RoBF: An Auto-Tuning Bloom Filter for Mixed Queries on LSM-tree

Ruicheng Liu, Peiquan Jin, Shouhong Wan, Bei Hua
[1]*School of Computer Science and Technology, University of Science and Technology of China, Hefei, China*
[2]*Key Lab. of Electromagnetic Space Information, Chinese Academy of Sciences, Hefei, China*
jpq@ustc.edu.com

***Abstract*—Bloom filter is an efficient technique to improve query performance in LSM-tree-based databases, such as RocksDB, HBase, and Cassandra. However, the original Bloom filter uses a fixed false positive rate (FPR), which makes it inefficient for mixed queries that involve both point and range queries. To solve this problem, in this paper, we present an improved Bloom filter called RoBF (Range-Query-Oriented Bloom Filter), which uses a mixture of Bloom filters and can process mixed queries on LSM-tree efficiently. We design an efficient algorithm for generating the solution based on the query distribution. We compare our proposal with the trie-based filter and find out that each has its own advantages for various scenarios. Therefore, we propose to use different filters with varied sizes for different levels on LSM-tree. Following this idea, we present an algorithm to generate specific filters with a specific size for different levels on LSM-tree to optimize the performance of mixed queries under limited memory space. We conduct comparative experiments and compare the proposed RoBF with various competitors, and the results show that RoBF can improve the performance of evaluating mixed queries by up to 6x to 30x, compared to the original Bloom filter in RocksDB.**

***Keywords*—Mixed query, Bloom filter, LSM-tree**

I. Introduction

LSM-tree [1] has been widely used in many key-value data stores due to its high write performance. There are many database based on LSM-tree, for example, Cassandra [2], HBase [3], and Rocksdb [4]. One of the main challenges for such databases is to avoid query performance degradation caused by the multi-level write buffers of LSM-tree [5].

To accelerate the query performance of LSM-tree, many researchers proposed to add Bloom filters into the LSM-tree structure. Bloom filter is an effective scheme with only false positive errors and no false negative errors, which is ideal for reducing the query amplification in the LSM-tree [6]. Presently, Bloom filters have been used in many index structures to accelerate query performance, such as BloomTree [7].

However, although Bloom filters can improve query performance for point queries that aim to retrieve individual key-value pairs, they are not efficient for evaluating range queries. So far, researchers have proposed the Prefix Bloom filters [8] to record a fixed-length prefix for each key to respond to range queries. For example, if we want to know if an SST (Sorted String Table) contains a key between [*Hello, Henry*], we can simply return false if we are certain that the SST does not contain a string that begins with "He". However, the PBF scheme will lower the accuracy of point queries.

In this paper, we study the limitations of Bloom filters in handling mixed queries and propose a new solution. Briefly, the contributions of this study are four-fold.

(1) First, we found that using both of these filters can produce better results for mixed queries, and we call this filtering scheme DBF (Double Bloom Filter). Thus, we propose an algorithm to dynamically determine the parameters of the DBF, including the length of the prefix and the memory usage ratio of the two filters, which achieve ideal results on a particular query distribution.

(2) Second, to achieve high performance on a more general query distribution, we propose RoBF (Range-Query-Oriented Bloom Filter), which extends the number of prefix bloom filters from one to many. We extend the previous algorithm to be used to determine the parameters of RoBF. Similar to DBF, RoBF supports both point and range queries, and in most scenarios comprehensive queries are nearly twice as good as DBF. In comparison with SuRF, a trie-based hybrid query filter, RoBF has a significant performance advantage in small memory.

(3) Third, we found that for the same LSM-tree and the same query distribution, different levels of SST actually have completely different data and query distribution, meaning that different filters with different sizes should be applied to the data of different levels. Based on this idea, we propose an algorithm to determine the filter parameters for each level, including filter types and memory consumption.

(4) Finally, we conduct comparative experiments on a 100GB dataset and compare the proposed RoBF with various competitors, and the results show that RoBF can improve the performance of evaluating mixed queries by up to 6x to 30x, compared with the original Bloom filter of RocksDB.

II. Related Work

There are several optimizations for range queries for Bloom filters, such as the Prefix Bloom Filter (PBF) [8]. Compared with Bloom Filter, PBF uses a fixed-length key prefix to query keys, reducing the performance of point queries in exchange for support for range query filtering. PBF has long existed in RocksDB as an experimental function, and has officially become one of the main functions of RocksDB in recent years.

