Noname manuscript No.
(will be inserted by the editor)

Enabling Space-Time Efficient Range Queries with REncoder

Zhuochen Fan[1], Bowen Ye[1], Ziwei Wang[1], Zheng Zhong[1], Jiarui Guo[1], Yuhan Wu[1], Haoyu Li[1], Tong Yang[1]✉, Yaofeng Tu[2], Zirui Liu[1], Bin Cui[3]

Received: date / Accepted: date

Abstract A range filter is a data structure to answer range membership queries. Range queries are common in modern applications, and range filters have gained rising attention for improving the performance of range queries by ruling out empty range queries. However, state-of-the-art range filters, such as SuRF and Rosetta, suffer either high false positive rate or low throughput. In this paper, we propose a novel range filter, called REncoder. It organizes all prefixes of keys into a segment tree, and locally encodes the segment tree into a Bloom filter to accelerate queries. REncoder supports diverse workloads by adaptively choosing how many levels of the segment tree to store. In addition, we also propose a customized blacklist optimization for it to further improve the accuracy of multi-round queries. We theoretically prove that the error of REncoder is bounded and derive the asymptotic space complexity under the bounded error. We conduct extensive experiments on both synthetic datasets and real datasets. The experimental results show that REncoder outperforms all state-of-the-art range filters, and the proposed blacklist optimization can effectively further reduce the false positive rate.

Keywords Range Queries · Range Filters · Segment Tree · False Positive Rate · Throughput

Co-first authors: Zhuochen Fan, Bowen Ye, and Ziwei Wang.

✉ Tong Yang (yangtong@pku.edu.cn)
Zhuochen Fan (fanzc@pku.edu.cn)
Bowen Ye (bwye@stu.pku.edu.cn)
Ziwei Wang (wangziwei@stu.pku.edu.cn)
Zheng Zhong (zheng.zhong@pku.edu.cn)
Jiarui Guo (ntguojiarui@pku.edu.cn)
Yuhan Wu (yuhan.wu@pku.edu.cn)
Haoyu Li (lihy@pku.edu.cn)
Yaofeng Tu (tu.yaofeng@zte.com.cn)
Zirui Liu (zirui.liu@pku.edu.cn)
Bin Cui (bin.cui@pku.edu.cn)

[1] National Key Laboratory for Multimedia Information Processing & School of Computer Science, Peking University
[2] ZTE Nanjing Research and Development Center
[3] Key Laboratory of High Confidence Software Technologies (MOE) & School of Computer Science, Peking University

The preliminary version of this paper titled "REncoder: A Space-Time Efficient Range Filter with Local Encoder" [55] was published in the Proceedings of the 2023 IEEE 39th International Conference on Data Engineering (ICDE), Apr. 3-7, 2023, Anaheim, California, USA.

1 Introduction

1.1 Background and Motivation

Range queries are common operations in modern database applications [3, 5, 18, 28, 49]. A range filter is a data structure to answer range membership queries [11, 38, 60]. Unlike a Bloom filter [14] that only supports point queries (*e.g.*, is key 87 in the set?), a range filter determines whether a queried range contains any item (*e.g.*, is any key ranging from 56 to 7982 in the set?). Range filters have gained much attention because they can reduce the number of I/Os by eliminating empty range queries.

Below we introduce three use cases of range filters. **Use Case 1: Log-structured merge (LSM)-tree** [42]. LSM-trees are widely used in DBMS [1, 5, 7, 12, 16, 21, 34, 46] and have many applications such as time-series databases [32, 54] and graph databases [4, 33], thanks to their excellent writing performance. However, because an item can reside in Sorted String Tables (SSTables) from all levels in the LSM-tree, we have to access multiple SSTables from disk when retrieving the item. It wastes expensive disk I/Os when the item

does not exist in the SSTables. For point queries, an LSM-tree typically maintains a Bloom filter in memory for each SSTable to check the existence of items before issuing disk I/Os [20, 37]. For range queries, range filters can benefit an LSM-tree in a similar way. When processing a range query, before accessing an SSTable, we first query the corresponding range filter to check whether there are items within the queried range. If the filter returns true, there is a high probability that the range contains at least one item (could be a false positive), and we should load the SSTable from disk to verify and retrieve the desired item(s). Otherwise, we can skip searching the SSTable because we are 100% sure that the result set is empty for this range. Empty ranges are common especially for LSM-trees with many levels/runs. Therefore, range filters are effective in reducing the number of I/Os and thus improving query performance.

Use Case 2: B+tree [17]. B+trees are the most widely used index structures in DBMSs. Typically, a B+tree has a large fanout and its leaf nodes are not cached in memory. To save unnecessary leaf node accesses, we can maintain a range filter in memory for each leaf node so that we visit a particular leaf node only when the corresponding range filter returns positive. In this way, empty point and range queries do not incur any disk I/Os. Range filters can also be applied to optimize many other B-tree variants [25].

Use Case 3: R-tree [26]. Range filters can also benefit an R-tree and its variants [13, 50]. An R-tree is a generalization of a B-tree in multi-dimensional space. Take 2-dimensional R-trees (*i.e.*, the keys in the R-tree are 2-dimensional) as an example. We denote the 2-dimensional key using (x, y). A spatial query such as retrieving the items satisfying $42 < x < 100$ and $58 < y < 111$ can be regarded as a 2-dimensional range query. Similar to a B+tree, for each leaf node of an R-tree, we include an in-memory range filter to avoid unnecessary disk I/Os. Since the keys in R-tree are 2-dimensional, we first transfer them to 1-dimensional by Z-order[1] [40] and then store them in the range filers.

Designing an efficient range filter is challenging. There are three primary goals. First, a range filter must be ***compact*** so that it can fit in memory. Second, it must be ***accurate*** (*i.e.*, low false positive rate) to save as many unnecessary I/Os as possible. Finally, it must be ***fast*** so as not to significantly increase the CPU usage of target applications. As the state-of-the-art solution [38] has approached the theoretical lower bound in space [24], in this paper, we focus on improving range filters' performance while retaining the space-efficiency.

[1]For a 2-dimensional key, interleave the binary representations of its x and y to obtain the corresponding 1-dimensional key.

1.2 Prior Works

There are four current state-of-the-art (SOTA) range filters: SuRF [60], Rosetta [38], SNARF [52] and Proteus [31]. SuRF is based on trie, its key idea is to prune the lower levels of the trie and then succinctly encode the remaining trie. Because of pruning, however, SuRF does not provide theoretical guarantee on the false positive rate (FPR), and the FPR increases significantly in correlated workloads[2]. Rosetta overcomes the shortcomings of SuRF by using Bloom filters. It organizes all prefixes of keys in a segment tree [39, 45] (refers to binary segment tree by default), and uses a series of Bloom filters to store the segment tree. However, Rosetta has a relatively low in-memory performance because of many queries to Bloom filters. SNARF learns a CDF model of the keys, then uses the model to store information of the data and answer range queries. By using the learned model, SNARF achieves lower FPR. However, like SuRF, the FPR of SNARF increases significantly in correlated workloads. Proteus combines the trie and the Bloom filter. It proposes Contextual Prefix FPR (CPFPR) model, by which it can choose a design (of trie and Bloom filter) that achieves a low FPR. However, using the CPFPR model requires sampling the workload before constructing the range filter, which is impractical for some use cases.

1.3 Our Proposed Solution

We propose **Range Encoder (REncoder)**, a novel range filter that improves over the state-of-the-art solutions in the aforementioned design goals. Based on REncoder, we also propose REncoderSS and REncoderSE for different use cases. We define three use cases: A) REncoderSS version: sampling queries is forbidden, and the theoretical error bound is not required, with the relevant scenario being the real-time trading system; B) REncoderSE version: sampling queries is allowed, and the theoretical error bound is required, with the relevant scenario being the metadata management of the distributed file system; C) REncoder version: sampling queries is forbidden, but the theoretical error bound is required, with the relevant scenario being the consistency checking mechanism of the distributed database. According to evaluation on both synthetic and real-world datasets: 1) REncoder(SS/SE) is compact, with a size close to the theoretical lower bound;

[2]The queried keys are similar to the stored keys.

Table 1: Comparison of range filters[3].

Use Case	Range Filter	FPR (ratio)	Filter Throughput (ratio)	Overall Throughput (ratio)	Theoretical error bound	Need sample query
A	SuRF	0	4.5	1	No	No
	SNARF	**3.8**	1.3	16.8	No	No
	ProteusNS	0.7	**7.2**	0.3	No	No
	REncoderSS	**3.1**	**5.2**	**24.3**	No	No
B	Rosetta	2.2	1	1.9	Yes	Yes
	Proteus	**5.1**	**4.2**	23.8	Yes	Yes
	REncoderSE	**3.9**	**5.3**	**24.8**	Yes	Yes
C	REncoder	**2.4**	**4.6**	**2.5**	Yes	No

2) it is accurate, with an FPR ranking in the top two for all use cases; 3) it is fast, with a filter throughput ranking in the top two for all use cases; 4) it has the best overall throughput for all use cases. The comparison results are summarized in Table 1.

