A Bloom Filter Hierarchy for Non-key Search in Key-Value Stores

Wojciech Macyna
Wroclaw University of Technology
Poland

Carlos Ordonez
University of Houston
USA

***Abstract*—Key-value stores are a well-established technology for big data management, with many leveraging the Log-Structured Merge (LSM) tree for its high write throughput and efficient primary key lookups. However, searching for non-key values in LSM trees is slow, as it typically requires scanning all LSM tree files. Secondary indexes are a common solution, but they typically require rebuilding the entire LSM tree and involve a challenging selection of indexing attribute(s). To overcome these limitations, we propose a Bloom filter hierarchy to accelerate searching for non-key values in LSM trees. In a nutshell, a Bloom filter is built for each data file in the LSM tree, and then a hierarchy (another tree) of these Bloom filters is created. Experiments show our new indexing mechanism outperforms existing LSM methods by 80% with a small space overhead.**

I. INTRODUCTION

Key-value stores, a class of NoSQL databases, have gained significant popularity in big data applications due to their scalability, simplicity, and flexibility. Nowadays, they are used in various areas such as social networks, sensor networks, and other big data applications. Some popular data stores include HBase [5], MongoDB [2], LevelDB [6], and Cassandra [7]. Key-value stores typically maintain data in a key-value format using a Log-Structured Merge (LSM) tree [10]. In general, LSM-based stores offer high write throughput and rapid lookups of primary keys. This capability, though, is insufficient to support query processing on non-key attributes.

The efficient secondary index management in key-value stores is a hot research topic (see [4], [8], [9], [11]). However, all exisiting approaches require reloading all stored entries and constructing a storage data structure containing entries in the form $e =< value, key >$. This process creates a secondary index, where entries are sorted by $value$.

A. Motivating Example

Consider a Twitter database, a representative example of a NoSQL system. Each tweet is represented as $e =< id, \{user, text\} >$, denoting that a $user$ posted a tweet with the content $text$ and assigned identifier id. Here, id serves as the key for entry e, but $user$ and $text$ are non-key values. In that case, searching a tweet e with the specific id is very easy and fast. Let us consider the following SQL query:

```
SELECT *  FROM e
WHERE user = 'Smith';
```

Unfortunately, finding all tweets of the $user$ "Smith" entails scanning the entire LSM tree, which is slow. Implementing a secondary index on the $user$ attribute would significantly improve query efficiency. On the other hand, a secondary index $e_u =< user, id >$ could significantly improve lookup performance for queries on $user$, but it would not accelerate queries with a predicate on $text$. To address this limitation, an additional secondary index $e_t =< text, id >$ would also need to be created.

B. Contributions

To address the non-key search limitation introduced above, we propose a Bloom filter hierarchy to accelerate the search of non-key values, which allows a more efficient scanning of LSM-tree stores with a small additional space overhead.

Our approach can be summarized as follows: whenever a new data file is created in the LSM-tree, a corresponding Bloom filter file is built. When evaluating a query searching for a specific value, the Bloom filter is checked first. If the filter indicates that the value may exist, then the corresponding data file is scanned.

We construct a Bloom filter hierarchy similar to a B+-tree. In this hierarchy, leaf nodes correspond to the Bloom filter files associated with the data files, whereas the intermediate nodes are created by performing a bitwise "or" operation on their child nodes. Since Bloom filters are bit arrays of a fixed length, the bitwise "or" operation on these bit arrays is straightforward to perform. This approach allows us to start the lookup at the root node, thereby avoiding unnecessary checks on many Bloom filter files and, consequently, reducing the need to scan numerous data files. We utilize a Bloom filter hierarchy as a mechanism to accelerate the search of non-key values. To validate our proposed indexing data structure, we implemented the proposed method and integrated it into LevelDB (see [6]), a widely-used key-value store. We chose LevelDB due to its popularity in commercial applications and ease of implementation in C++. We conducted preliminary experiments on large synthetic files to validate the effectiveness of the proposed method.

II. BACKGROUND

In this section, we outline the core concepts discussed in the paper. We begin by elucidating the data management principles within the LSM-tree store. Following this, we provide a brief

overview of the fundamental assumptions and algorithms underlying Bloom filters. Lastly, we offer a conceptual depiction of Bloom filter hierarchy, highlighting its organizational data structure and functionality.

A. Data Management with LSM Tree

The LSM tree [10] is a data structure used in many database systems. It is designed to efficiently manage data storage and retrieval, particularly in scenarios involving high write throughput and large volumes of data. The data, typically key-value entries, are stored in distinct files known as SSTables (Sorted String Tables). The size of each SSTable remains relatively constant, typically around a few megabytes. The LSM tree consists of k sorted levels (L_0 to L_k) (see Fig. 1). Each level encompasses a predetermined number of SSTables, resulting in a size limit specific to each level.

The LSM tree is organized into multiple levels, each containing numerous key-value entries sorted by key within that level. The top level is typically stored in RAM, while the other levels reside on disk. New entries are always inserted into the top level. When the number of entries in a level exceeds its predefined limit, that level is merged with the level below.