DOI reference number: 10.18293/SEKE2021-142

Unlike Bloom filters, PBF hashes each prefix of key length L in the set, rather than the entire key. PBF queries the prefix of target key length L in the point query, which reduces the accuracy of the point query to some extent. However, PBF fails to support prefix queries with the prefix length greater than L.

We compare the performance of RoBF with that of PBF in our experiments, and find that in most cases, the filtering performance of PBF is at least half of that of RoBF, and in many cases even close to that of RoBF. Because PBF has fewer parameters and is easy for database administrators to adjust, we think it is a useful filter.

SuRF is a recently proposed filter that employs fast succinct tries [9]. SuRF is a trie-based filter whose basic idea is to store the exponentially expanded part of trie and other parts separately to achieve higher compression rate. SuRF has the highest performance of trie-based filters that I know of. We regard SuRF as the representative of trie-based filters. Therefore, we often compare RoBF and SuRF in experiments.

Trie-based filters generally save prefix information with length not less than y, that is, for every key in S, the prefix with key length L will be fully recorded in the trie-based filter. In this way, the filter will always correctly answer interval queries $[a,b]$ if neither a nor b is longer than L; Even if the length of a or b exceeds L, the filter can be replaced with the result of $[a',b']$, where a' and b' are prefixes for the length of a and b not exceeding L, respectively.

The disadvantages of this filter are twofold. On the one hand, if the filter wants to answer a prefix query with length L, it needs to keep the L bits before each key completely, which sometimes consumes unnecessary space. On the other hand, trie-based filter performance is very sensitive to space, and its accuracy will be low if the space does not reach a certain threshold. In general, trie-based filters perform well when space exceeds a threshold, but not all scenarios tolerate such a high memory consumption filter.

Monkey [10] is another Optimal navigable key-value store. Monkey discusses the need to apply memory filters of different sizes to different levels. According to the characteristics of Bloom Filter, Monkey provides the optimal solution for point-only query scenarios. We extend this idea to the mixed query scenario and select different filters for different levels to optimize the mixed query performance.

III. Design of RoBF

A. Motivation

We have mentioned the design of two filters. The first one is Hash-based filters, such as PBF [8], which record the hash information of prefixes to support prefix queries with length no less than L. This kind of filters is designed for small memory scenarios. Another one is trie-based filters, such as SuRF [9], which record the complete information of prefixes to support range queries for large memory scenarios.

We show the differences between the two filters in Fig. 1(a), which shows the distribution of mixed queries consisting of three types of prefix queries and point queries. Each horizontal partition represents a prefix query or a point query, and the horizontal width represents its percentage in the query. In a partition, the top half represents the hit rate of the query, and the bottom half represents the miss rate, which means that a good filter always covers as much of the lower part of the figure as possible. Figure 1(f) shows a trie-based filter that can accurately answers queries with a prefix length less than L, but it has an error rate for queries with a prefix length greater than L. Figure 1(b) shows a hash-based filter that can answer a prefix query with length greater than L, but it cannot answer a prefix query with length less than L.

Note that the range query filter may have three types of errors:

(1) Trie-based filter will generate false positives when answering queries with length greater than L, but the frequency of such false positives will decrease with the increase of memory;
(2) Hash-based filters generate false positives due to hash conflicts.The frequency of such false positives also decreases as memory increases;
(3) Hash-based filters generate false positives when replacing interval queries with prefix queries.Such false positives are inherent and cannot be corrected by adding more memory. This type of errors cause hash-based filters to encounter performance bottlenecks when memory is large enough.