Similar to Rosetta, REncoder organizes all prefixes of keys in a segment tree and use the segment tree to support range queries. Each node of the segment tree corresponds to a range, and the range of the parent node is the union of the ranges of its child nodes. Note that the ranges of the nodes at the same level are non-overlapping. The node of the segment tree records whether there is a key in its range. It means that for each key, the segment tree records the existence of not only the key itself but also all of its prefixes (ranges containing the key). Figure 1 shows an example. Inserting binary key 1101 (13) into a segment tree will record not only 1101 but also 110 ($[12, 13]$), 11 ($[12, 15]$), and 1 ($[8, 15]$). In this way, we can divide the range query into up to $\log_2(R)$ point queries of prefixes, where R is the range size. For example, querying range $[0010, 1111]$ ($[2, 15]$) is equivalent to querying the existence of prefix 001 ($[2, 3]$), 01 ($[4, 7]$), and 1 ($[8, 15]$).

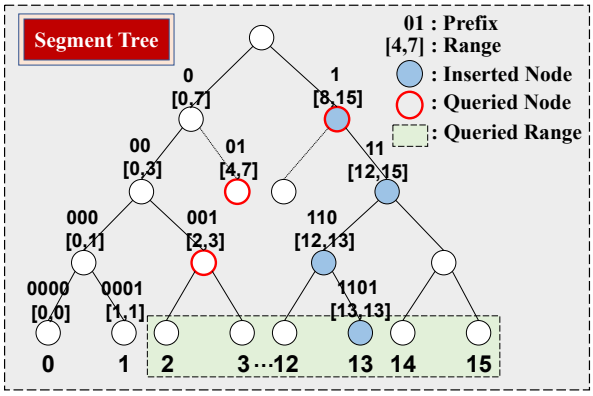

Fig. 1: Example of segment tree.

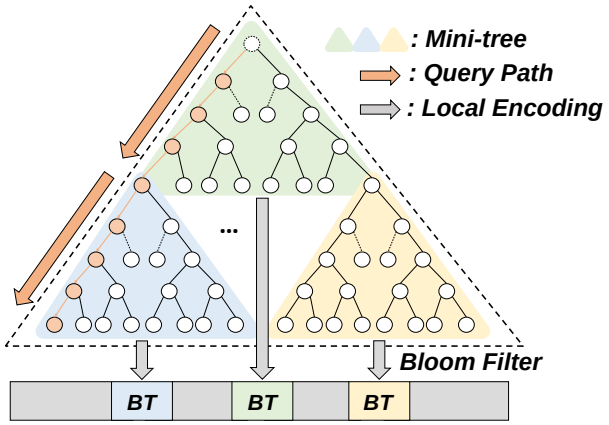

Fig. 2: REncoder takes advantage of query locality in segment tree.

Below are the key techniques of REncoder. REncoder stores the segment tree by using Bloom filter, and uses a compact encoding algorithm to improve the speed. The key idea of our encoding method is to leverage the query locality of the segment tree, so as to reduce the number of memory accesses. Specifically, as shown in Figure 2, we first divide the entire segment tree into many mini-trees. Then we encode each mini-tree into a bitmap: we make each node in the mini-tree correspond to a bit in the bitmap. If the node exists, the corresponding bit will be set to 1. Otherwise, the corresponding bit will be set to 0. The encoded bitmap is called **Bitmap Tree** (BT). After encoding, we insert each BT into a Bloom filter independently, so that the information in the same BT, *i.e.*, the nodes in the same mini-tree, will be encoded into the Bloom filter locally. For a mini-tree with N nodes, the size of its corresponding BT is N bits. When $N \leqslant 512$ (the maximum data size that SIMD instructions can handle [8]), we only need one memory access to obtain the information of all nodes in the mini-tree. Figure 2 shows an example of $N = 31$. Following the query path, the traditional

method used by Rosetta needs to query a total of 8 nodes (8 memory accesses). In contrast, REncoder only needs to query 2 mini-trees (2 memory accesses).

Table 2: Space cost of REncoder.

Version	FPR				
	50%	**25%**	**10%**	**1%**	**0.1%**
REncoder	6.5	8.5	10.5	16	21
REncoderSS(SE)	2	3	4.5	9.5	14.5

Due to the diversity of dataset/workload (keys/queries) in practice, we also propose an optimized REncoder. It can adaptively choose how many levels of the segment tree to store to remain efficient in different datasets. Based on the optimized **REncoder**, we propose two new versions: one **S**elects the **S**tart level (from which level to start storing) according to the dataset, called **REncoderSS**; the other **S**elects the **E**nd level (to which level to end storing) according to the workload, called **REncoderSE**. In addition, we also propose an optimization called blacklist that can be applied to all the above REncoder versions specifically for multi-round queries. Then, we prove that REncoder has a theoretical error bound, and we derive the asymptotic space complexity under the bounded error. When the keys (64-bit integers) and queries are uniformly-distributed, given various bounded error (FPR), the space (bits per key) required by each version of REncoder is shown in Table 2. The source code of REncoder is available on GitHub [9].

We make the following contributions in this paper.

1. We propose a novel range filter, called REncoder. It achieves great query performance by taking advantage of the locality. The core design is locally encoding the segment tree into the Bloom filter. The encoding scheme is generic, and it can be applied to various tree structures.
2. We propose an optimized REncoder which can adaptively adjust the number of stored levels according to the workload. Based on the optimized REncoder, we propose two new versions for different use cases: REncoderSS and REncoderSE.
3. We propose a blacklist optimization specifically for multi-round queries, which is applicable to all REncoder versions.
4. We theoretically prove that the error (*i.e.*, FPR) of REncoder is bounded. Given the bounded error ϵ, the space that REncoder needs is $O(N(k + \log \frac{1}{\varepsilon}))$, where N is the number of the items, and k is the number of the hash functions of the Bloom filter.
5. We carry out extensive experiments on synthetic and real datasets. The results show that when using the same amount of memory, REncoder outperforms the SOTA solutions.

2 Preliminaries and Related Work

2.1 Definition

Range filter is a data structure for representing a set S, Given a query range $R = [a, b]$: 1) if the range contains any item in the set (*i.e.*, $R \cap S \neq \emptyset$), the range filter must report true; 2) if the range contains no item in the set (*i.e.*, $R \cap S = \emptyset$), the range filter reports false with probability $1 - \epsilon$, while ϵ is false positive rate of the range filter.

2.2 Range Filters

Most range filters can be divided into two categories: trie-based solutions [11, 60] and Bloom filter-based solutions [22, 38]. There are some range filters that do not fall into either of the two categories. We denote them as novel solutions [31, 52].

Trie-based Solutions: Trie-based range filters include Adaptive Range Filter (ARF) [11], Succinct Range Filter (SuRF) [60], *etc.*. ARF first builds a full trie, then determines which nodes to truncate by training on sample queries, finally encodes the truncated trie into a bit sequence. Different from ARF, SuRF does not need training, the trie is truncated at a certain length. In addition, SuRF uses a hybrid encoding scheme [27] to encode the trie. SuRF also have some advanced versions which save various additional information for each key including hashed key suffixes, real key suffixes and mixed key suffixes.

Bloom filter-based Solutions: Bloom filter-based range filters include Prefix Bloom filter [22], Robust Space-Time Optimized Range Filter (Rosetta) [38], and bloomBF [41], *etc.* Prefix Bloom filter inserts predefined-length prefixes of each key into Bloom filters, and it can only be used for corresponding fixed-prefix queries. In contrast, Rosetta inserts every prefix of each key into Bloom filters. For L-bits keys, there are L Bloom filters in Rosetta. Prefixes of length i of

[3] FPR = ln(FPR of SuRF / FPR of current range filter)
Filter Throughput (FT) = FT of current range filter / FT of Rosetta
Overall Throughput (OT) = OT of current range filter / OT of SuRF
FPR and FT take the average of all experiments, and OT takes the average of experiment on range queries.

each key are inserted into i^{th} Bloom filter. In essence, Rosetta builds an "implicit segment tree" on the Bloom filters. bloomRF mainly guarantees the local order of sub-ranges by proposing monotonic hashing. However, under the premise of guaranteeing FPR, the number of hash functions it requires increases as the data sparsity increases, thereby affecting its query speed, while REncoder's can remain unchanged.

Novel Solutions: Novel range filters include Sparse Numerical Array-Based Range Filters (SNARF) [52], and Self-designing Approximate Range Filter (Proteus) [31]. SNARF learns a CDF model of the keys, and uses the model to map the keys into a sparse bit array. Then, the sparse bit array is compressed to save space. To answer a range query, SNARF uses the learned model to obtain the bit positions corresponding to the left and right boundaries of the query. Then SNARF checks whether there is a bit between the two bit positions in the compressed bit array (*i.e.*, whether there is a key within the range). The key of Proteus is the CPFPR model, which formalizes the FPR of prefix-based filters in various design spaces. Proteus combines the trie-based range filters and Bloom filter-based range filters. It uses both an FST (Fast Succinct Trie) [60] and a prefix bloom filter, and uses the CPFPR model to design (the depth of FST and the prefix length of prefix bloom filter) to achieve optimal FPR.

