them against traditional LSM-tree and REMIX as well as the KV-store systems derived from them, namely RocksDB and RemixDB. To further show the trade-off of Disco, we compare with a baseline LSM-tree index implementation, which we coined "Full-Index", that organizes all the keys and pointers to the key-values on a LSM-tree in a $B^+$-tree. We also provide another implementation, referred to as "No-Index", that ablates the index component of DiscoDB to show the overhead of maintaining indexes.

We first evaluate the numbers of I/O operations (I/Os below) per query using micro-benchmarks to see how Disco saves I/Os. We then show the overall read and write performance of the KV-stores using workloads that emulate real-world scenarios, as well as the standardized YCSB benchmark suite.

### 4.1 Micro-benchmark Experiments

To evaluate Disco performance, we implement a micro-benchmark framework that measures I/Os per query of sorted runs. These experiments have the same setup as the motivational experiment in Section 2. They simulate the partitions of KV-stores as a set of sorted runs filled with keys that are randomly selected from an email address dataset.

Together, the partitions contain 1.8 million email address keys, which together with 120 byte values form 256 MB of key-value data. We consider point queries (probes), which return the existence or non-existence of a given query key as well as range queries (seeks) which find the smallest key greater than or equal to the query key. The experiments issue queries using a uniform random key distribution and record the I/O traces. The I/O traces are later replayed on a cache simulator that implements a classic least recently used (LRU) cache replacement policy with different cache sizes ranging from 0.1% to 100% of the dataset size.

In the micro-benchmark experiments, we compare the number of I/Os per query of Disco to that of REMIX and LSM-tree. We also include the results of $B^+$-tree for reference. To compare these data structures fairly, we use the same 256 MB key-value pairs for all. The $B^+$-tree organizes all data in a single sorted file while LSM-tree, REMIX, and Disco use 8 sorted runs.

*Point query experiments.* In the first set of experiments, we focus on the effectiveness of Disco in reducing the number of I/Os associated with point queries. We measure and compare the average I/Os per point query of Disco, REMIX, LSM-tree, and $B^+$-tree. The LSM-tree includes a Bloom filter with each sorted run. According to a well accepted production configuration [22], the Bloom filters are configured to use 10 bits per key to achieve a false positive rate of 1%. The experiments use a workload of a mix of about 75% existing and 25% non-existing keys. The point query only checks for the existence of the given key.

Figure 7 shows the results. For the LSM-tree, when the cache is too small to fit all Bloom filters, the Bloom filter approach uses many more I/Os per query than all other methods. As the cache size increases to 2% of the dataset size, where most Bloom filters could fit in the cache, the LSM-tree only takes about one I/O per query to load the target key from disk. In comparison, REMIX uses

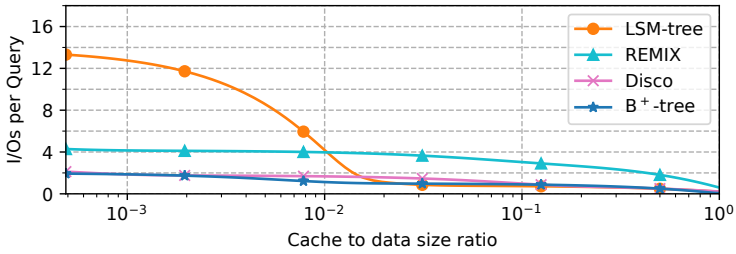



Fig. 7. Comparisons of numbers of I/Os per point query (probe) for LSM-tree, B$^+$-tree, REMIX, and Disco with different caching budgets. The LSM-tree, REMIX, and Disco have 8 sorted runs.

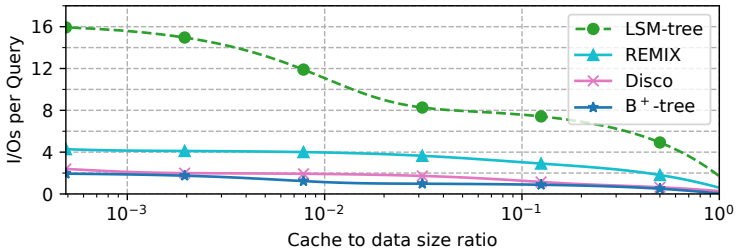



Fig. 8. Comparisons of numbers of I/Os per range query (seek) for LSM-tree, B$^+$-tree, REMIX, and Disco with different caching budgets. The LSM-tree, REMIX, and Disco have 8 sorted runs.