B. Data Structure

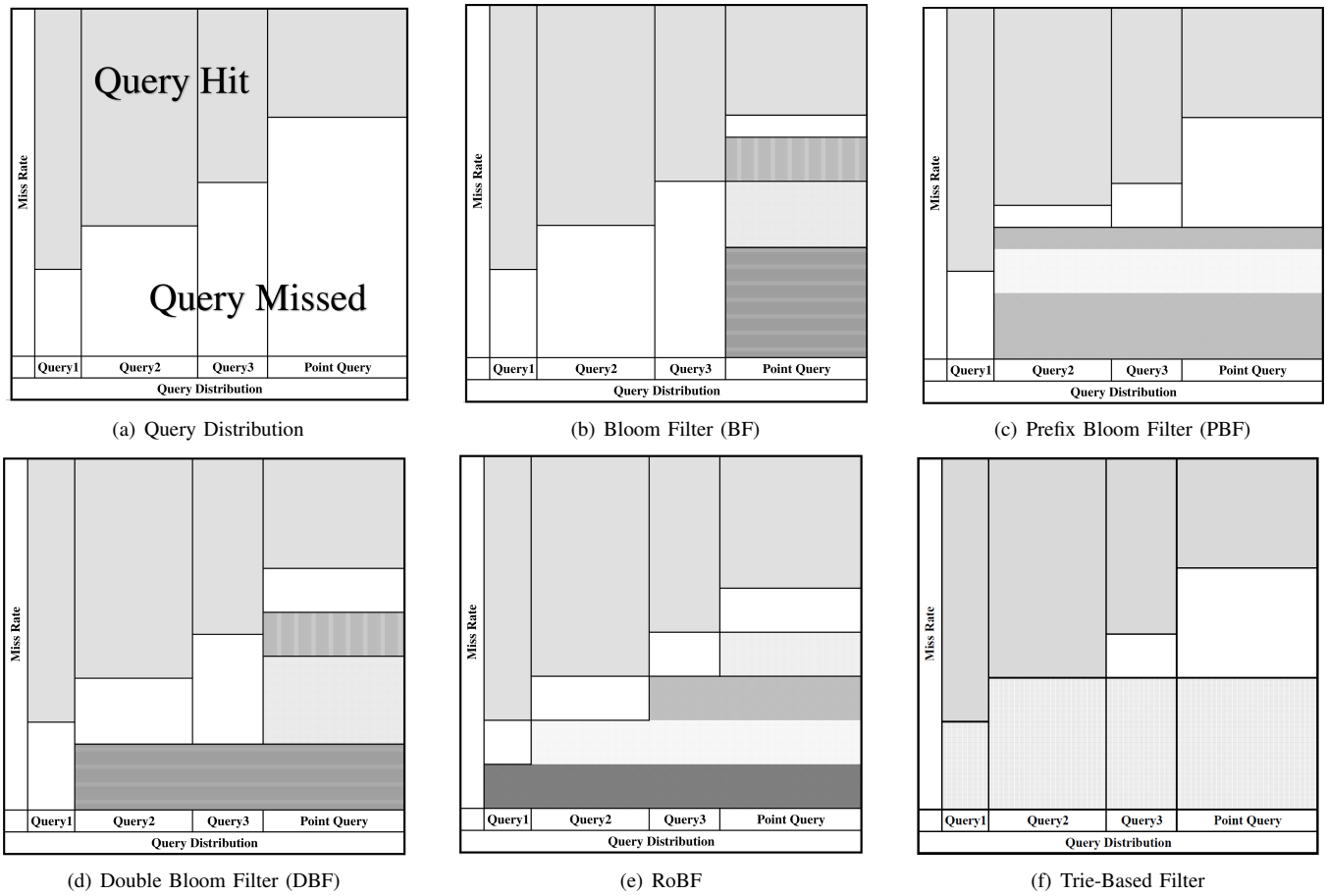
Before introducing RoBF, we first explain the working process of of hash-based filters.

Figure 1(c) shows how PBF works. It can only respond to point queries and prefix queries with length L or greater. Figure 1(d) shows how a DBF works, which is essentially a hybrid filter consisting of a PBF and a BF. Range queries need to be tested by PBF, while point queries need to pass both PBF and BF tests. RoBF is an improved version of DBF that contains multiple PBFs and one BF. Figure 1(e) shows how RoBF works. RoBF can be regarded as a number of prefix hash filters with different lengths, each of which has a prefix length ranging from 1 to a threshold *Lmax*. Here, *Lmax* represents the upper bound of the key length. Similar to Bloom filters, each test uses a different hash function, and all test results are stored in the same hash table.

In general, BF is more suitable for scenes with a large proportion of point queries, and PBF is more suitable for scenes with a fixed width of range queries. In addition, DBF can be dynamically adjusted according to the proportion of point queries and range queries. RoBF needs to decide the filter parameters according to the query distribution information. Generally, BF, PBF, and DBF can be considered as a special case of RoBF. In practical applications, we need to choose a right filter according to the load characteristics and determine the parameters of the filter.