2.3 Variants of Bloom Filters

Bloom filters [14] are widely used in database and network, thanks to their three advantages: fast, compact and only have one-sided errors. There are many variants of Bloom Filters for different uses [15, 30, 35, 36, 43, 47, 51, 53, 56–58]. Among them, the closest to REncoder are Shifting Bloom filter (ShBF) [58] and Persistent Bloom Filters (PBF) [43]. ShBF is proposed to improve the performance of standard Bloom filter. Its key novelty is encoding partial information of an item in a location offset. Both ShBF and REncoder take advantage of the locality to reduce hash operations. However, ShBF takes advantage of the locality by locally encoding partial information of the item, while REncoder takes advantage of the locality by locally encoding prefixes of the item. In fact, ShBF is orthogonal to REncoder.

PBF is used for temporal membership queries, *e.g.*, has this item appeared between 6am and 8am? Both PBF and REncoder use segment trees and Bloom filters, but in a totally different way. The segment tree of PBF records time ranges, while that of REncoder records key ranges. PBF uses several Bloom filters to store the segment tree, while REncoder only need one Bloom filter. Moreover, REncoder takes advantage of locality to significantly improve its performance, and can adaptively choose the stored levels of the segment tree according to datasets.

3 Range Encoder

In practice, take LSM-tree as an example, a REncoder is constructed for each SSTable of a LSM-tree. When executing a point or range query, before accessing an SSTable, we first query the corresponding REncoder, and only when the REncoder returns true, we will load the SSTable from the disk. Whenever the LSM-tree performs a merge operation, the REncoder needs to be rebuilt using the new items. Below we will discuss the construction and the query of REncoder in detail. The terms used in this paper are shown in Table 3.

Table 3: Terms used in this paper.

Term	**Meaning**
L	Length of key
p	Prefix of key
k	Number of hash functions
h_i	The i^{th} hash function
L_s	Number of stored levels
R	Range query size
R_{max}	Maximum range query size
P_1	The proportion of 1 in the bit array of RBF

3.1 Constructing REncoder

Similar to Rosetta, REncoder organizes all prefixes of all keys into a segment tree, and stores the segment tree using the Bloom filter. However, REncoder uses a novel encoding scheme to utilize the locality, which can significantly improve performance. For each key, we first encode all its prefixes into several BTs. Then we insert the BTs into one special Bloom filter named **Range Bloom Filter** (RBF). RBF is similar to the standard Bloom filter. The difference is that the standard Bloom filter can only insert one item at a time, *i.e.*, set one bit to 1 at a time, while RBF can insert a bitmap in one memory access, so as to set multiple bits to 1 simultaneously. Once all keys are encoded and inserted into the RBF, the construction of the REncoder is done. Take the example of encoding 4 consecutive prefixes into one BT. The insertion procedure is described in Algorithm 1. Thanks to RBF, the construction efficiency of REncoder is significantly improved compared with Rosetta.

Theoretically, the magnitude of the improvement is proportional to the number of consecutive prefixes encoded into one BT.

Algorithm 1: Insert

Input: key to be inserted

1 $i \leftarrow 4$;
2 **while** $i \leqslant L$ **do**
3 $\quad key_{suffix} \leftarrow$ key & $0x0000000F \mid 0xFFFFFFE0$;
4 $\quad bt \leftarrow$ CodeIntoBitmap(key_{suffix});
5 $\quad$ RBF.Insert($key >> 4, bt$);
6 $\quad key \leftarrow key >> 4$;
7 $\quad i \leftarrow i + 4$;
8 **end**

Algorithm 2: RBF.Insert

Input: p_{hash}, bt

1 $i \leftarrow 1$;
2 **while** $i \leqslant k$ **do**
3 $\quad pos \leftarrow h_i(p_{hash})$;
4 $\quad *(array + pos) \leftarrow *(array + pos) \mid bt$;
$\quad$ `// array is the start address of the array of RBF`
5 $\quad i \leftarrow i + 1$;
6 **end**

We now explain the insertion of RBF. The procedure is presented in Algorithm 2. Similar to the standard Bloom filter, the insert position is calculated by the hash function (Line 3). However, RBF takes the insert position as the starting point and inlays the bitmap using **OR** operation, instead of only setting the bit of insert position to 1 (Line 4).

An insertion example: The left part of Figure 3 shows an insertion example of REncoder. The insertion is divided into three steps: 1) We split the key 164 (corresponding to 10100100) into prefix 1010 and suffix 0100. Note that the suffix 0100 actually represents the last 4 consecutive prefixes of key 164 (1010**0**, 1010**01**, 1010**010**, 1010**0100**). 2) We encode suffix 0100 into a 32-bit (4-byte) BT. First, we build a **virtual** segment tree with a depth of 5, which can record the range $[0000, 1111]$. Then we number each node of the segment tree in breadth-first order. The root node is the 1^{st} node, 0, 01, 010 and 0100 corresponds to 2^{nd}, 5^{th}, 10^{th} and 20^{th} node, respectively. Next we set the corresponding positions in the BT to 1, and obtain BT 11001000010000000001000000000000. In this way, we encode the segment tree recording key 0100 into a BT. 3) We hash the prefix 1010 into k indices of RBF, and inlay the BT using operation **OR**. In this way, we store the built virtual segment tree in RBF. Note that we do not build a real segment tree, but use the structure of the segment tree to organize keys. The insertion of the next suffix 1010 is the same. Note that there is no prefix before 1010. Therefore, the prefix for hash functions can be 0 or any other constant. Obviously, after insertion, the number of bits set to 1 in RBF is the same as that in Bloom filters of Rosetta, which guarantees that the accuracy of REncoder can match Rosetta.

3.2 Range Queries with REncoder

The difference between REncoder and Rosetta in range queries lies in the queries to Bloom filter. In Rosetta, each query to Bloom filter can check the existence of one prefix. In REncoder, each query to RBF obtains one BT which can check the existence of several (*e.g.*, 4) consecutive prefixes. Thanks to the locality of range queries, *i.e.*, consecutive prefixes are often accessed in the same range query, REncoder significantly improves query efficiency.

We now illustrate how a range query is executed in REncoder. The procedure of query is divided into two stages: *Decomposition* stage and *Verification* stage. In *Decomposition* stage, similar to Rosetta, we decompose the target range (R_t) into several non-overlapping sub-ranges, each of which corresponds to a prefix that can exactly cover all keys in the range. Specifically, we denote the range corresponding to the current prefix as R_p. We start from the shortest prefix (the empty prefix), which means R_p is $[0, maxkey]$. We then compare R_p with R_t. There are three cases: 1) if R_p is non-intersected with R_t, we do nothing; 2) if R_p is contained in R_t, **we record R_p as a sub-range**; 3) if R_p is intersected with R_t, we append 0 (and 1) to the current prefix to get the new R_p $[0, maxkey/2]$ (and $[maxkey/2 + 1, maxkey]$), then compare the new R_p with R_t. Here, "append 0" and "append 1" correspond to the first half and second half of the range, respectively. We repeat the above process until there is no new R_p. Take the 4-bit key as an example. For the target range $[0, 4]$, we start from the max R_p $[0, 15]$. $[0, 15]$ is intersected with $[0, 4]$, we compare $[0, 7]$ ($[\mathbf{0}000, \mathbf{0}111]$, prefix 0) and $[8, 15]$ ($[\mathbf{1}000, \mathbf{1}111]$, prefix 1) with $[0, 4]$. $[8, 15]$ is non-intersected with $[0, 4]$, we do nothing. $[0, 7]$ is intersected with $[0, 4]$, we compare $[0, 3]$ (append 0 by [0, 7]→[**00**00, **00**11], prefix 00) and $[4, 7]$ (append 1 by [0, 7]→[**01**00, **01**11], prefix 01) with $[0, 4]$. $[0, 3]$ is contained in $[0, 4]$, we record $[0, 3]$ as a sub-range. Similarly, we can get another sub-range $[4, 4]$ (corresponding to the prefix 0100).