The **insert** operation proceeds as follows. Initially, key-value entry $e =< k, v >$ are written into L_0, which is referred to as the MemTable and resides in main memory. If L_0 exceeds the size limit, the data are flushed to the first on-disk level L_1. When L_1 exceeds its size limit, it is merged with L_2. This process involves fetching all entries from L_1 and L_2, sorting them by key k, and writing the sorted entries back to L_2. Notably, when the capacity of L_2 is surpassed, it is merged with L_3, and so forth. Consequently, the levels expand, resulting in L_{i+1} being significantly larger than L_i. The LSM tree prioritizes write operations, as updates are confined to the in-memory level L_0. The LSM tree's optimization for writes involves updating only the in-memory level L_0.

A **search** on key in the LSM-tree begins at L_0 and progresses through L_1, L_2, and so on, until the required entry is located. This ensures retrieval of the most recent version of the entry, meaning the latest updated version is found first. If the required entry has been deleted, the tombstone entry e_d is encountered first, and the search stops. Looking up a key in the LSM-tree is very fast. Since each level is sorted by key, efficient binary search can be applied. Hence, the search is performed in $O(\log N)$ time, where N is the number of entries in the LSM tree. Additionally, each SSTable contains a key range known as a zone map, which allows the required key to be quickly located in the appropriate SSTable. Then, the search is performed in $O(\log n)$ time, where n denotes the number of entries in the SSTable.

We now discuss less frequent, but still important, delete and update operations. The **delete** operation in the LSM-tree is performed as follows. When an entry $e =< k, v >$ is deleted, a tombstone entry $e_d =< k, null >$ is inserted into the MemTable (L_0). The tombstone is then propagated through the levels by the merge process until it reaches the level containing the original entry. At that point, both the tombstone and the original entry are removed from the LSM-tree.

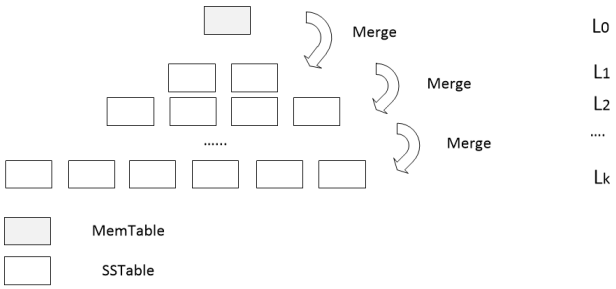

Fig. 1: *LSM Tree.*

An **update** is performed in a similar way. When the entry $e =< k, v >$ is updated in the LSM key-value storage, a new entry $e' =< k, v' >$ is inserted into the MemTable. In such cases, the "newest" entry always resides higher in the LSM-tree than the old one.

B. Bloom filter

A Bloom filter [1] is a space-efficient data structure to answer membership queries on a set of elements. Let S be a set of n elements: $S = \{s_1, s_2, ..., s_n\}$. A Bloom filter is an array of m bits initialized to 0. It uses r independent hash functions: $h_1, h_2, ..., h_r$ with range $\{1...m\}$. For each element $s_j \in S$, the bits $h_i(s_j)$ are set to 1 for $1 \leq i \leq r$. To check if $x \in S$, we compute $h_i(x)$, which takes time $O(1)$. The filter checks if all the positions generated by the hash functions $h_i(x)$ are set to 1. If at least one of them is set to 0, it means that $x \notin S$ (no false negatives). Otherwise, we assume that $x \in S$ although it may be wrong with a small probability. Such probability is called a false positive rate and it is estimated as: $p \approx \left(1 - e^{-\frac{r*n}{m}}\right)^r$. The probability is minimal when $r = \lceil m/n \log 2 \rceil$ and can be lowered by increasing the size m of the Bloom filter. Typical false positive rates for Bloom filters are very low, generally below 1% and even 0.1%, making them attractive for applications where a small probability of false positives is acceptable.

In our approach, a separate Bloom filter is created for each SSTable. The detailed description of this integration is presented in Section III.

C. Bloom filter hierarchy

The Bloom filter hierarchy (see [3]) operates on the principle of constructing a tree data structure (see Figure 2). In this tree, the leaves represent the individual Bloom filters associated with the SSTables, while the parent nodes are Bloom filters constructed by performing a bitwise "or" operation on their child nodes. This process continues bottom-up until reaching the root of the tree. Each bloom filter within the hierarchy should possess identical parameters, including the same number of bits and utilize identical hash functions. We define an order parameter d. Each non-leaf node maintains f child pointers, where $d \leq f \leq 2d$ for all non-root nodes, and