C. Algorithm of Generating RoBF

In this section, we present the algorithm of generating RpBF, which is named *RoBF-generator*. Basically, the opti-



(a) Query Distribution (b) Bloom Filter (BF) (c) Prefix Bloom Filter (PBF)

(d) Double Bloom Filter (DBF) (e) RoBF (f) Trie-Based Filter

Fig. 1. **Performance of different filters** - Each column represents one type of query. The width of each column represents the weight of the query. The height of the gray rectangle represents the percentage of queries that are hit, and the height of the white rectangle represents the percentage of queries that are missed. The shaded portion represents the filtered query. Each shaded section, except for Figure 1(f), represents a hash test.

Filter Name	Hash Test						
Bloom Filter	L	L	L	L	L	L	L
Prefix Bloom Filter	13	13	13	13	13	13	13
Double Bloom Filter	13	13	13	13	L	L	L
RoBF	11	13	13	14	L	L	L

Fig. 2. **Four kinds of hash-based filters with seven hash tests**. Each grid represents a test, the number represents the prefix length of the test, and L represents the length of the key. For example, DBF contains four hash tests for prefixes with length 13 and three hash tests for the full key.

mization of filters means that there are as few false positives as possible, that is, the lower half of Fig. 1(e) should be covered as much as possible. For this purpose, we need to not only understand the distribution of the query, but also collect the characteristics of the dataset. For the sake of efficiency, we always assume that for any queries with prefixes whose length is p, the total number of the prefixes with length p is always close to the total number of the keys. Such an assumption allows us to make a general filter parameter recommendation without knowing the key distribution for each different SST.

We first use a simplified version of RoBF-generator to set the parameters of the DBF. When we try to set length p as the prefix length for DBF, we need to determine how much memory PBF and BF use. Here, the theoretical FPR is a convex function relative to the memory usage of a filter, and this conclusion can be used to speed up the search speed of the algorithm.

Furthermore, in the LSM-tree, each write buffer level actually has a separate distribution of query requests, even though the global query is the same for each level. For example, in the LSM-tree, it is likely that all keys in an SST of the underlying buffer share the same 10-byte prefix, which means that prefix queries with length less than 10 will be filtered out during range checking. However, this case will not happen in the upper buffer.

D. BPK-Balancer Algorithm

We present a new algorithm called BPK-balancer to search filter parameter combinations for each write buffer level for finding the best parameter balance. The term *BPK* is the abbreviation of BitsPerKey. At the beginning of the algorithm, we provide an optimal solution to accelerate the search speed, and this solution is based on the conclusion in Monkey [10].

Algorithm 1: RoBFGenerator (Current, Tests, Filtered)

```
   input : Queries
   output: BestSolution
1  Function Search(Current, Tests, Filtered):
2      if Current = Length then
3          X = Solution.Set(Queries, Current, Tests);
4          if Solution > BestSolution then
5              BestSolution = Solution;
6          end
7          return X;
8      end
9      MaxCovered = 1;
10     TernarySearcher.Set(0, Tests);
11     for i ∈ TernarySearcher do
12         X = Solution.Set(Current, i);
13         Covered = X * Missed.Sum(Current,Length);
14         Rest = Search(Current+1, Test-i,
             Filtered+X);
15         Covered = Covered + Rest;
16         TernarySearcher.AddSolve(i, Covered);
17         if Covered > MaxCovered then
18             MaxCovered = Covered;
19         end
20     end
21     return MaxCovered;
```

Algorithm 2: BPK-Balancer (Current, Bits)

```
   input : BitsPerKey
   output: BestSolution
1  Function WaitForTest(Level, Bits):
2      if
         FilterRecord.Lookup(Level, Bits) = False
         OR FilterRecord.Time(Level.Bits) >
         TimeLimit then
3          for type ∈ FilterSet do
4              SetFilter(Level, type, Bits);
5              FPR = RealWaitForTest(Level);
6              FilterRecord.Update(Level, Bits, FPR)
7          end
8      end
9      return FilterRecord.Latest(Level, Bits)
10 Function Balancer(Level, Bits):
11     if Current = 0 then
12         X = Solution.Set(Current, Bits);
13         if Solution > BestSolution then
14             BestSolution = Solution;
15         end
16         return X
17     end
18     MinFPR = 1;
19     TernarySearcher.Set(0, Bits);
20     for i ∈ TernarySearcher do
21         X = Solution.Set(Current, i);
22         FPR = WaitForTest(Current, i);
23         Rest = Balancer(Current − 1, Bits − i);
24         TernarySearcher.AddSolve(i, FPR +
             Rest);
25         if FPR + Rest < MinFPR then
26             MinFPR = FPR + Rest;
27         end
28     end
29     return MinFPR
```

IV. Performance Evaluation

A. Workload

The workload we used in the experiment is similar to the workload in YCSB [11], except that the queries in our workload include half point queries and half range queries.