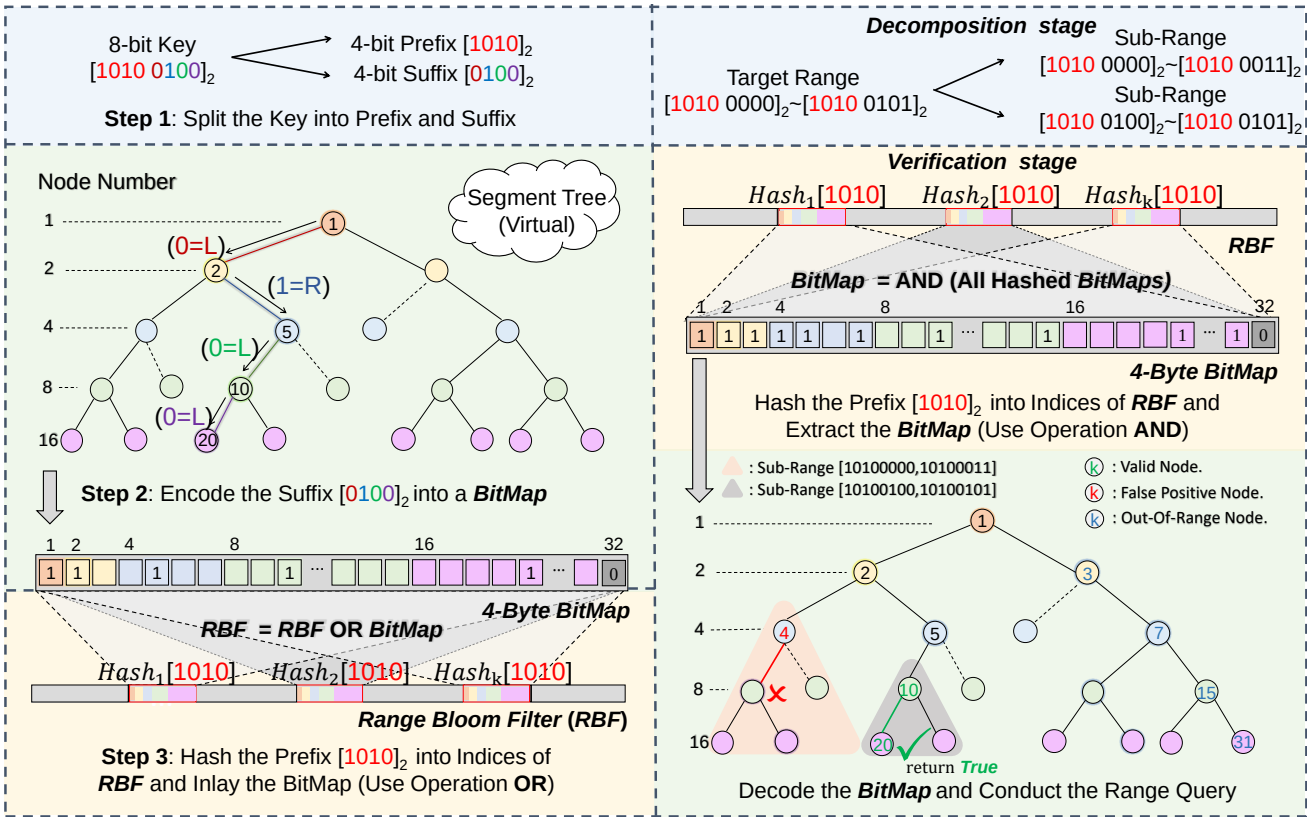

Fig. 3: Examples of REncoder.

After the decomposition of the target range, we turn to *Verification* stage. First, we query RBF for the existence of the prefix corresponding to the first sub-range. If it returns negative, we continue to query for the prefix corresponding to the next sub-range until all prefixes have been queried. If none of them returns positive, REncoder reports that the target range is empty. If one of the queries returns positive, we perform a depth-first traversal of the mini-tree corresponding to the prefix to further verify the existence of it. The traversal procedure is as follows: Starting from the root node of the mini-tree, we query RBF for the existence of the prefix corresponding to the current node, if it returns positive, continue to traverse down the tree, otherwise terminate the current path. If the traversal reaches a leaf node and the query to RBF returns positive, REncoder reports that the sub-range is not empty, which also indicates that the target range is not empty. Otherwise, REncoder reports that the sub-range is empty. Note that only when all the sub-ranges in *Verification* stage are empty will REncoder report that the target range is empty.

Still take the example of encoding 4 consecutive prefixes into one BT, the procedure of a range query is shown in Algorithm 3. We first decompose the target range into several sub-ranges, and get the corresponding prefixes (Line 1). Then, we verify the prefixes one by one (Lines 2-7). If there is a verification returns true, the query returns true (Lines 4-6). If none of the verification returns true, the query returns false (Line 8). In *Verification* stage (Lines 9-20), we first query RBF for the existence of the current prefix (*i.e.*, check root node), there are three conditions: 1) If the query returns false, the verification stops and returns false (Lines 10-12). 2) If the query returns true and the length of prefix is equal to that of the key, the verification stops and returns true (Lines 13-15). 3) Otherwise, we query RBF for the existence of the current prefix appended with 0 (*i.e.*, check left mini-tree) and 1 (*i.e.*, check right mini-tree) and return the result of the query (Lines 16-19).

We now specially discuss the query to RBF, the procedure is shown in Algorithm 4. We first extract the hash prefix from the queried prefix by GetHashPrefix function (Line 1). If the current hash prefix is the same as the hash prefix of the previous query, it indicates that the target information of the two queries is in the same BT, so we can directly use the BT obtained from the previous query (Lines 2-3). Otherwise, we have to perform hash operations on the current hash prefix to get the BT that contains information about the queried prefix (Lines 4-12). We also need to store the current hash prefix and BT for the next query (Lines 13-14).

Algorithm 3: Range Query

```
   Input: low, high
   // [low, high] is the target range
1  plist ← Decompose(low, high);
   // plist is a list of the prefixes
      corresponding to sub-ranges
2  for each p ∈ plist do
3  |  l ← length of p;
4  |  if Verify(p,l) then
5  |  |  return true;
6  |  end
7  end
8  return false;
9  Function Verify(p,l):
10 |  if !RBF.Query(p, l) then
11 |  |  return false;
12 |  end
13 |  if l == L then
14 |  |  return true;
15 |  end
16 |  if Verify(p,l+1) then
17 |  |  return true;
18 |  end
19 |  return Verify(p + 2^{L-l-1}, l + 1);
20 end
```

Finally, we extract the bit that indicates the existence of the queried prefix from BT by GetBitFromBitmap function and return it (Line 16).

A range query example: The right part of Figure 3 shows a range query example of REncoder. Suppose the target range is $[160, 165]$ (corresponding to $[10100000, 10100101]$), and key 164 and some other keys (not included in the target range) have been inserted. We first decompose the target range into two sub-ranges $[10100000, 10100011]$ and $[10100100, 10100101]$. Then it turns to *Verification* stage. For the sub-range $[10100000, 10100011]$ (corresponding to prefix 1010**00**), we extract the hash prefix 1010 by which we can obtain the BT 11111010010000100001000000000010 from RBF. As discussed in the insert example, each bit in the BT corresponds to a node in the segment tree. Therefore, we can decode the BT to a segment tree. We find that the bit corresponding to the 4^{th} node (prefix 101000) of the segment tree in the BT is 1, so the traversal of the mini-tree corresponding to 4^{th} node begins: We first check the prefix 1010**000**, *i.e.*, traverse to 8^{th} node. Since the hash prefix of prefix 1010000 is still 1010, there is no need to query RBF again, we can directly use the BT (segment tree) obtained from the previous query. It turns out that the bit corresponding to the 8^{th} node in the BT is 0, so the current path is terminated; Then we check the prefix 1010**001**, *i.e.*, traverse to 9^{th} node. We can still use the BT obtained from the previous query because of the same hash prefix, and the bit corresponding to the 9^{th} node is 0 too. It indicates that the current sub-range is empty, while the 4^{th} node is a false positive node which is coincidentally set to 1 by other BTs. Thus, we turn to verify the next sub-range $[10100100, 10100101]$ (corresponding to prefix 1010**010**). Since the hash prefix of this sub-range has not changed, the previously obtained BT is still available. We first query prefix 1010010 (10^{th} node), its corresponding bit in BT is 1, so we continue to check the prefix 1010**0100**, *i.e.*, traverse to 20^{th} node, the corresponding bit is also 1. Because the 20^{th} node is a leaf node, the verification returns true, which reports that the current sub-range, as well as the target range are not empty. Note that in this example, REncoder only queries RBF once, while Rosetta needs to query Bloom filter 5 times, so the performance of REncoder should be almost 5 times that of Rosetta.

Algorithm 4: RBF.Query

```
   Input: p_query, l
1  p_hash ← GetHashPrefix(p_query, l);
2  if p_hash = p_cache then
   |  // p_cache is the hash prefix of the
   |     previous query
3  |  v ← v_cache;
   |  // v_cache is the BT obtained from the
   |     previous query
4  else
5  |  pos ← h_1(p_hash);
6  |  v ← *(array + pos);
7  |  i ← 2;
8  |  while i ⩽ k do
9  |  |  pos ← h_i(p_hash);
10 |  |  v ← v & *(array + pos);
11 |  |  i ← i + 1;
12 |  end
13 |  p_cache ← p_hash;
14 |  v_cache ← v;
15 end
16 return GetBitFromBitmap(v, p_query, l);
```

3.3 FPR Optimization Through Choice of Stored Levels

A natural question arises: how many levels of the segment tree should we store in RBF? *i.e.*, how many prefixes should be stored for each key? For keys with a size of 64 bits, if we store all 64 prefixes of them, the required space will be unacceptable. Therefore, we have to make a trade-off and only store partial prefixes for each key.