as many as 4 I/Os to finish a query. This is because REMIX has to issue I/Os to read and compare $\log_2 N$ keys, where $N$ is the number of keys in a segment. In our experiment setup where $N = 32$, REMIX is expected to perform at most 5 string comparisons to locate a key. Meanwhile, Disco and B$^+$-tree consistently maintain less than 2 I/Os per query across all cache sizes. When the cache is limited, Disco and B$^+$-tree both use 2 I/Os (or less) to finish a query. For Disco, these 2 I/Os include one access to the Disco metadata plus one access to the probe key at line 6 in Algorithm 2. The final key comparison results in a cache hit because the data block has been loaded into the cache when accessing the probe key. For the given dataset, the B$^+$-tree has height 2. For very small cache sizes, the 2 I/Os for B$^+$-tree can be explained by one I/O that accesses the internal nodes and another I/O that accesses the leaf nodes. For larger cache sizes, the internal nodes are typically in the cache, and B+ tree incurs one I/O per query.

We conclude from these results that Disco is effective in reducing the I/Os of point queries. Even though Disco has a structure similar to REMIX, we see that Disco consistently uses fewer I/Os to finish queries than REMIX across all cache budgets. The I/Os per query of Disco even approach that of the B$^+$-tree for parts of cache size range.

*Range query experiments.* In the second set of experiments, we look at how effective Disco is in minimizing range query I/Os. Similar to the point query experiments above, these experiments also use a workload of a mix of existing and non-existing keys. However, instead of reporting on the existence of the key, range queries find the smallest key that is greater than the key given by the user. This requirement renders the Bloom filters in LSM-tree ineffective. Similar to production LSM-tree based systems [21, 25], the LSM-tree in this experiment does a range query on each sorted run and uses a min-heap merging iterator to sort-merge the results on the fly.

Figure 8 shows the results of the range query experiment. Unsurprisingly, we see that the LSM-tree has the most I/Os per query across all memory budgets because it needs to separately search each individual run. Compared to Figure 7, we find the range query results of REMIX and B$^+$-tree largely match the point query results. This is expected because REMIX and B$^+$-tree do not have any specific optimizations on one type of the query and they process the point and range queries in the same way. We do see a slight increase in the I/Os per query for Disco. When the cache is limited, the I/Os per query for Disco is slightly higher than 2. In Figure 8, the gap between B$^+$-tree and Disco is larger than the gap in Figure 7. The difference is because Disco is able to skip I/Os to empty point queries when the partial keys do not match as described in Section 3.2.

In conclusion, these experiments show that Disco is effective in reducing the I/Os of range queries. When compared with REMIX, Disco only uses half of the I/Os to complete a query. When compared with LSM-tree, Disco reduces the number of I/Os per query by about 87.5%. The I/O saving can be observed across all cache budgets, and the saving is more prominent with a smaller cache.

### 4.2 DiscoDB Performance

In the following section, we evaluate the performance of DiscoDB and compare it against RemixDB, RocksDB as well as our two baseline adaptations from DiscoDB, namely Full-Index and No-Index, as described at the beginning of Section 4.

*Experimental setup.* Unless otherwise specified, the evaluations are run on CloudLab c220g2 instances. The system runs Ubuntu 22.04 LTS with Linux 5.15. It has two Intel Xeon E5-2630 CPUs and 128 GB of DRAM. The experiments run on an Intel S3500 480G SSD [30] that is formatted with an Ext4 file system.

In our experiments, we use 16-byte fixed-length keys, each having a 64-bit integer using hexadecimal encoding. We choose value size of 120 bytes to see how each system performs under the representative key-value size [19]. To simulate our target setup where a system has a different small memory budget, we use cgroups to limit the memory resource when running the experiments. This memory resource would be used by the memory component of the LSM-tree and the OS page cache. We use RocksDB `v8.10.0` configured according to the official tuning guide [22]. RemixDB is configured to have segment size $D = 32$, while DiscoDB uses 32-bit partial keys. All the experiments are done using only four client threads.

*4.2.1 Read performance.* This section shows how Disco helps minimize the query I/Os and improve the read performance of DiscoDB. To better understand how Disco helps in DiscoDB, we will be looking at the throughput of a uniformly random query workload and its average I/O per query. We then use a skewed query workload to see the performance of DiscoDB under more realistic settings. Besides the experiments on SSDs, we also study the effectiveness of Disco on a system with a faster (Optane) storage device where the I/O cost is low.