The maximum interval length of the range query is set t0 100 by default, and the interval length distribution is uniform. The keys used in the database are strings with length 16, and the values of the keys are set to no more than 10^{14} positive integers. All the keys follow a uniform distribution. In addition, we set the size of the LSM-tree to contain 3^9 key-value pairs, with a total key size of about 50GB.

In the test of a single SST, we set the hit rate of point query and range query at about 10%. In the test of the LSM-tree, we set the global point query hit rate and range query hit rate at about 10%. When testing a single SST, we set the BPK (BitsPerKey) to 24, which is three times the default value, because it is convenient for us to compare the characteristics of RoBF and SuRF. For the global testing, we set the BPK to 8-16, which is 1-2 times the default value.

B. Filters Compared

We mainly compare our RoBF with the following filters in the experiment.

- *Prefix Bloom Filter (PBF)* [8]. We will not use the basic Bloom Filter as a comparison, because our test contains a large number of range queries, and we will use PBF instead.
- *Double Bloom Filter (DBF)*. We use only one PBF and one BF to explore whether it is necessary for RoBF to use multiple parameters.
- *Perfect Prefix Filter (PPF)*. PPF is a filter that can theoretically answer any prefix query accurately. We construct such a Filter to observe the high performance limit of the hash-based Filter.
- *SuRF-Real (SuRF)* [9]. SurF is a trie-based filter based on FST (Fast Succinct Trie). For most SST, SuRF cannot be created at the BPK value less than 15.

C. Results

Figure 3 shows the relationship between the false positive rate of different filters and BPK.

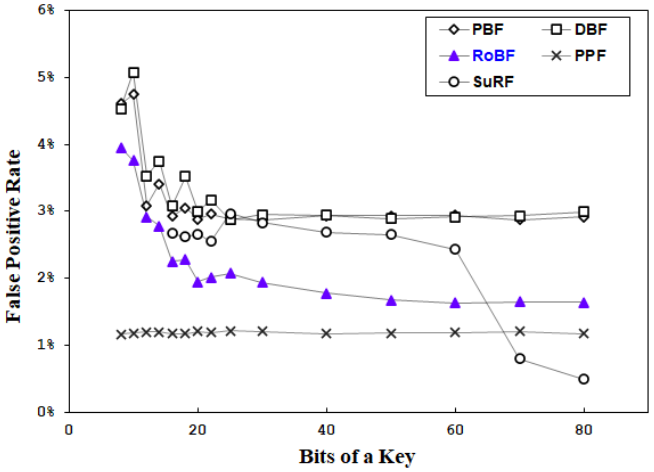


Fig. 3. **FPR of filters under different BPK**. With the exception of the Perfect Prefix Filter (PPF), each curve represents the performance change of a Filter as the BPK increases. PPF's FPR is theoretically fixed because it always answers any prefix query accurately.

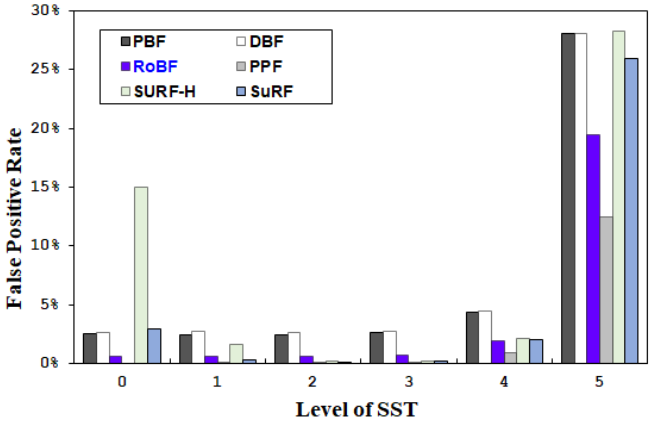


Fig. 4. The performance of filters in different levels of LSM-tree

As we can see, only SuRF finally surpasses the Perfect Prefix Filter (PPF) with the increase of memory.That is to say, in this test, when the memory size of the filter exceeds 70 BPKs, that is, the average memory allocated per key is greater than 8.4 bytes of space, the performance of the hash-based filter will hardly surpass that of the trie-based filter.