In *Verification* stage, the queries for prefixes start from the prefix that can exactly cover all keys in the sub-range, which means the prefixes before will not be queried. It is obvious that when the maximum range query size is R_{max}, only the last $log_2 R_{max} + 1$ prefixes need to be stored. Considering that analytical systems (*e.g.*, column store [10]) serve range queries of $R > 64$, while filters are more suitable for range queries of $R \leqslant 64$ [38], the maximum number of prefixes that need to be stored is $log_2 64 + 1$, *i.e.*, 7. However, during the experiments, we found that when the memory is given: in some datasets, only storing the last $log_2 R_{max} + 1$ prefixes still takes up excessive space, resulting in high FPR; in other datasets, the last $log_2 R_{max} + 1$ prefixes only occupy little space. In this case, we can store more prefixes and perform additional queries for them to further reduce FPR, *e.g.*, for range $[10100000, 10100011]$, before querying prefix 101000, query prefix 1, 10, 101, 1010, 10100 in turn. Therefore, how to adaptively choose the number of stored levels L_s for different datasets is the key to optimizing FPR.

Algorithm 5: Insert_SelfAdapt

```
   Input: keys
   Output: L_s
1  start_level ← 0;
2  while true do
3  |  P_1 ← 0;
4  |  for key ∈ keys do
5  |  |  Insert(key, start_level);
6  |  end
7  |  P_1 ← RBF.GetOneRate();
8  |  if 0.5 − P_1 ≤ threshold or P_1 ≥ 0.5 then
9  |  |  break;
10 |  end
11 |  start_level ← start_level + 4;
12 end
13 L_s = start_level + 4;
14 return L_s;
```

Although RBF is not exactly the same as the standard Bloom filter, they share some characteristics, such as when the proportion of 1 in the bit array of the Bloom filter (P_1) is close to 0.5, the FPR is almost the lowest [14]. As the length of the bit array and the number of hash functions are determined, P_1 is only related to the number of inserted keys (n_i). Given a dataset containing n distinct keys, n_i of the standard Bloom filter is n regardless of the key distribution (Standard Bloom filter only inserts the key itself). REncoder also inserts several prefixes of the key, thus n_i of it is related to the key distribution and the number of prefixes to be inserted for each key (*i.e.*, the number of stored levels, L_s). For example, given two different datasets $A\{000,001,010\}$, $B\{000,010,100\}$. We denote n_i of REncoder for A and B as A_n and B_n, respectively. When L_s is 1, A_n is 3 ($\{000,001,010\}$), B_n is 3 ($\{000,010,100\}$). When L_s is 2, A_n is 5 ($\{000,001,010,\mathbf{00},\mathbf{01}\}$), B_n is 6 ($\{000,010,100,\mathbf{00},\mathbf{01},\mathbf{10}\}$). When L_s is 3, A_n is 6 ($\{000,001,010,00,01,\mathbf{0}\}$), B_n is 8 ($\{000,010,100,00,01,10,\mathbf{0},\mathbf{1}\}$). Suppose when n_i is 6, P_1 is close to 0.5 (FPR is the lowest). In order to achieve optimal FPR, REncoder needs to store 3 levels for dataset A, and 2 levels for dataset B. In practice, it is time-consuming to calculate n_i and corresponding P_1 under various L_s. Therefore, we can gradually increase L_s during insertion until n_i is close to optimal (P_1 is close to 0.5). The modified insertion procedure is described in Algorithm 5. We insert the prefixes of the keys by round. In the first round, we only insert the last 4 prefixes of each key, and then count the P_1 of the bit array of RBF (Lines 4-7). If the difference between P_1 and 0.5 is less than a predetermined threshold or P_1 is greater than 0.5, the insertion is terminated (Lines 8-10). Otherwise, continue to the next round of insertion and insert 4 more prefixes of each key. The Insert($key, start\_level$) in Line 5 is an overload of Insert(key), which inserts the next 4 prefixes of key starting from $start\_level$ into RBF. Its implementation is similar to Insert(key), so we will not discuss it here due to the space limitation. Note that the number of prefixes inserted for each key in each round can be set according to different needs: set large for better insertion performance, set small for better query performance.

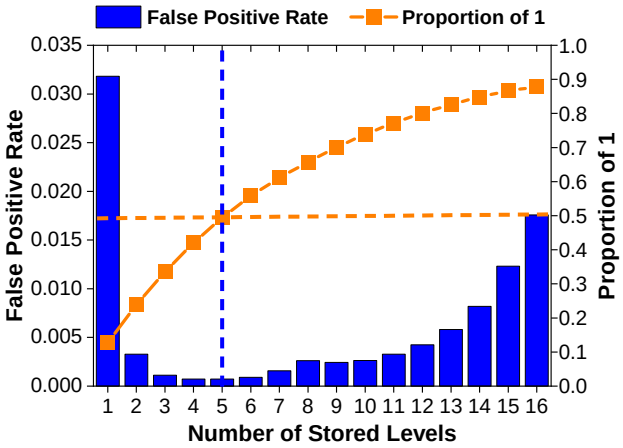

Fig. 4: FPR and P_1 with different L_s.

The modified insertion procedure realizes the adaptive choice of L_s, but its adjustment granularity is coarse, *i.e.*, L_s can only be a multiple of 4. It is obvious that we cannot choose the best L_s in most datasets. An example is shown in Figure 4. In this dataset, when L_s is 5, P_1 is close to 0.5, and the FPR is the lowest. However, using Algorithm 5, we can only choose L_s as 4 or 8. To address this problem, we can reduce the number of prefixes inserted for each key in each round, so as to achieve more fine-grained tuning. When only 1 prefix is inserted for each key in each round, the optimal L_s can always be chosen, but it will also significantly reduce the insertion efficiency. Therefore, we need to make a trade-off between insertion performance and query performance on a case-by-case basis. When the filter needs to be constructed quickly, we can increase the number of prefixes inserted for each key in each round. When there is no requirement for construction efficiency, we can reduce the number of prefixes inserted for each key in each round to improve query performance.

Here comes another question: should storing always start from the lowest level? The answer is no. Still take the dataset $B\{000,010,100\}$ as an example. There is no need to store the lowest level ($\{000,010,100\}$), as the penultimate level ($\{00,01,10\}$) is enough to distinguish all keys. It means that we can start from a higher level to store more significant information. Therefore, we propose REncoderSS. Before inserting keys, REncoderSS counts the maximum length of the longest common prefix (LCP) between any key-key pair (denoted as l_{kklcp}). Instead of the lowest level (*i.e.*, L^{th} level), REncoderSS starts storing from the $(l_{kklcp}+1)^{th}$ level, which is enough to distinguish all keys. Normally, the FPR of REncoderSS is lower than REncoder. But in correlated workloads, the FPR of REncoderSS increase significantly like SuRF because of the absence of the lower levels. To compensate for this shortcoming of REncoderSS, we propose REncoderSE. REncoderSE needs to sample some queries before inserting. After sampling, REncoderSE counts not only l_{kklcp} but also the maximum length of the LCP between any key-query pair[4] (denoted as l_{kqlcp}). Levels below the $(l_{kqlcp})^{th}$ level are necessary because only they can distinguish between certain stored keys and queries. Therefore, when $l_{kqlcp} \leq l_{kklcp}$, REncoderSE starts storing from the $(l_{kklcp}+1)^{th}$ level like REncoderSS (necessary levels $(l_{kqlcp}, l_{kklcp}+1]$ are stored). When $l_{kqlcp} > l_{kklcp}$, REncoderSE starts storing from the $(l_{kqlcp}+1)^{th}$ level, but in the opposite direction. In this case, the $(l_{kqlcp}+1)^{th}$ level is regarded as the end level. By storing the necessary levels (levels below the $(l_{kqlcp})^{th}$ level), REncoderSE remains low FPR in correlated workloads.

[4]Define $lcp(x, y)$ as the length of LCP between x and y. The length of LCP between key and query[left,right] is $max(lcp(\text{key,left}), lcp(\text{key,right}))$

3.4 FPR Optimization Through Proposed Blacklist

In order to further reduce the FPR, we propose another optimization called *blacklist* based on the original REncoder, which is specially designed for multi-round queries. Multi-round queries are common for range queries, *e.g.*, a specified range may need to be periodically queried in many network scenarios [19, 23, 59], so improving its accuracy is very important in many practical applications. Note that the previously proposed REncoder, REncoderSS, and REncoderSE are mainly for single-round queries, and the blacklist in this section have no optimization effect on single-round queries. The basic data structure of the blacklist is a hash table. Next, we will show the related operations.

Insertion: For a given range R, if true is returned after querying Rencoder, we will try to find it on the disk: if we find that it does not actually exist, we need to insert all keys contained in R into the blacklist. The specific insertion process is as follows. First, we divide the range R into multiple sub-ranges (such as decomposition in the query described above); Second, we find the shortest common prefix length l among all sub-ranges; Finally, for each sub-range, we take the first l bits as the key, encode the remaining bits into a bitmap as the value, and insert them into the blacklist.