The experiments are loaded with about 1 billion key-value pairs that have 16-byte keys and 120-byte values in a randomly shuffled order. As a result, each system has about 128 GB of key-value data after loading. We then issue queries using the keys sampled from the key range using the selected distribution. We use cgroups to limit the memory to 16 GB, 32 GB, and 64 GB to simulate system setups that have limited cache resources. Before every experiment, we clear the page cache and warm it up by issuing a batch of uniformly random query until the throughput saturates. We measure the I/Os in kernel using eBPF.

*Uniformly Random Workloads.* Figures 9 and 10 show the throughput and I/O results of the range query experiments. The experiments use seek operations with the query keys selected uniformly

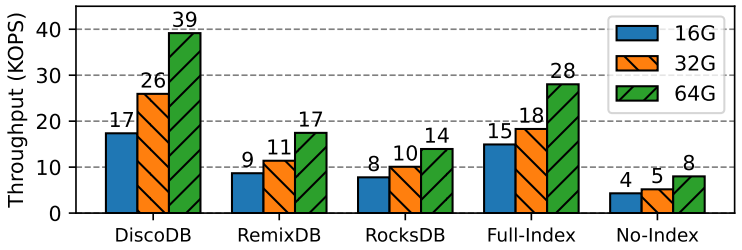



Fig. 9. Throughput of range queries (seek) with a uniform workload.

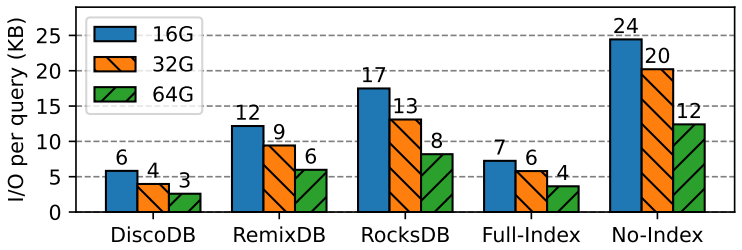



Fig. 10. Average I/O per query of range queries (seek) with a uniform workload.

at random. Figure 9 shows that the read throughput of DiscoDB is up to 2.3× and 2.8× as much as RemixDB and RocksDB across three different memory budgets. Having all the keys indexed by a $B^+$-tree, Full-Index achieves a throughput closer to DiscoDB. Full-Index has a lower throughput because of the larger size of the index, which results in more I/O on average.

Figure 10 shows the average I/O per query. It is calculated by dividing the total read I/O, measured in the unit of bytes, by the number of queries. This experiment is performed using only one worker thread to avoid overhead of synchronization in the eBPF program. Based on the I/O traces we collected, the I/Os issued by these systems are mostly in size of of 4 KB. The results in Figure 10 show that when the systems run with a 16 G memory budget, DiscoDB uses only about 1.45 I/Os per query, whereas RemixDB and RocksDB use about 3.0 and 4.38 I/Os per query, respectively. Full-Index achieves a result close to DiscoDB at an 1.8 I/Os per query under 16 G memory budget. The No-Index results show an average I/O cost of 6.1 I/Os per query, the highest among all the systems.

Combining the two Figures 9 and 10, we can also find that the throughput is roughly inversely proportional to the average I/O per query. This indicates that the I/O cost is the dominant cost in these systems. Hence, Disco is effective at improving the read throughput of DiscoDB in range queries by reducing the I/O it takes to finish queries.

*Point Queries.* Figure 11 shows the throughput of point query experiments. The experiments issue probe operations uniformly at random. The probe operations check for the existence of a given key. Different from the previous range query experiments, RocksDB can utilize the Bloom filters to filter out false queries. Full-Index and DiscoDB can also skip I/Os to the runs for empty queries. For this reason, RocksDB and DiscoDB have higher throughput than the range query experiments in Figure 10. RemixDB and No-Index cannot decide whether a key exists or not without scanning the runs. However, compared to seek, they can determine more quickly that a key does not exist in

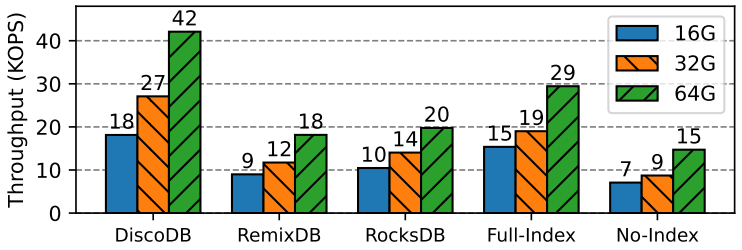



Fig. 11. Throughput of point queries (probe) with a uniform workload.