This is due to the third type of false positives mentioned earlier. Since hash-based filters can only process prefix queries, there is an inherent false positives rate. With the increase of BPK, the performance of RoBF gradually converges to PPF, and the false positives rate cannot be further reduced, while trie-based does not have such a problem.

After the BPK reached 70, the false positive rate of SuRF decreased significantly, which is also the characteristic of trie-based filter. Such a filter must increase the length of the stored prefix in order to effectively support a small range of queries, rather than simply adjusting the position of the test points as with a hash-based filter, which often means higher memory consumption.

In practice, most filters cannot provide more than 8 bytes per key. It can be seen from the first half of the curve that the false positive rate of PBF/DBF/RoBF decreases obviously when BPK increases from 8 to 20, which reflects the feature that the hash-based filter is suitable for small memory. In contrast, DBF has relatively close performance to PBF and relatively poor performance relative to RoBF, which means that RoBF has more parameters than DBF that are not redundant.

As memory increases further, the false positive rate of PBF and DBF no longer changes significantly, but the false positive rate of RoBF can still decrease further, because RoBF can use memory to support a range of queries of different lengths, rather than only reducing the false positive rate of individual prefix queries.

As shown in Fig. 1, We can see the advantages and disadvantages of the two types of filters: the trie-based filter must continuously store the prefix information of the key, otherwise it cannot be guaranteed that there is no false negative error, so the trie-based filter always needs some space to store the prefix information that is of no value for the reply; On the other hand, the hash-based filter, even if it can answer the prefix query perfectly, does not improve the hit ratio of the range query further, thus causing performance bottlenecks when memory is large enough.

Figure 4 shows the results of our second experiment. We compared the false positive rates of different filters at different levels of LSM-tree. In order to show the characteristics of the filter more intuitively, we randomly selected a SST from each level and showed its FPR.

As can be seen from the figure, the false positives rate of SuRF decreases first and then rises with the increase of Level, which means that the trie-based filter has poor performance in both the upper and lower levels, but the reasons are different: in the upper level, the distribution of keys is relatively sparse, which means that SuRF needs more space to store prefix information that is not helpful for query; In the lower level, the range query requires the last few characters to respond correctly, and the trie-based filter requires a higher BPK to better support such a small range query.

The performance of RoBF is relatively stable, but there is a significant drop in performance in the last two levels.This is due to the fact that as the keys become denser, there are fewer interval queries that can be filtered by the prefix, i.e. the proportion of type 3 errors increases.

As mentioned earlier, allocating more memory for each filter in the upper write buffer can improve query performance. This is due to the fact that there is less data in the upper write buffer and more queries, and the same amount of memory can reduce more false positives for queries.

Taking the case of BPK=12 as an example, a basic scenario is to assign a filter of 12 BPKs to each file, in which case the average false positive rate per file is 0.6%; But if we map the keys in levels 0-2 directly to memory, and use SuRF filters with BPK of 78 and 34 in levels 3-4, respectively, and RoBF filters with BPK of 9 in level 5, the average memory

BPK	8		10		12		14		16	
Level-0	MAP	128	MAP	128	MAP	128	MAP	128	MAP	128
Level-1	MAP	128	MAP	128	MAP	128	MAP	128	MAP	128
Level-2	MAP	128	MAP	128	MAP	128	MAP	128	MAP	128
Level-3	SuRF	93	SuRF	66	SuRF	78	SuRF	110	SuRF	62
Level-4	SuRF	28	SuRF	43	SuRF	34	SuRF	33	SuRF	30
Level-5	RoBF	5	RoBF	6	RoBF	9	RoBF	11	RoBF	14
After Balance	1.339%		1.330%		1.280%		1.284%		1.284%	
Before Balance	10.797%		8.652%		6.318%		6.551%		6.435%	

Fig. 5. **The optimal filter configuration scheme under different BPK** - The figure shows optimal solutions for five scenarios. For example, when BPK is 12, Level-3 is recommended to use SuRF with 78 BPKs. "MAP" represents a full mapping strategy, while 128 represents a total key length of 16 bytes.

consumption per file still does not exceed 12 BPKs, and Fig. 5 shows that the false positives rate drops to 0.11%, a reduction of 79.74%.