Example 1: Suppose a false positive occurs in the range [10100000, 10100101]. First, we divide it into two sub-ranges [10100000, 10100011] and [10100100, 10100101], whose common prefixes are 101000 and 1010010, with lengths of 6 and 7, respectively. Second, we get the shortest common prefix is 101000 with a length of 6. For the first sub-range, its key is the first 6 common bits (101000), and its value is encoded by the last two bits. Specifically, the last two bits include four values 00, 01, 10, and 11, using one-hot encoding: 00 corresponds to 0001, 01 corresponds to 0010, 10 corresponds to 0100, and 11 corresponds to 1000, so the value of the entire range is 1111. For the second sub-range, its key is the first 6 common bits (101001), and the next two bits have only two values 00 and 01, corresponding to 0001 and 0010, so the value of this range is 0011. Just insert them into the blacklist.

Query: The process of querying the blacklist can wait for the blacklist to be built after the first round of range queries in REncoder, or it can be performed at the same time as the above insertion. When querying range R for the second time, Rencoder returns true when querying a certain leaf node. Then, we try to query the leaf node from the blacklist: if it does exist, this is actually a

false positive and we should continue querying instead of reporting its existence. Generally, there are no false positives in the second round and all subsequent rounds of queries, because the blacklist has already been used in the first round of queries.

Example 2: Suppose for the range [10100000, 10100101], true is returned when the leaf node 10100000 is queried. Then, we try to query the leaf node from the blacklist. The key is 101000 and the value is 1111, which means that the suffix 00 exists, *i.e.*, 10100000 is in the blacklist and does not exist in the disk. Therefore, we should continue the query.

In summary, we further trade off reduction in FPR for multi-round range queries at the cost of a small memory footprint through blacklist optimization.

3.5 Future Work: Support for Float/Double Types.

In this section, we propose **Two-Stage REncoder** to support float/double types. For convenience, we only discuss the float type (the solution is similar for the double type). We only discuss positive keys, as negative keys can be converted to positive keys by adding the absolute value of the smallest key.

Strawman Solution. We multiply all float keys by a factor, and discard the decimal part. The factor is set as large as possible while ensuring that all multiplied float keys do not exceed $2^{64}-1$. In this way, we convert float keys into 64-bit integer keys, which are supported by REncoder. Since the decimal part is discarded, the FPR increases, especially when the stored keys and queries can only be distinguished by the decimal part.

Two-Stage REncoder. Float key consists of a sign bit, an 8-bit exponent, and a 23-bit mantissa. As discussed before, we ignore the sign bit, then the float key can be regarded as a 31-bit integer key. We design a Two-Stage REncoder to store the integer key. **In Stage 1**, we store the exponent. Storing starts from the 8^{th} level and goes up (the higher the level, the larger the range). Storing ends when P_1 reaches a predetermined threshold ($T_{exp} < 0.5$). **In Stage 2**, we store the mantissa. Storing starts from the 9^{th} level and goes down (the lower the level, the higher the precision). Storing ends when P_1 is close to 0.5. The query of Two-Stage REncoder is the same as REncoder. We can set T_{exp} according to dataset/workload to achieve better performance, which is left for future work.

4 Mathematical Analysis

In this section, we analyze the detail of the implementation of the algorithm and provide an error bound. Let $[a, b]$ be the range in *Verification* stage and $L_q = \log(b - a + 1)$ be the number of query levels. For convenience, we assume that:

1. The range in *Verification* stage consists of a complete binary tree, i.e. there exists some $s > 0$ satisfying $b - a = 2^s - 1, 2^s | a$.
2. The number of query levels shall be no more than the number of stored levels, i.e. $L_q \leq L_s$.
3. We always assume that the first $L - L_s$ bits of the key exists in the Bloom filter, so we just find a match for the last L_s bits.
4. When P_1 is not too small, whether every bit in the Bloom filter will be set to 1 is independent.

Table 4: Test of Independence in Bloom Filter

	$P_.$	$P_{.\|0}$	$P_{.\|1}$	$P_{.\|00}$	$P_{.\|01}$	$P_{.\|10}$	$P_{.\|11}$
0	0.5233	0.5250	0.5214	0.5367	0.5264	0.5121	0.5160
1	0.4767	0.4750	0.4786	0.4633	0.4736	0.4879	0.4840

Based on the assumptions above, false positive occurs if and only if all nodes from the root to the mini-tree are set to 1 and there exists a path to one of its leaves.

4.1 Overall Error Bound for REncoder

Lemma 1 *Let $\{a_n\}$ be a sequence with $a_1 = 1$, $a_{n+1} = 2pa_n - p^2a_n^2$, where $0 < p < 1$ is a constant. Then:*

1. *If $0 < p < \frac{1}{2}$, then a_n converges exponentially to 0.*
2. *If $p = \frac{1}{2}$, then $a_n = O(\frac{1}{n})$.*
3. *If $\frac{1}{2} < p < 1$, then $\lim_{n\to\infty} a_n = \frac{2p-1}{p^2}$.*

Theorem 1 *Let $p = P_1$. If there is no item in range $[a, b]$, then the probability that our algorithm reports false positive is bounded.*

$$P([a,b] \text{ reported false positive}) \leq (P_1^{L_s - L_q} \cdot a_{L_q})^k, \quad (1)$$

where k is the number of hash functions.

Proof If our algorithm reports false positive, then the query shall first enter the mini-tree, then find a path to one of its leaf. Since the number of queried levels is L_q and the number of stored levels is L_s, the query enters the mini-tree after $L_s - L_q$ steps, and this attempt succeeds if and only if all nodes here are set to 1. After entering the mini-tree, it shall find a path to one of its leaves. If we define a_n as the probability of finding a path when the height of mini-tree is n and l, r be the bit of the left and right son of the root, by induction we know that

$$\begin{aligned} a_{n+1} =& P(l + r = 1)a_n + P(l + r = 2)[1 - (1 - a_n)^2] \\ =& 2P_1(1 - P_1) \cdot a_n + P_1^2 \cdot (2a_n - a_n^2) \\ =& 2P_1 \cdot a_n - P_1^2 a_n^2. \end{aligned}$$

(2)

Hence a_n satisfies the equation in Lemma 1. Finally, we know that for one hash function h_i, the following inequality holds:

$$P([a,b] \text{ reported false positive by } h_i) \leq P_1^{L_s-L_q} \cdot a_{L_q}. \tag{3}$$

Assume that whether every hash function reports false positive is independent, we get

$$P([a,b] \text{ reported false positive}) \leq (P_1^{L_s-L_q} \cdot a_{L_q})^k. \tag{4}$$

4.2 Trade-off for Hash Functions and Stored Levels

However, we need to make some trade-offs in the algorithm. When there are too many hash functions, the P_1 will exceed 0.5, which can lead to the sharp increase of FPR. Also, while the increase of stored levels can decrease the number in the right hand side of Equation 1, it can increase P_1 as well. In this part, we analyze the relationships between number of hash functions, number of stored levels and P_1. We assume that we will adjust the memory to keep P_1 stable.

Lemma 2 *Let M denote the memory of Bloom filter and N denote the number of items inserted into the Bloom filter, then*

$$P_1 \leq \frac{kL_sN}{M}. \tag{5}$$

Proof Each insert operation will set at most L_s bits to 1 for every hash function. There are k hash functions and N items to be inserted, so

$$P_1 \leq \frac{kL_sN}{M}. \tag{6}$$

The lemma above shows that P_1 is approximately a linear function with respect to k and L_s. As a result, to keep P_1 constant without extra memory, we shall keep $k \cdot L_s$ nearly constant.

Theorem 2 *When both P_1 and kL_s are kept constant, the right hand side of Equation 1 is a monotonous increasing function with respect to k. As a result, the number of hash functions shall not be set too large.*

Proof We can figure out that

$$(P_1^{L_s-L_q} \cdot a_{L_q})^k = P_1^{kL_s} \cdot \left(\frac{a_{L_q}}{P_1^{L_q}}\right)^k \tag{7}$$

We can prove that $\lim_{n\to\infty} \frac{a_n}{P_1^n} = +\infty$. Hence the value above increases when k increase. Moreover, if we want to keep it small, k shall not be set too large.

The inferiority of more hash functions compared to more stored levels can be explained by the fact that every more hash function results in one more copy of every prefix, but one more stored levels will only add one bit into the Bloom filter for each item. However, simply increasing stored levels is not necessarily effective. We will analyze it in depth in Section 4.3.