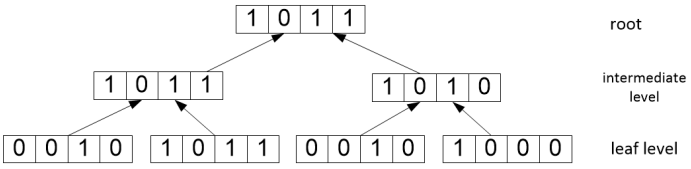

Fig. 2: *Bloom filter hierarchy (m=4, d=2).*

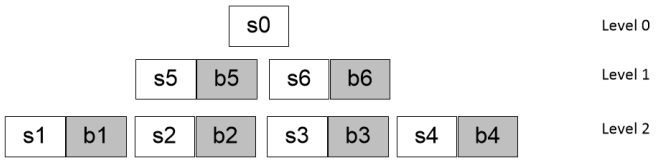

Fig. 3: *LSM-tree with Bloom filters.*

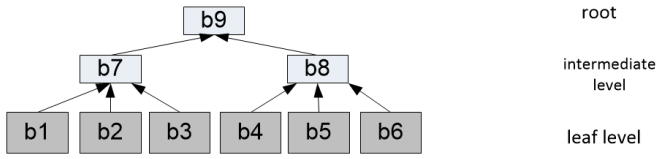

Fig. 4: *Bloom filter hierarchy for LSM-tree.*

$2 \leq f \leq 2d$ for the root. Figure 2 presents a Bloom filter hierarchy with four leaf nodes, two non-leaf nodes and one root. The size m of each Bloom filter is 4, and the order d is 2. The estimated depth of the Bloom filter hierarchy is $\log_d B$, where B is the number of Bloom filters at the leaf level. When the probability of false positives is low, the Bloom filter hierarchy provides $O(d \log_d B)$ search cost and $O(B)$ storage cost. The Bloom filter hierarchy can be modified either by inserting or deleting the Bloom filter at the leaf level. In both cases, the entire subhierarchy must be reconstructed.

The key property of this index is that each non-leaf Bloom filter in the tree represents the union of the sets represented by the Bloom filters in the subtree rooted at that node. Consequently, if an object matches a Bloom filter at the leaf level, it matches all Bloom filters along the path from that leaf to the root. Conversely, if a specific Bloom filter does not match an object, there is no match in the entire subtree rooted at that node. In this case, searching the Bloom filters of this subtree is not needed, which leads to pruning the search space. This property is highly powerful and serves as the core of the proposed method.

III. A Bloom filter hierarchy in the LSM tree

In this section we present our research contribution, assembling together the components defined in Section II.

A. Integration of the Bloom filter hierarchy with LSM tree

We first present the integration of the Bloom filter hierarchy with the LSM-tree stores.

Figure 3 shows the LSM-tree that consists of six SSTables (s_1 to s_6) and one MemTable (s_0). Each SSTable (s_i) is paired with its own distinct Bloom filter (b_i). As s_0 resides in the main memory, we do not create a Bloom filter for it. Checking the Bloom filter (b_i) makes it straightforward to determine if the requested value is present in SSTable (s_i). The Bloom filters (b_1 to b_6) paired with SSTables constitute the leaf level of the Bloom filter hierarchy (see Fig. 4). Bloom filters at the intermediate and root levels (b_7 to b_9) are generated by performing bitwise "or" operations on their respective children.

Figure 5 provides a more detailed view of an LSM-tree with the Bloom filter hierarchy shown in Figure 3. Each SSTable (s_1, s_2, s_3, s_4, s_5, and s_6) contains six key-value entries. The elements are sorted by the key within the level. Each SSTable (s_i) is paired with a corresponding Bloom filter (b_i).

B. Maintaining the Bloom filter hierarchy in the LSM tree

In this subsection, we outline the basic operations of an LSM-tree and their effect on the Bloom filter hierarchy.

LSM tree modifications. The LSM-tree can be modified through the insertion, updating, and deletion of key-value entries. All these operations are performed on s_0 at level 0 (typically in batches). Since s_0 resides in main memory and does not have an associated Bloom filter, these operations do not impact the Bloom filter hierarchy.

Level merging. Level merging is a fundamental operation in the LSM tree. During a merge operation, two consecutive LSM tree levels: L_i and L_{i+1} are combined to create a new level L_j. The algorithm works as follows.

- All key-value entries from L_i and L_{i+1} are fetched and sorted by the key.
- The sorted entries are inserted into SSTables at the new level L_j. For each SSTable in L_j a new Bloom filter is created.
- The Bloom filter hierarchy must be recreated.
- All SSTables and their corresponding Bloom filters in L_i and L_{i+1} are removed.

C. Searching a value in the LSM tree

During the search operation, the traversal of the Bloom filter hierarchy begins at the root. If the required value v matches

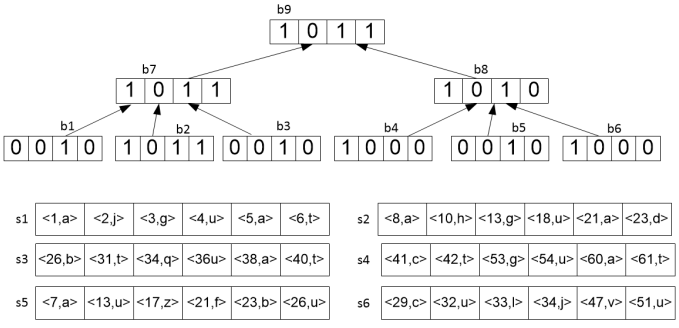

Fig. 5: *SSTable with the Bloom filter hierarchy.*

the Bloom filter at the root, all its child Bloom filters must be examined. Otherwise, none of the Bloom filters in the subtree can yield a match. Once the leaf level of the Bloom filter hierarchy is reached, the requested value v is searched in the related SSTable. This approach allows for scanning the Bloom filter hierarchy rather than traversing all SSTables or all Bloom filters at the leaf level.