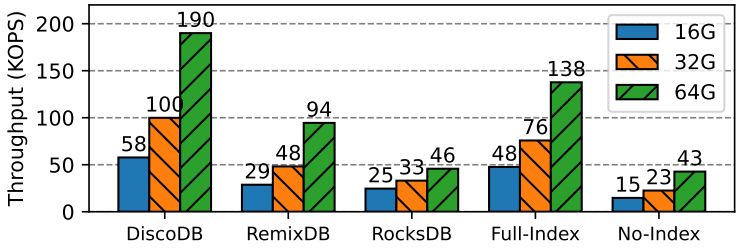



Fig. 12. Throughput of range queries (seek) with a skewed workload.

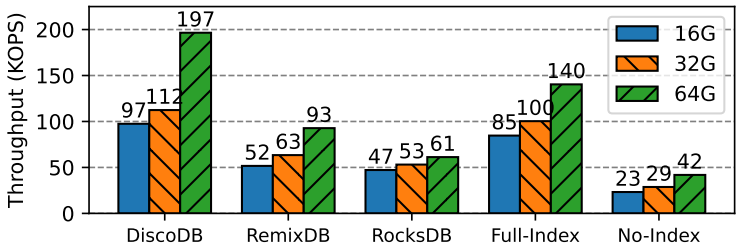



Fig. 13. Throughput of range queries (seek) with a uniform workload on an Optane drive.

a given run. To summarize, Disco's advantage in range queries also applies to point queries, albeit to a somewhat lesser extent.

*Skewed Workloads.* Figure 12 shows the results of range queries on a skewed query distribution that represents more realistic workloads [2, 9, 19, 24]. We use a composite-Zipfian distribution [24] that selects 1,000 hot spots from the key range uniformly at random then adds a Zipfian distribution using the YCSB standard $\theta = 0.99$. This distribution represents workloads that are less skewed than Zipfian. We run the skewed queries until the throughput is saturated.

Under this skewed workload, most of the hot entries would be cached in memory. The throughput is significantly higher compared to uniform workloads for all three systems. Comparing to Figure 9, the throughput of the skewed workload is more than 4× as much as that of the uniform workload. With 64 G memory budget, DiscoDB and RemixDB achieve up to 3.9× and 4.4× speedup over the uniform workloads, respectively. This shows that DiscoDB is less sensitive to the skewed workload than RemixDB. Nevertheless, the throughput of DiscoDB is higher than RemixDB and RocksDB achieving speedups up to 1.2× and 2.2×, respectively.

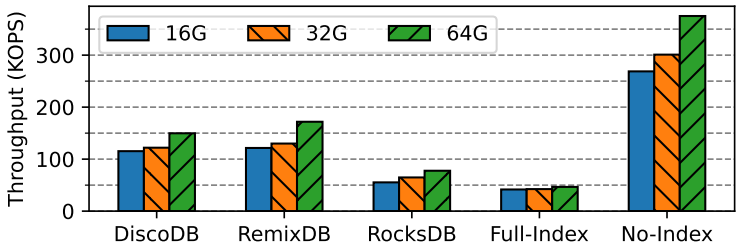



Fig. 14. Throughput of loading 1 billion key-value pairs in a shuffled order.

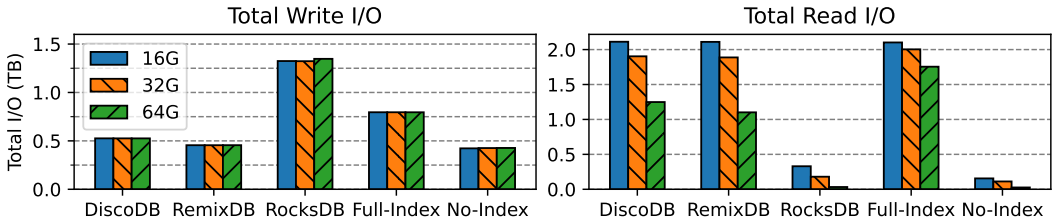



Fig. 15. Total write and read I/O of the loading experiment.

*Fast Storage Devices.* To see how the DiscoDB performs on systems where the I/O cost is low, we run our experiments on a system with a similar hardware configurations as the previous experiments except the storage device is an Intel Optane SSD 905P 960G [29]. Figure 13 shows the experiment results. The results show that DiscoDB still maintains the same improvements on the query throughput when compared to RemixDB and RocksDB.