Theorem 3 *Assume that P_1 is kept constant. For a given range $[a,b]$, to ensure that FPR is less than ε, our algorithm needs $O(N(k+\log\frac{1}{\varepsilon}))$ memory.*

Proof We require

$$P([a,b] \text{ reported false positive}) \leq (P_1^{L_s-L_q} \cdot a_{L_q})^k \leq \varepsilon$$
$$\Rightarrow L_s \geq L_q - \frac{\log\frac{1}{a_{L_q}}}{\log\frac{1}{P_1}} + \frac{\log\frac{1}{\varepsilon}}{k\log\frac{1}{P_1}}. \tag{8}$$

Hence,

$$M \approx \frac{kL_sN}{P_1} = \frac{kN}{P_1}\left(L_q - \frac{\log\frac{1}{a_{L_q}}}{\log\frac{1}{P_1}}\right) + \frac{N\log\frac{1}{\varepsilon}}{P_1\log\frac{1}{P_1}} \tag{9}$$
$$= O(N(k+\log\frac{1}{\varepsilon})).$$

Since we always use a limited number of hash functions, the asymptotic space complexity in Theorem 3 can be written as $O(N\log\frac{1}{\varepsilon})$, which perfectly demonstrates the overall better performance.

4.3 Analysis for More Complex Situation

In the proof above, we assume that every node from the root to leaf is set to 0 as to query range. However, this assumption can be problematic when some items inserted into RBF are close to the range in the query. These items share the same prefix with some items in the range and can set some nodes to 1 in advance. We define a *distance* as following:

$$d([a,b]) = \min_{\substack{a\leq x\leq b,\\ y\in keys}} \{k : x >> k = y >> k\} \tag{10}$$

Clearly $d([a,b]) = 0$ when $[a,b] \cap keys \neq \varnothing$. Also, if false positive never occurs, the last $d([a,b])$ nodes in the tree shall all be set to 0. So the distance measures the *difficulty* of false positive as the number of wrongly-set 1 in the tree shall be $d([a,b])$ when reporting false positive.

Theorem 4 *If $d([a,b]) > 0$, the right hand side of Equation 1 has a lower bound.*

$$P([a,b]\textit{reported false positive}) \leq \begin{cases} a^k_{d([a,b])} & (L_q \geq d([a,b])) \\ \left(P_1^{d([a,b])-L_q} \cdot a_{L_q}\right)^k & (L_q < d([a,b])) \end{cases} \quad (11)$$

Proof By the definition of d we know that $\exists y \in keys$ which shares the same $L_s - d([a,b])$ bits with the range $[a,b]$. Hence, false positive occurs when the last $d([a,b])$ bits in the mini-tree are set to 1. If $d([a,b]) \leq L_q$, the probability is just $a^k_{d([a,b])}$. If $d([a,b]) > L_q$, the probability can be figured out by replacing L_s with $d([a,b])$. Finally, we get

$$P([a,b]\text{reported false positive}) \leq \begin{cases} a^k_{d([a,b])} & (L_q \geq d([a,b])) \\ \left(P_1^{d([a,b])-L_q} \cdot a_{L_q}\right)^k & (L_q < d([a,b])) \end{cases} \quad (12)$$

The theorem above shows that despite the superiority of more stored levels compared to more hash functions, simply increasing stored levels is *not* an effective approach to lower error rate because finally L_s will be greater than $d([a,b])$ in this case. As a result, more hash functions still play a role in our algorithm.

5 Experimental Results

In this section, we illustrate the experimental results of REncoder. We compare three versions of REncoder with the SOTA range filters: SuRF, Rosetta, SNARF and Proteus. All the experiments are conducted based on LSM-tree.

We run the experiments on a server with 18-core CPU (36 threads, Intel CPU i9-10980XE @3.00 GHz), which have 128GB memory. The operating system is Ubuntu version 18.04 LTS. All the algorithms are implemented in C++ and built by g++ 9.3.0 and -O2 option. The hash functions we use are 32-bit Bob Hash [2] with random initial seeds. We use SIMD [8] to accelerate the process of inserting/extracting a bitmap into/from RBF.

5.1 Datasets and Workload

Synthetic Dataset: Synthetic dataset contains 50M 64-bit integer keys which are generated from uniform distribution.

SOSD Dataset: SOSD [29] is a benchmark for Learned Indexes. It contains four real datasets: **amzn** is the book sale data of amazon.com, **face** is user ID data of Facebook, **osmc** is uniformly sampled data of OpenStreetMap, **wiki** is edit timestamps of Wikipedia article. All of these datasets contain 200M 64-bit integer keys. We uniformly sample 10M keys from each of them for experiments. Ordered by skewness, there is wiki > face > amzn > osmc.

Workload: We generate four types of queries: range queries of range $2 \sim 32$ and $2 \sim 64$, correlated range queries and point queries. The number of each type of queries is 10M. For $2 \sim 32$ range queries, we first generate 10M integer keys from uniform distribution as left boundaries of the range queries. Then we randomly select an integer from 2 to 32 as the range size for each query. $2 \sim 64$ range queries and point queries are the same as $2 \sim 32$ range queries, except that the range sizes of $2 \sim 64$ range queries are randomly selected from 2 to 64, and the range sizes of point queries are set to 1. For correlated range queries, we first randomly select 10M keys from datasets, then we increment the keys by 32 and set them as left boundaries of the range queries. In this way, all queried ranges are very similar to stored keys. The range sizes of correlated range queries are randomly selected from 2 to 32. For each real dataset, we generate 1M real range queries. We randomly select 1M keys from the remaining 190M keys in the dataset, and set them as left boundaries of the range queries. The range sizes of real range queries are randomly selected from 2 to 32. Since a range filter is best evaluated by empty queries, all five types of queries above are set to empty.

5.2 Metrics

False Positive Rate (FPR): FPR measures the accuracy of range filters. In general, FPR means the ratio of the negatives that are incorrectly reported as positives to all negatives, it is defined as:

$$FPR = \frac{FP}{FP + TN} \quad (13)$$

where FP is the number of negatives that are incorrectly reported as positives, TN is the number of negatives that are correctly reported as negatives. For range filters, positive means the queried range contains stored item, while negative means the queried range does not contain stored item.

Filter Throughput: Filter throughput measures the probing speed of range filters. Its unit is million operations per second. (Mops/s).

Overall Throughput: Overall throughput measures the probing speed of queries using range filters. In experiments, we build a simulation environment with two-level storage. The range filters are stored in the first

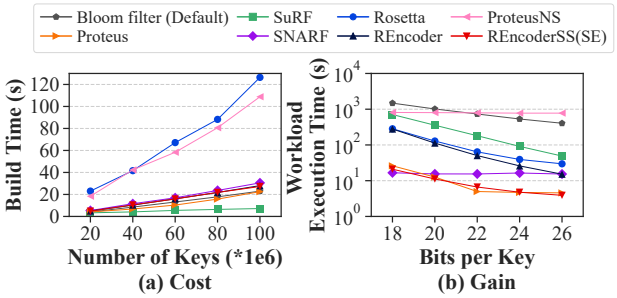

Fig. 5: Cost and gain of using REncoder in LSM-tree.

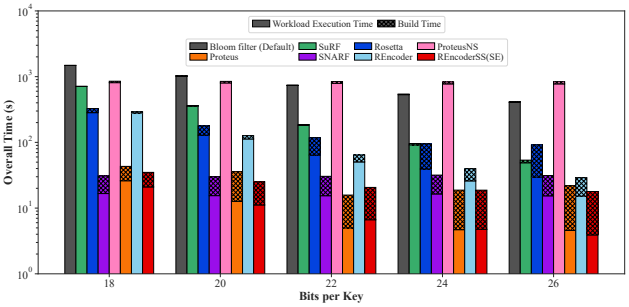

Fig. 6: Experimental comparisons on overall time.

level, while the items (Key-Value pairs) are stored in the second level. When a query coming, we first query the range filters in the first level, only when the range filters return positive, we access second level for the items. Overall throughput measures the speed of the entire process. Its unit is the same as filter throughput.

5.3 Experiments Settings

In experiments, we implement optimized version of REncoder, REncoderSS and REncoderSE, and their corresponding blacklist-optimized versions, denoted as REncoder-BL, REncoderSS-BL, REncoderSE-BL, respectively. In addition, we use SIMD to accelerate the process of inserting/extracting a bitmap into/from RBF. Specifically, we encode 8 successive prefixes into one bitmap of length 512. We can store/fetch the bitmap with a single memory access, thanks to AVX-512 of SIMD instruction sets. For SuRF, we use its mixed version, namely SuRF-Mixed. SuRF-Mixed stores both hashed key suffixes and real key suffixed. We allocate the same bits for two suffix types. For Rosetta and SNARF, we use its default setting. For Proteus, we use two versions: 1) sampling queries is allowed, and the design is determined by the CPFPR model, denoted as Proteus; 2) sampling queries is forbidden, and the default design (a prefix Bloom filter with a prefix length of 32) is used, denoted as **ProteusNS** (**Proteus** with **N**o **S**ampling). The memory allocated for each range filter is represented by bits per key (BPK). When dataset contains 50M keys and BPK=16, the memory allocated for each range filter is $16 \times 50 \times 10^6 = 8 \times 10^8$b $\approx$ 95.37MB. Due to the space limitation, we do not present specific statistics of range filters in following text, but summarize them in Table 1.