For example, when searching for the entry with value q, we must obtain a positive result from b_3 because the required entry is in s_3 (see Fig. 5). Consequently, all the Bloom filters above it must also return positive results. We start the search from the root node of the Bloom filter hierarchy (b_9), which must return a positive result for value q. We then check b_7 and b_8. Since b_7 must return a positive result, we proceed to check b_1, b_2, and b_3. On the other hand, b_8 can return a negative result, in which case we do not need to check its child nodes. The leaf Bloom filter b_3 must return a positive result, prompting us to check s_3. If either b_1 or b_2 returns a positive result, we would also scan s_1 or s_2. In this case, a false positive result would occur. If we aim to locate a value that does not exist in the level, such as z, the Bloom filter in the root node typically returns a negative result. Consequently, rather than traversing all Bloom filters at the leaf level or all SSTables, we can promptly verify that the requested value does not exist.

Algorithm 1 finds a set of key-value entries in the Bloom filter hierarchy. The first part of the algorithm (lines 3 to 7) describes the recursive search through the non-leaf nodes. The second part of the algorithm (lines 8 to 11) checks for the existence of the value v in the leaf Bloom filter. If v is found, the corresponding SSTable is scanned.

Using this approach, we outline the method for scanning the LSM-tree (see Algorithm 2), which consists of three parts. The first part involves traversing the Bloom filter hierarchy, starting from the root (refer to Algorithm 1). Since the MemTable does not have an associated Bloom filter, it must be scanned afterward. Finally, the keys must be validated (see Algorithm 3). This validation algorithm ensures that the entry returned by the search is valid. Multiple versions of the same key-value entry may exist in the LSM-tree (see Section II-A). In such cases, the most recent version of the entry should be retrieved. If the value of this version differs from v, the entry should be excluded from the result set (lines 5 to 8). Additionally, if the entry has been deleted and an obsolete version remains in the result, that entry should also be discarded (lines 9 to 11).

IV. Experiments

This section outlines the experiments conducted. We implemented the Bloom filter hierarchy in C++ and integrated them with LevelDB [6]. LevelDB is a popular choice for persistence in many commercial systems due to its efficient key-value storage capabilities and reliability. Its usage in prominent systems such as Google Chrome's IndexedDB, Riak, and InfluxDB underscores its suitability for various applications, including web browsers, distributed databases, and time-series databases.

```
1  SearchInBloom(input: Value v, BloomFilter b, output:
     Key[] keys)
2  Let k[] denote the set of keys returned by the search
3  if b ∉ LeafLevel & check(b, v) = true then
4  |   foreach Bloom bc ∈ b.children do
5  |   |   SearchInBloom(v, bc)
6  |   end
7  end
8  if b ∈ LeafLevel & check(b, v) = true then
9  |   k:= SearchRelatedSSTable(v)
10 |   keys:=keys ∪ k
11 end
```

Algorithm 1: Search in Bloom Filter.

```
1  LSMSearch(input: Value v, output: Key[] keys)
2  Let root denote a root node in the Bloom filter
     hierarchy
3  Let keys[] denote the set of keys returned by the
     search
4  keys := ∅
5  SearchInBloom(v, root, keys)
6  k:= SearchMemTable(v)
7  keys:=keys ∪ k
8  keys:=ValidateKeys(v, keys)
9  return keys
```

Algorithm 2: Search in LSM-tree.

A. Setup

The experiments were conducted on a Linux server Intel® Core™ i9-9960X 3.10 GHz. The processor has 16 cores and 32 threads, and the cache lines configured as follows: L1 cache of 516KB, L2 cache of 22MB, and L3 cache of 22MB. The server also has a total of 132GB of RAM and SSD 1,7TB.

All experiments were programmed in C++ and compiled using GNU g++ 13. We utilized the LevelDB library from [6]. In our experiments, we assume that the Bloom filters at the leaf level are stored in persistent storage, while the intermediate levels of the Bloom filter hierarchy are maintained in RAM. Both the key and value of each entry in LevelDB are stored as strings.

```
1  ValidateKeys(input: Value v, output: Key[] keys)
2  Let keys[] denote the set of keys returned by the
     search
3  keys:= RemoveDuplicate(keys)
4  foreach Key k ∈ keys do
5  |   value:= GetValue(k)
6  |   if value ≠ v or k is deleted entry then
7  |   |   Remove k from keys
8  |   end
9  end
10 return keys
```

Algorithm 3: Keys validation.

TABLE I: Basic operation time in LevelDB without using Bloom filters (in seconds).

Insert operation number (N)	Insertion N key-value pairs	Lookup of one non-key value	Lookup of one key
10M	21	5	0.000243
50M	103	27	0.000088
100M	210	54	0.000257
500M	1141	281	0.000868

TABLE II: SSTable and Bloom Filter size.

Number of entries	10M	50M	100M	500M	1B
SSTable size (MB)	293	1200	2420	12500	25200
Leaf Bloom filter size (MB)	29	148	296	1540	3078
NonLeaf Bloom filter size (MB)	7	37	75	405	769

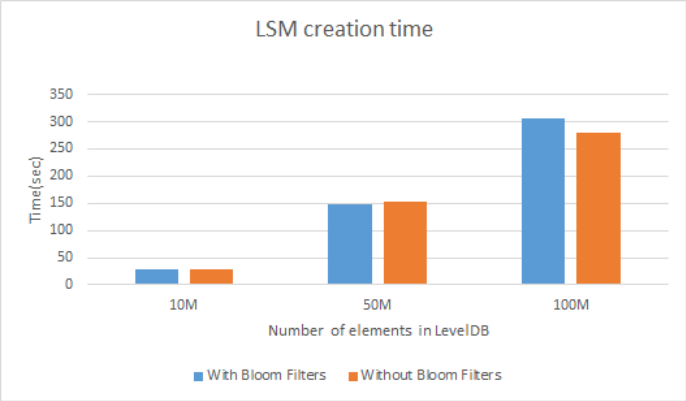

Fig. 6: *LSM Tree Creation Time* .