*4.2.2 Write performance.* This section investigates the effect of Disco on the write performance of DiscoDB. In these experiments, we load each system with 1 billion key-value pairs in a shuffled order and measure the throughput and total read and write I/O issued during the experiments. Each key-value pair has 16-byte key and 120-byte value. As a result, each system has in a total of 128 GB of key-value pairs after loading. Similarly to the previous experiments, we limit the available memory for the systems to 16 GB, 32 GB, and 64 GB to see how they perform under different memory budgets. The experiment results are shown in Figures 14 and 15.

The results show that the write throughput of DiscoDB is 2% to 10% lower than RemixDB across different memory budgets. This is slowdown is mostly due to the need to construct partial keys for rebuilding Disco and the extra write to persist them in DiscoDB. Nonetheless, the write throughput of DiscoDB is still 1.9× to 2.1× as high as RocksDB in these experiments. Bearing the cost of maintaining Disco, loading in DiscoDB is about 50% slower than No-Index, where no indexes are maintained at all. When compared with Full-Index which organizes all keys in a B$^+$-tree, DiscoDB is 2× as fast in loading.

Figure 15 shows the write and read I/O used by these systems in the loading experiments. The write I/O results show that DiscoDB uses 16% more write I/O than RemixDB, whereas RocksDB write I/O is 2.7× and 3.2× to that of DiscoDB and RemixDB, respectively. Comparing to Full-Index, DiscoDB uses about 30% less write I/O due to the compact design of Disco. On the other hand, the read I/O results show that DiscoDB, RemixDB, and Full-Index use 5.4× more read I/O than RocksDB and No-Index with 16 G memory budget, largely explained by the need to maintain and rebuild the index. However, for DiscoDB the extra read I/O does not reduce the loading throughput.

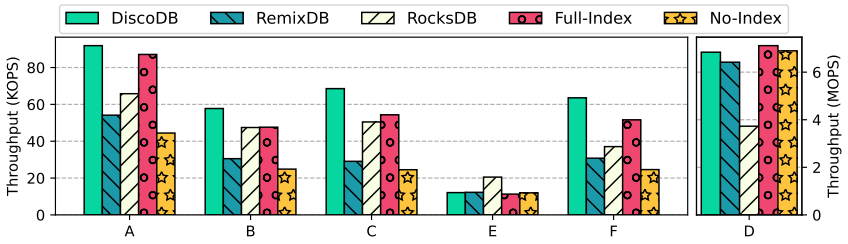



Fig. 16. YCSB results on SSD with 16 G memory limit.

In conclusion, DiscoDB has a competitive write performance versus RemixDB and still achieves a higher write throughput than RocksDB.

### 4.3 YCSB

YCSB [13] is a popular benchmark suite for key-value stores. It uses synthetic workloads that emulate real-world production workload patterns with different read-write ratios. We run the YCSB workloads on DiscoDB, RemixDB, RocksDB, Full-Index, and No-Index to see how they compare in more realistic settings. For each workload, we use the 128 GB stores generated from the previous loading experiments so the keys are 16-byte fixed-length keys and values are 120 bytes long. We run each workload for 200 seconds. The details of the YCSB workloads are shown in Table 2. Except for workload D which uses the latest query distribution, all other workloads use the composite-Zipfian distribution described in the previous section. Each workload is done using four client threads.

Table 2. YCSB workloads

| Workload | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| Operations | 50% U<br>50% R | 5% U<br>95% R | 100% R | 5% I<br>95% R | 5% I<br>95% S | 50% R<br>50% M |

*I: Insert; U: Update; R: Read; S: Seek-Scan next 50 keys; M: Read-Modify-Write.

*SSD.* We run the YCSB experiments on CloudLab instances with SSD storage, described in Section 4.2. Figure 16 shows the YCSB results of experiments with memory limited to 16 GB. The results show that DiscoDB outperforms RemixDB and RocksDB in workloads A, B, C, and F that mostly consist of read operations. Because workload D uses the latest query distribution that is skewed towards recently inserted items, most of the queries are served from the MemTable of the LSM-tree, so workload D is not of interest to our analysis of Disco. Here, DiscoDB, RemixDB, Full-Index, and No-Index achieve over 2× improvements than RocksDB largely because they all employ Wormhole [50] as the MemTable. In workload E, the throughput of the other systems is lower than RocksDB. This is because workload E scans the next 50 keys after a seek, most of the query cost is dominated by the I/O to scan the keys.