This conclusion is consistent with the law of practice. In the case of RocksDB, when RocksDB is configured, the index of the upper SST tends to have higher access frequency, which means that the index of the upper SST is actually resident in memory as long as we can provide sufficient cache space.

Since the upper-level SST filters use more memory, the memory usage of the lower-level SST is significantly lower than average, and the performance benefits of RoBF in small-memory scenarios are easier to realize. It is worth noting that SuRF is used as a middle level filter, which is consistent with our analysis in the previous section, i.e., trie-based filters perform poorly when the key distribution is too sparse or too dense.

V. Conclusions and Future Work

In this paper, we propose a multi-parameter filter RoBF to improve the range queries of Bloom filters in LSM-tree. We propose an algorithm to determine the parameters of RoBF based on query distribution. We conducted comparative experiments to compare RoBF with various filters, and the results suggested the efficiency of RoBF. RoBF adopts multi-filter strategy, because we believe that different filters should be used in different levels of LSM-tree. Particularly, the lower the level, the higher the BPK provided for the filter should be. We designed an algorithm to search for the optimal solution. The results of this algorithm confirm our conjecture that the optimal solution generated by the algorithm can reduce the global FPR to 20% of the static solution.

At present, we determine the type and parameters of the filter through the data distribution and the memory size of each level in LSM-tree, while the memory size of filter in each level is determined by another algorithm. Two independent algorithms will lead to the calculation of the optimal solution takes a long time, which may affect the real-time performance of the parameters of the filter. In the future work, we plan to integrate the two algorithms together and adjust the parameters with a faster algorithm, which can also provide better theoretical support for the experimental results of this work. Also, we will extend RoBF to make it suitable for persistent memory-based key-value stores [12], e.g., integrating RoBF with the persistent memory-friendly adaptive Radix tree [13] or B+-trees [14].

Acknowledgements

This work was supported by the National Science Foundation of China (No. 62072419). We also thank the anonymous reviewers for for their suggestions and comments to improve the quality of the paper. Peiquan Jin is the corresponding author.

References

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
[2] "Cassandra," http://cassandra.apache.org/.
[3] "Hbase," http://hbase.apache.org/.
[4] "Rocksdb," http://rocksdb.org/.
[5] Y. Wang, P. Jin, and S. Wan, "Hotkey-lsm: A hotness-aware lsm-tree for big data storage," in *Proc. of IEEE International Conference on Big Data (Big Data)*, pp. 5849–5851.
[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[7] P. Jin, C. Yang, C. S. Jensen, P. Yang, and L. Yue, "Read/write-optimized tree indexing for solid-state drives," *VLDB J.*, vol. 25, no. 5, pp. 695–717, 2016.
[8] J. Lee, M. Shim, and H. Lim, "Name prefix matching using bloom filter pre-searching for content centric network," *J. Netw. Comput. Appl.*, vol. 65, pp. 36–47, 2016.
[9] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Surf: Practical range query filtering with fast succinct tries," in *Proc. of the 2018 International Conference on Management of Data (SIGMOD)*, 2018, pp. 323–336.
[10] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proc. of the 2017 ACM International Conference on Management of Data (SIGMOD)*, 2017, pp. 79–94.
[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010, pp. 143–154.
[12] R. Liu, P. Jin, X. Wang, Z. Zhang, S. Wan, and B. Hua, "Nvlevel: A high performance key-value store for non-volatile memory," in *Proc. of the 21st IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2019, pp. 1020–1027.
[13] J. Zhang, Y. Luo, P. Jin, and S. Wan, "Optimizing adaptive radix trees for nvm-based hybrid memory architecture," in *Proc. of IEEE International Conference on Big Data (Big Data)*, 2020, pp. 5867–5869.
[14] Y. Luo, P. Jin, Q. Zhang, and B. Cheng, "Tlbtree: A read/write-optimized tree index for non-volatile memory," in *Proc. of the 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 1889–1894.