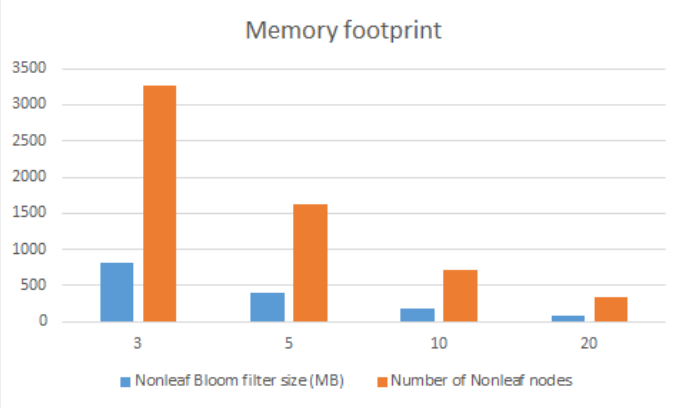

Fig. 7: *Memory footprint of non-leaf Bloom Filters for different order* d.

B. Preliminary experiments

Initially, we conducted some preliminary experiments. We created databases using standard LevelDB settings, which included a maximum of 7 LSM-tree levels and approximately 2MB SSTable size. Additionally, the size ratio, defined as the factor by which each level in the tree grows relative to the previous one, is set to 10. Hence, the database with 500 million key-value entries consists of 6434 SStables. All the experiments always fetched the data from the secondary storage. Table I illustrates the time required for the insertion of N key-value entries, as well as the time taken for key and $value$ lookup in LevelDB. Evidently, key lookup operations are exceptionally fast, whereas $value$ lookup operations exhibit a notably slower performance.

LevelDB utilizes a builder, where new key-value entries are inserted before the merge process. When the builder reaches its maximum file size, a new SSTable is created. We insert a value into the Bloom filter simultaneously with the insertion of the key-value entry into the builder. We measured the overhead of Bloom filter insertion time and found it to be very similar to the database insertion time (see Figure 6). Similarly, the cost of creating the Bloom filter hierarchy is negligible. It is worth noting that LevelDB performs its merge process in a separate thread, which can result in slight variations in the time taken for the LSM-tree creation.

Table II displays the sizes of SSTables, leaf Bloom filters, and non-leaf Bloom filters in megabytes for LSM trees containing varying numbers of items. We set the Bloom filter size to 2 million and used an order d equal to 5, meaning that each Bloom filter in the hierarchy, except for one, has exactly 5 child Bloom filters. The table shows that non-leaf Bloom filters occupy relatively little space, making them well-suited for storage in RAM.

The order d significantly impacts the memory footprint of non-leaf Bloom filters, as it determines the number of these filters within the hierarchy. When d is smaller, more non-leaf Bloom filters are required to cover all entries, which increases the overall memory usage (see Fig. 7). The experiment was carried out for 500M key-value entries. A Bloom filter size was fixed as 2M bits.

C. Bloom filter hierarchy

This subsection presents the experiments that validate the effectiveness of a Bloom filter hierarchy. In the first experiment (Fig. 8), we established four databases containing 10, 50, 100, and 500 million elements, respectively. Then, we measured time of finding the entry with the specific value in these databases. We compared three methods: SSTable scan, classical Bloom scan, and our proposed method (Bloom hierarchy scan). In the SSTable scan, all SSTables are traversed to check the requested value. For the classical Bloom scan, all leaf-level Bloom filters are scanned. If a Bloom filter returns a positive result, the associated SSTable is examined to find the requested entry. The Bloom hierarchy scan refers to the approach proposed in this paper. We observe that there is no discernible difference between the SSTable scan and classical Bloom scan methods. Conversely, the Bloom hierarchy scan method significantly outperforms the other methods. Worth mentioning is the fact that the false-positive rate of the Bloom filter is always below 0.1%.

In the second experiment (Fig. 9), we conducted a comparison using the same methods, but with different order d in the