*Fast storage devices.* We repeat the same YCSB experiments on a similar system except using Intel Optane SSD as the storage device to see how the numbers change when the I/O cost is low. Figure 17 shows the experiment results. When compared with RemixDB and RocksDB, DiscoDB still has similar trends as the previous experiments on SSD in workloads A, B, C, and F. Comparing Figures 16 and 17, one noticeable difference is that RemixDB achieves higher throughput than RocksDB in these workloads. The benefits of fast range queries in RemixDB primarily apply to high

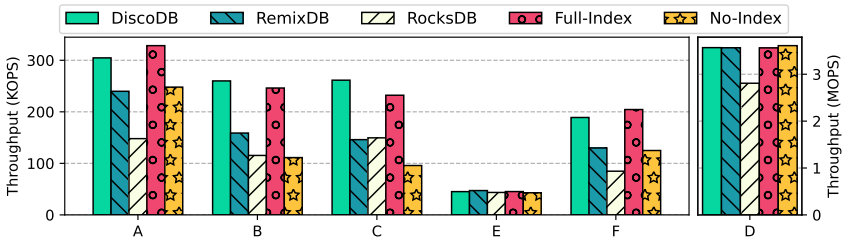



Fig. 17. YCSB results on an Optane drive with 16 G memory limit.

speed I/O systems where the CPU cost is more dominant. Evidence of this can also be seen in the similarity between RemixDB and No-Index performance. Another interesting result is the relatively strong performance of the Full-Index design, despite its higher memory consumption. These YCSB workloads have relatively short (16-byte) keys, which limits the advantage of compactness in Disco. At the same time, Disco does incur some extra overhead compared to full-index, particularly in extracting partial keys, offsetting the smaller compactness gains.

## 5 RELATED WORK

There exists a branch of research that uses filters to reduce the query I/Os on LSM-trees. Filters could be categorized into point filters [7, 16, 18, 52] and range filters [31, 36, 47, 51]. Filters improve query performance by skipping reading the files that the filter thinks the target key does not exist. Comparing to Disco's indexing approach, filters become less effective when the memory budget is limited. In addition, filters do not help locate keys. Extra I/Os are needed to actually locate the keys after filtering, whereas Disco directly pinpoints the target key positions.

Maplets [12] are a probabilistic hash index for LSM-trees. A Maplet maps a query key to a possible run in a level, eliminating the need to probe the filters at each run when performing queries. The Maplet approach is actually aligned with Disco. It avoids using filters and builds an index for a set of runs to map a key directly to a run. However, Maplets do not speed up range queries.

Much effort has been devoted to making B-trees and $B^+$-trees more compact and versatile [3, 10, 34, 42]. The idea of extracting discriminative bits for a group of keys has been discussed in various occasions [5, 23, 27]. The Height Optimized Trie [6] maximizes the fan-out of an in-memory trie by extracting discriminative bits in various lengths to adaptively build an internal node. To the best of our knowledge, Disco is the first to apply discriminative bits to an LSM-tree design that enables I/O efficient queries.

*Improving reads on LSM-trees.* Nova-LSM [28] indexes runs only on the first level of an LSM-tree. Jungle [1] combines LSM-trees and copy-on-write $B^+$-trees to improve the read performance at higher storage cost. Wisckey [35] separates the storage of keys from values to reduce the size of the LSM-tree and speed up reads. However, it introduces an extra layer of indirection and is not effective for small values. Bourbon [14] use learned models LSM-trees to accelerate reads, but it is limited to integer keys. Disco directly locates a result key for a query and supports variable-length string keys at a small storage cost.

*Reducing writes on $B^+$-trees.* $B^\epsilon$-tree [4, 8] allocates a buffer in each internal node to improve write performance on $B^+$-tree. Compared to Disco, $B^\epsilon$-tree still has mediocre read performance especially for range queries. In [41] the authors reduce the write amplification on $B^+$-tree using special hardware that support transparent compression. Disco maintains efficient writes on commodity hardware.

## 6 CONCLUSION

In this work, we presented Disco, a compact multi-run index for LSM-trees. Disco enables point queries and short range queries to locate keys using just 1–2 I/Os. Our experimental evaluation showed Disco reduces the I/O cost of queries by 51% of the state-of-the-art LSM-tree indexing techniques like REMIX. We integrated Disco into an LSM-tree database, DiscoDB, demonstrating throughput improvements of 150–220% across different workloads and memory budgets. Disco brings LSM-tree read performance much closer to $B^+$-trees, while retaining fast writes. The techniques presented provide a promising direction for developing LSM-tree indexes that are efficient for both reads and writes.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This material is based upon work supported by the National Science Foundation under Grants #2210616 and #2114218. Any opinions, findings, and conclusions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## REFERENCES


[1] Jung-Sang Ahn, Mohiuddin Abdul Qader, Woon-Hak Kang, Hieu Nguyen, Guogen Zhang, and Sami Ben-Romdhane. "Jungle: Towards Dynamically Adjustable Key-Value Store by Combining LSM-Tree and Copy-On-Write B+-Tree". In: *11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019, Renton, WA, USA, July 8-9, 2019*. 2019.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. "Workload analysis of a large-scale key-value store". In: *ACM SIGMETRICS / PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMATRICS'12)*. 2012, pp. 53–64.

[3] Rudolf Bayer and Karl Unterauer. "Prefix B-Trees". In: *ACM Trans. Database Syst.* 2.1 (1977), pp. 11–26.

[4] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. "An Introduction to B$\epsilon$-trees and Write-Optimization". In: *login Usenix Mag.* 40.5 (2015).

[5] Daniel J. Berstein. *Crit-bit trees*. URL: https://cr.yp.to/critbit.html.

[6] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. "HOT: A Height Optimized Trie Index for Main-Memory Database Systems". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 2018, pp. 521–534.

[7] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (1970), pp. 422–426.

[8] Gerth Stølting Brodal and Rolf Fagerberg. "Lower bounds for external memory dictionaries". In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*. 2003, pp. 546–554.

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

[10] Chen Chen, Wenshao Zhong, and Xingbo Wu. "Building an efficient key-value store in a flexible address space". In: *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. 2022, pp. 51–68.

[11] Félix Cloutier. *PEXT — Parallel Bits Extract*. URL: https://www.felixcloutier.com/x86/pext.

[12] Alex Conway, Martin Farach-Colton, and Rob Johnson. "SplinterDB and Maplets: Improving the Tradeoffs in Key-Value Store Compaction Policy". In: *Proc. ACM Manag. Data* 1.1 (2023), 46:1–46:27.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. 2010, pp. 143–154.

[14] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees". In: *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. 2020, pp. 155–171.

[15] Our World In Data. *Historical cost of computer memory and storage*. URL: https://ourworldindata.org/grapher/historical-cost-of-computer-memory-and-storage.

[16] Niv Dayan and Stratos Idreos. "Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 2018, pp. 505–520.

[17] Niv Dayan and Stratos Idreos. "The Log-Structured Merge-Bush & the Wacky Continuum". In: *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*. 2019, pp. 449–466.

[18] Niv Dayan and Moshe Twitto. "Chucky: A Succinct Cuckoo Filter for LSM-Tree". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 2021, pp. 365–378.

[19] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. "Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience". In: *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*. 2021, pp. 33–49.

[20] Facebook. *Index Block Format*. URL: https://github.com/facebook/rocksdb/wiki/Index-Block-Format.

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

[22] Facebook. *RocksDB Tuning Guide*. URL: https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide (visited on 04/01/2024).

[23] Michael L. Fredman and Dan E. Willard. "BLASTING through the Information Theoretic Barrier with FUSION TREES". In: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*. 1990, pp. 1–7.

[24] Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. "EvenDB: optimizing key-value storage for spatial locality". In: *Proceedings of the Fifteenth EuroSys Conference 2020 (EuroSys'20)*. 2020, 27:1–27:16.

[25] Google. *LevelDB*. URL: https://github.com/google/leveldb.

[26] Google. *leveldb File format*. URL: https://github.com/google/leveldb/blob/main/doc/table_format.md.

[27] Andy Goth. *critbit*. URL: https://wiki.tcl-lang.org/page/critbit.

[28] Haoyu Huang and Shahram Ghandeharizadeh. "Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 2021, pp. 749–763.

[29] Intel. *Intel Optane SSD 905P Series 960GB, 2.5in PCIe x4, 3D XPoint*. URL: https://www.intel.com/content/www/us/en/products/sku/147529/intel-optane-ssd-905p-series-960gb-2-5in-pcie-x4-3d-xpoint/specifications.html.

[30] Intel. *Intel Solid-State Drive DC S3500 Series*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-s3500-spec.pdf.