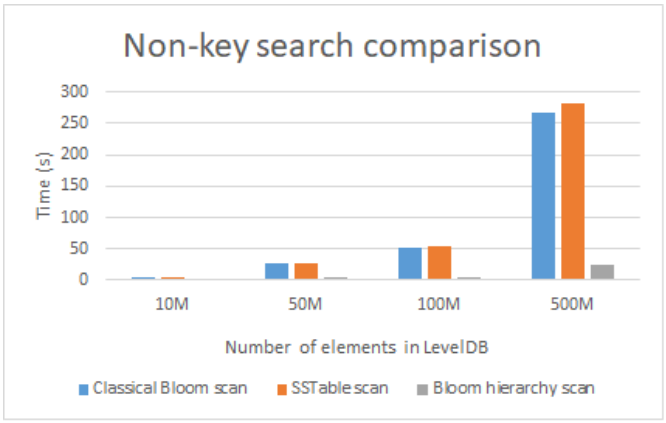

Fig. 8: *Impact of the number of elements in LevelDB.*

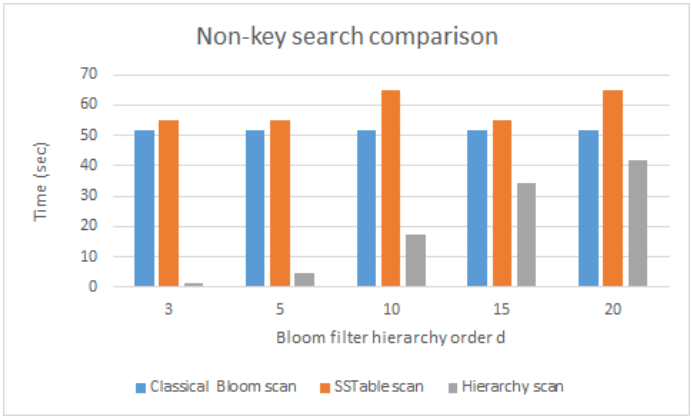

Fig. 9: *Impact of order d in the Bloom filter hierarchy.*

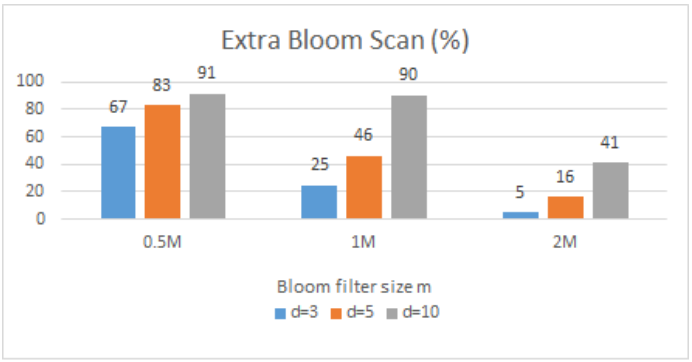

Fig. 10: *Impact of the Bloom filter size (m) and Bloom filter hierarchy order (d).*

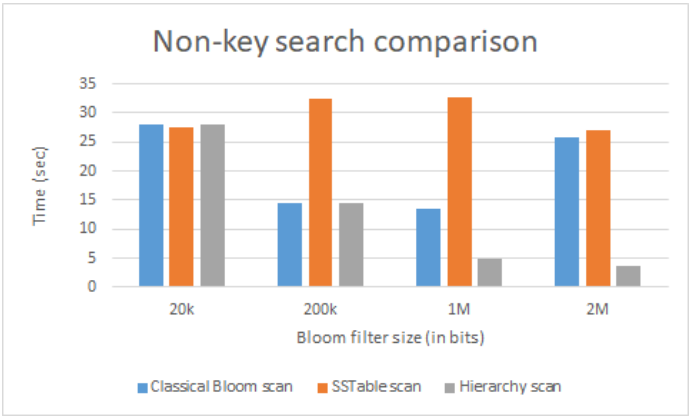

Fig. 11: *Impact of the Bloom filter size (m).*

Bloom filter hierarchy. We observe that increasing the order results in fewer Bloom filters being created in the intermediate level. Consequently, the intermediate Bloom filters have more "1" bits inside. This situation necessitates checking more Bloom filters during the search process. The experiment was carried out for 100M insert operations. A Bloom filter size was fixed as 500k.

In the next experiment, we investigated the impact of the Bloom filter size on the efficiency of hierarchy scan. We created three databases and inserted 10 million entries into each database, resulting in 119 database files. For each database, we varied the Bloom filter size m and the Bloom filter hierarchy order d. In this experiment, the requested entry was stored in one SSTable. The objective of this experiment is to demonstrate how the Bloom filter hierarchy facilitates the efficient skipping of irrelevant Bloom filters at the leaf level. Figure 10 illustrates that with a small Bloom filter size (m) and a high Bloom filter hierarchy order (d) a significant number of Bloom filters need to be checked. The optimal outcome is observed when $m = 2$ million and $d = 3$. In this case, only 6% of the Bloom filters are accessed instead of scanning all of them. According to our experiments, only one of these accessed filters yields a positive result. Consequently, only one SSTable needs to be scanned to find the required entry.

When the Bloom filter size is small, such as 20k, it tends to return many false positives. Consequently, the Bloom filters in the intermediate levels may not function accurately. As depicted in Fig. 11, when the Bloom filter size is small, the hierarchy scan does not outperform the other methods. This is because nearly all leaf Bloom filters must be accessed to find the requested value. An interesting case arises for a Bloom filter size of 200k. In this scenario, the Bloom scan performs better than the SSTable scan. This is attributed to the fact that the Bloom filter is more accurate and, simultaneously, relatively small. Therefore, the overhead for Bloom filter scanning is minimal. The best performance is achieved with a large and accurate Bloom filter, such as for a size of 2 million. The results are confirmed on Fig. 12. It illustrates the number of accessed leaf and non-leaf Bloom filters. When the Bloom filter size is small, a large number of Bloom filters are scanned. Conversely, when the Bloom filter is large, only two leaf Bloom filters and a few non-leaf Bloom filters are accessed.