[31] Eric R. Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. "Proteus: A Self-Designing Range Filter". In: *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. 2022, pp. 1670–1684.

[32] Cockroach Labs. *Cockroach Labs, the company building CockroachDB*. URL: https://www.cockroachlabs.com/.

[33] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *Operating Systems Review* 44.2 (2010), pp. 35–40.

[34] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. "Tree Indexing on Solid State Drives". In: *Proc. VLDB Endow.* 3.1–2 (2010), pp. 1195–1206.

[35] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "WiscKey: Separating Keys from Values in SSD-conscious Storage". In: *14th USENIX Conference on File and Storage Technologies (FAST'16)*. 2016, pp. 133–148.

[36] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. "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 (SIGMOD'20)*. 2020, pp. 2071–2086.

[37] John C. McCallum. *SSD and Flash Memory Price Decreasing with Time*. URL: https://jcmit.net/flash2015.htm.

[38] Ju Hyoung Mun, Zichen Zhu, Aneesh Raman, and Manos Athanassoulis. "LSM-Trees Under (Memory) Pressure". In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022*. 2022, pp. 23–35.

[39] Arjun Narayan and Peter Mattis. *Why we built cockroachdb on top of rocksdb*. URL: https://cockroachlabs.com/blog/cockroachdb-on-rocksd/#fast-scans.

[40] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. "The Log-Structured Merge-Tree (LSM-Tree)". In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[41] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. "Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression". In: *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*. 2022, pp. 69–82.

[42] Ohad Rodeh. "B-Trees, Shadowing, and Clones". In: *ACM Trans. Storage* 3.4 (2008).

[43] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. "Constructing and Analyzing the LSM Compaction Design Space". In: *Proc. VLDB Endow.* 14.11 (2021), pp. 2216–2229.

[44] ScyllaDB. *ScyllaDB*. URL: https://github.com/scylladb/scylla.

[45] Facebook Open Source. *MyRocks | A RocksDB storage engine with MySQL*. URL: http://myrocks.io/.

[46] Chenlei Tang, Jiguang Wan, Zhi-hu Tan, and Guokuan Li. "Accelerating range queries of primary and secondary indices for key-value separation". In: *Proceedings of the 13th Symposium on Cloud Computing, SoCC 2022, San Francisco, California, November 7-11, 2022*. 2022, pp. 226–239.

[47] Kapil Vaidya, Tim Kraska, Subarna Chatterjee, Eric R. Knorr, Michael Mitzenmacher, and Stratos Idreos. "SNARF: A Learning-Enhanced Range Filter". In: *Proc. VLDB Endow.* 15.8 (2022), pp. 1632–1644.

[48] Hengrui Wang, Te Guo, Junzhao Yang, and Huanchen Zhang. "GRF: A Global Range Filter for LSM-Trees with Shape Encoding". In: *Proc. ACM Manag. Data* 2.3 (2024).

[49] Ziwei Wang, Zheng Zhong, Jiarui Guo, Yuhan Wu, Haoyu Li, Tong Yang, Yaofeng Tu, Huanchen Zhang, and Bin Cui. "REncoder: A Space-Time Efficient Range Filter with Local Encoder". In: *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. 2023, pp. 2036–2049.

[50] Xingbo Wu, Fan Ni, and Song Jiang. "Wormhole: A Fast Ordered Index for In-memory Data Management". In: *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*. 2019, 18:1–18:16.

[51] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. "SuRF: Practical Range Query Filtering with Fast Succinct Tries". In: *Proceedings of the 2018 International Conference on Management of Data, (SIGMOD'18)*. 2018, pp. 323–336.

[52] Yueming Zhang, Yongkun Li, Fan Guo, Cheng Li, and Yinlong Xu. "ElasticBF: Fine-grained and Elastic Bloom Filter Towards Efficient Read for LSM-tree-based KV Stores". In: *10th*

*USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Boston, MA, USA, July 9-10, 2018*. 2018.

[53] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. "REMIX: Efficient Range Query for LSM-trees". In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 2021, pp. 51–64.

[54] Zichen Zhu. "SHaMBa: Reducing Bloom Filter Overhead in LSM Trees". In: *Proceedings of the VLDB 2023 PhD Workshop co-located with the 49th International Conference on Very Large Data Bases (VLDB 2023), Vancouver, Canada, August 28, 2023*. Vol. 3452. 2023, pp. 17–20.

Received July 2024; revised September 2024; accepted November 2024