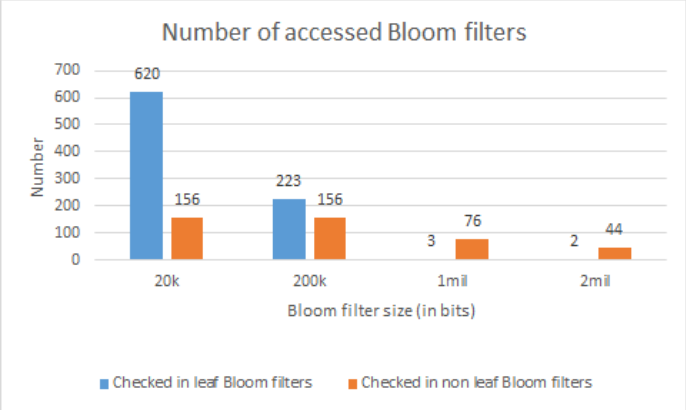

Fig. 12: *Impact of the Bloom filter size (m).*

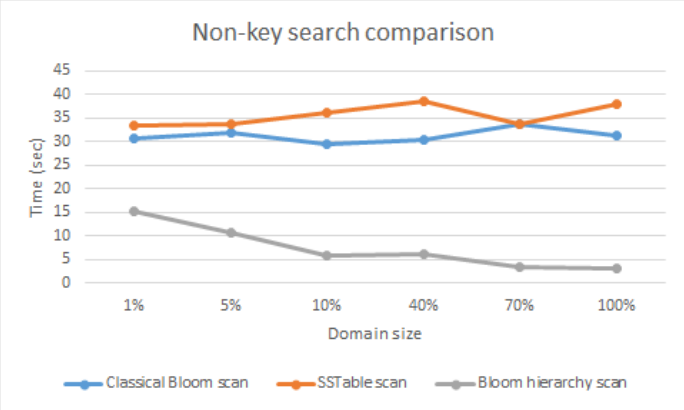

Fig. 13: *Impact of the value domain size.*

TABLE III: Number of accessed SSTables depending on the value domain range.

Value Domain Range (%)	Leaf Bloom Filters Found	SSTables Found	Leaf Bloom Filters Scanned
1	56	55	250
5	28	26	190
10	13	11	105
40	5	5	125
70	2	1	70
100	0	0	65

D. Different value domain range

In the last experiment, we insert 50 million entries into 599 SSTables. As mentioned earlier, each entry $e =< k, v >$ must have a unique key k but the value v does not have to be distinct. In the experiment, we compared the search mechanisms for different domain sizes (see Figure 13). By domain size, we mean the number of distinct values v that can exist in the entire entry set. For example, a domain size of 10% indicates a maximum of 5 million distinct values, while a domain size of 100% indicates that all inserted values are unique. Clearly, if the domain size is small (e.g., 1%), the required value v can be found in many key-value entries. We observe that the time required for the Classical Bloom scan and SSTable scan is very similar across all domain sizes. However, the performance of the Bloom hierarchy scan depends on the domain size. As the domain size decreases, the lookup time increases. This is because a smaller domain size results in many repeated values, leading to the scanning of multiple Bloom filters. In such cases, the method's efficiency is reduced. When the domain size approaches 0%, it becomes more advantageous to use a traditional scan method instead.

Table III confirms these observations. When the value domain size is small, many leaf Bloom filters must be scanned (column 4). Columns 2 and 3 present the number of Bloom filters that returned positive results and the number of SSTables where the required values are stored, respectively. As we can see, both numbers are similar, clearly indicating that the Bloom filter's false positive rate is very low.

E. Discussion

The proposed experiments confirm the effectiveness of our method. It is worth mentioning that the approach can seamlessly integrate with various LSM-tree optimizations. Many commercial LSM engines, such as RocksDB, use a tiered merge policy, compacting only portions of a level at a time. Our method assumes data is stored in raw files, with each compaction simply transferring data between files. This design allows the Bloom filter hierarchy to work with any compaction strategy.

V. Related work

The efficient secondary index management in key-value stores is a hot research topic. The authors in [4] discussed two indexing strategies for distributed key-value stores: one based on distributed tables, which leverages the underlying system's table model for index management, and the other using a co-location approach, which stores the secondary index on the same node as the corresponding base table records. Both strategies were implemented and integrated into HBase [5].

The paper [8] proposed a secondary index based on the LSM-tree. A key assumption of this approach is the existence of two LSM-trees. The first LSM-tree stores entries in the form $e =< key, value >$, while the second one holds the secondary index in the form $ind_e =< value, key >$. A required value can be quickly located, but afterward, the key must be searched in the main LSM-tree storage. However, this approach has some drawbacks. The primary issue with this method is the validation strategy. When the entry e changes, the corresponding index entries ind_e must also be updated. To address this problem, the authors considered eager and mutable-bitmap strategies.

The paper [9] proposed a partial secondary index in LSM-tree stores. Typically, all items in the database are equally covered by the secondary index. However, this approach is not effective in big data stores where some items are queried frequently while others are rarely, if ever, accessed. The key idea is to create a secondary index adaptively as a byproduct of query processing. Consequently, the database is indexed

partially, depending on the query workload.

A thorough comparison of secondary indexes in NoSQL databases was presented in [11]. The comprehensive experimental study and theoretical evaluation demonstrate that none of these indexing techniques outperforms the others in all aspects: embedded indexes offer superior write throughput and are more space-efficient, whereas stand-alone secondary indexes achieve faster query response times. Therefore, the optimal choice of secondary index depends on the application workload.

Bloom filters [1] are widely used to speed up key lookups in the primary index of an LSM tree. In LevelDB, each SSTable includes a Bloom filter to quickly determine whether a specific key exists within it. In contrast, Cassandra utilizes counting Bloom filters to keep track of the number of occurrences or manage deletions of elements within the SSTable [7].

Zone maps are another indexing technique often used with LSM-trees to enhance query efficiency by reducing unnecessary data scans [11]. Unlike Bloom filters, zone maps store metadata about value ranges within a data block (e.g., minimum and maximum values), enabling the system to bypass irrelevant blocks for a given query. This is particularly effective for accelerating key-based lookups, as entries are ordered by key. Although zone maps can be less effective as secondary indexes due to potentially large value ranges in SSTables, they can complement hierarchical Bloom filters.

A common challenge with secondary indexing is the selection of appropriate columns for indexing. Wide-column databases such as Cassandra and HBase enable storing column families together in the secondary storage, allowing developers to add elements to new columns without affecting existing ones or their data. In such cases, imposing a secondary index on a specific column becomes impractical. The utilization of the Bloom filter hierarchy provides a straightforward solution. Only one Bloom filter is required for each database file, accommodating storage for all column data. When the Bloom filter confirms the presence of data in the file, the file is scanned to determine which column the data belongs to.

VI. Conclusions and future research

In this paper, we proposed a new approach for efficiently retrieving key-value entries based on the ”value” in the LSM-tree. By employing the Bloom filter hierarchy, we avoided scanning all database files and were able to retrieve the file containing the required entries in logarithmic time. Our method outperforms existing LSM-tree methods by 80%. The approach has some limitations. Bloom filters only support point queries, meaning they can efficiently determine whether a particular item exists in the dataset, but they cannot handle range queries or other more complex search functionalities. Moreover, due to the use of hash functions, only exact comparisons are allowed, making it impossible to search for prefix values like ”W*”.

In future work, we will explore the use of Bloom filter hierarchies in the column family LSM stores. We will expand the underlying mathematical theory and focus on parallel processing with different query workloads.

References

[1] Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (Jul 1970). https://doi.org/10.1145/362686.362692, http://doi.acm.org/10.1145/362686.362692

[2] Chodorow, K., Dirolf, M.: MongoDB - The Definitive Guide: Powerful and Scalable Data Storage. O'Reilly (2010), http://www.oreilly.de/catalog/9781449381561/index.html

[3] Crainiceanu, A., Lemire, D.: Bloofi: Multidimensional bloom filters. Inf. Syst. **54**, 311–324 (2015). https://doi.org/10.1016/J.IS.2015.01.002, https://doi.org/10.1016/j.is.2015.01.002

[4] D'silva, J.V., Ruiz-Carrillo, R., Yu, C., Ahmad, M.Y., Kemme, B.: Secondary indexing techniques for key-value stores: Two rings to rule them all. In: Ioannidis, Y.E., Stoyanovich, J., Orsi, G. (eds.) Proceedings of the Workshops of the EDBT/ICDT 2017 Joint Conference (EDBT/ICDT 2017), Venice, Italy, March 21-24, 2017. CEUR Workshop Proceedings, vol. 1810. CEUR-WS.org (2017), https://ceur-ws.org/Vol-1810/DOLAP_paper_10.pdf

[5] George, L.: HBase: The Definitive Guide. O'Reilly Media, 1 edn. (2011), http://www.amazon.de/HBase-Definitive-Guide-Lars-George/dp/1449396100/ref=sr_1_1?ie=UTF8&qid=1317281653&sr=8-1

[6] Google: Leveldb. https://github.com/google/leveldb

[7] Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010). https://doi.org/10.1145/1773912.1773922, https://doi.org/10.1145/1773912.1773922

[8] Luo, C., Carey, M.J.: Efficient data ingestion and query processing for lsm-based storage systems. Proc. VLDB Endow. **12**(5), 531–543 (2019). https://doi.org/10.14778/3303753.3303759, http://www.vldb.org/pvldb/vol12/p531-luo.pdf

[9] Macyna, W., Kukowski, M., Zwarzko, M.: Multi-core adaptive merging of the secondary index for lsm-based stores. In: Database and Expert Systems Applications - 34th International Conference, DEXA 2023, Penang, Malaysia, August 28-30, 2023, Proceedings, Part II. pp. 245–257. Springer (2023). https://doi.org/10.1007/978-3-031-39821-6_20, https://doi.org/10.1007/978-3-031-39821-6_20

[10] O'Neil, P.E., Cheng, E., Gawlick, D., O'Neil, E.J.: The Log-Structured Merge-Tree (LSM-Tree). Acta Informatica **33**(4), 351–385 (1996). https://doi.org/10.1007/s002360050048, https://doi.org/10.1007/s002360050048

[11] Qader, M.A., Cheng, S., Hristidis, V.: A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018. pp. 551–566. ACM (2018). https://doi.org/10.1145/3183713.3196900, https://doi.org/10.1145/3183713.3196900