5.4 Experiments on Cost and Gain

In this section, we compare REncoder with LSM-tree's default filter (Bloom filter) and four SOTA range filters (SuRF, Rosetta, SNARF, Proteus & ProteusNS) in the simulation environment we built to show the cost and gain of using REncoder in LSM-tree. We use Synthetic Dataset and 2 $\sim$ 32 range queries. Note that Bloom filter handles range queries by sequentially checking the existence of all keys within the range.

Build Time (Figure 5(a)). We find that the build time of REncoder/REncoderSS(SE) is slightly slower than that of Bloom filter, SuRF, and Proteus, and faster than that of the other filters. Specifically, the results show that the build time of REncoder/REncoderSS(SE) is 1.22×/1.21×, 3.06×/3.03×, and 1.36×/1.35× slower than that of Bloom filter, SuRF, and Proteus on average, respectively, and 4.27×/4.32×, 1.09×/1.1×, and 3.8×/3.84× faster than that of Rosetta, SNARF, and ProteusNS on average, respectively. The build time of REncoder and all comparison filters increases linearly with the number of keys. For each key, although REncoder needs to insert several prefixes of the key, it can insert multiple prefixes simultaneously by using bitmaps and RBF to speed up its build, achieving an efficiency comparable to that of Bloom filter. Additionally, the cost in build time is negligible compared to the gain in query performance, as discussed below.

Workload Execution Time (Figure 5(b)). We find that the workload execution time of REncoderSS(SE) is the optimal among all comparison filters. Specifically, the results show that the workload execution time of REncoderSS(SE) is 87.8×, 29.2×, 11.5×, 1.67×, 83.4×, and 1.12× faster than that of Bloom filter, SuRF, Rosetta, SNARF, ProteusNS, and Proteus on average, respectively. REncoder/REncoderSS(SE)'s much fewer memory accesses (normally once) means higher throughput, and lower FPR means less I/Os, both of which play a large role in shortening workload execution time. It is worth noting that REncoderSS(SE) requires less workload execution time than that of REncoder because only fewer levels need to be queried.

Overall Time (Figure 6). Here, the overall time is obtained by adding the build time and workload execution time of the corresponding BPK. The results show that REncoderSS(SE)'s overall time performance is excellent and can even be optimal at some BPKs. The

build time only accounts for a small part of the overall time, although this percentage will increase as the BPK increases. Moreover, the degradation in build is negligible compared to the improvement in workload execution. In other words, the overhead of building range filters can be overshadowed by the improvement of query performance.

5.5 Experiments on Range Queries

In this section, we compare the performance of range filters in $2 \sim 32$ range queries and $2 \sim 64$ range queries using synthetic dataset.

FPR (Figure 7). *The FPR of REncoder(SS/SE) is the lowest or comparable to the lowest among all range filters no matter how the BPK changes.* For the $2 \sim 32$ range queries, when the BPK is 14, the FPR of SuRF and Rosetta is 0.041 and 0.029, respectively, while the one of REncoder reaches 0.027. As the BPK increases, the FPR of REncoder can reach 0.00016 which is comparable to 0.00011 of Rosetta, while the FPR of SuRF is 0.00064 at least. For the $2 \sim 64$ range queries, the FPR of REncoder also remains the lowest.

Filter Throughput (Figure 8(a)-(b)). *The filter throughput of REncoder(SS/SE) is much better than that of Rosetta and comparable to that of SuRF no matter how the BPK changes.* For the $2 \sim 32$ range queries, the filter throughput of each range filters generally remains stable with the change of BPK. The filter throughput of REncoder is between 5.37× and 6× higher than that of Rosetta. For the $2 \sim 64$ range queries, the filter throughput of REncoder is between 5.06× and 7.08× higher than that of Rosetta.

Overall Throughput (Figure 8(c)-(d)). *The overall throughput of REncoder(SS/SE) is higher than SuRF and Rosetta no matter how the BPK changes.* For the $2 \sim 32$ range queries, the overall throughput of REncoder is respectively between 4.56× and 6.51×, 2.11× and 2.39× higher than that of SuRF and Rosetta. For the $2 \sim 64$ range queries, the filter throughput of REncoder is respectively between 2.64× and 2.97×, 1.89× and 2.62× higher than that of SuRF and Rosetta.

Analysis. SuRF truncates part of nodes in the lower levels to save space, which may result in the loss of important information for range queries. While Rosetta and REncoder reserve these information through Bloom filters, and use additional queries to further guarantee the accuracy of the information. Therefore, Rosetta and REncoder achieve much lower FPR than SuRF. For filter throughput, SuRF performs much better than Rosetta because it uses a truncated trie internally. When the range query coming, SuRF only needs to traverse in the succinct trie which is very fast,

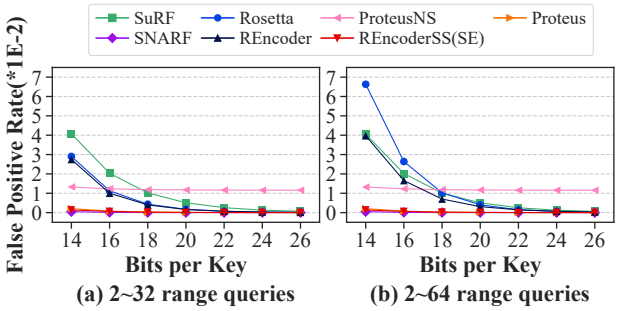

Fig. 7: FPR of range queries.

while Rosetta needs to perform many time-consuming queries to Bloom filters. In contrast, REncoder utilizes the locality of the queries to Bloom filters to achieve higher filter throughput than Rosetta while remaining low FPR. Overall throughput indicates the performance of range filters in practice. Since the speed of computations in first-level storage (*e.g.*, memory) are much faster than that of data fetching in second-level storage (*e.g.*, disk), although SuRF has higher filter throughput, it suffers in overall throughput because of more unnecessary data fetching in second-level storage caused by its higher FPR. In contrast, Rosetta and REncoder have higher overall throughput, thanks to their lower FPR. On the other hand, computations in first-level storage still take a non-negligible part in overall throughput. Therefore, REncoder has higher overall throughput than Rosetta because of its better performance in first-level storage. SNARF achieves low FPR by using a learned model, but the queries to compressed bit array severely limit the filter throughput. Proteus has both low FPR and high filter throughput, because the CPFPR model gives the optimal design by sampling queries. However, when sampling queries is forbidden, Proteus using default design (*i.e.*, ProteusNS) has much worse FPR than REncoder. Since both keys and queries are uniformly distributed, REncoderSS can achieve the same performance as REncoderSE, and we denote them as REncoderSS(SE). Compared with REncoder, REncoderSS(SE) stores higher levels that contain more significant information, leading to lower FPR and higher filter throughput. REncoderSS(SE) has the highest overall throughput among all range filters across all BPKs (except 22).

5.6 Experiments on Point Queries

In this section, we compare the performance of range filters in point queries using synthetic dataset. For the sake of fairness, we make Rosetta allocate memory according to $2 \sim 64$ range queries instead of point queries. In this way, Rosetta maintains the performance for range queries.

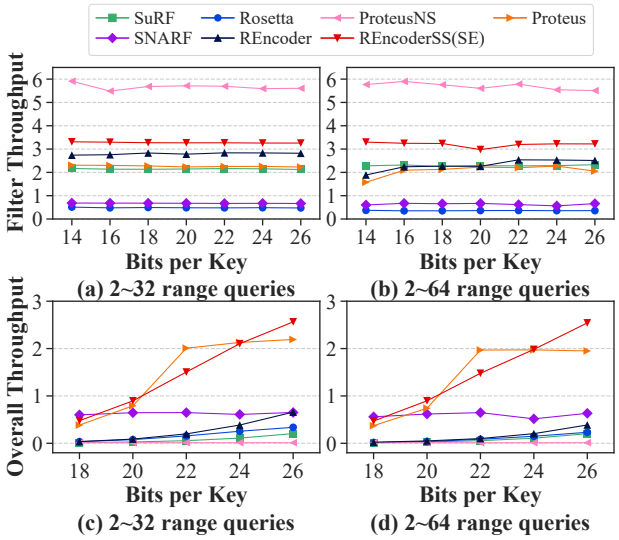

Fig. 8: Throughput of range queries.

FPR (Figure 9(a)). *REncoder(SS/SE) remains low FPR in point queries for all BPK settings.* With the increase of BPK, the FPR of SuRF is changed from 0.0101 to 0.000002, the FPR of Rosetta is changed from 0.0115 to 0.000038, while the FPR of REncoder is changed from 0.0014 to 0.000003.

Filter Throughput (Figure 9(b)). *REncoder(SS/SE) has slightly lower filter throughput than Rosetta.* The filter throughput of Rosetta is between 1.47× and 1.77× higher than that of REncoder.

Analysis. The FPR of SuRF, Rosetta and REncoder in point queries significantly decreases compared with range queries. For SuRF, its hashed key suffix provides additional reliable information for point queries which can help reduce FPR. For Rosetta and REncoder, they need fewer queries to Bloom filters in point queries than in range queries, thus their FPR which is the combination of the FPR of queries to Bloom filters is lower. REncoder still have much lower FPR than SuRF because of the accuracy provided by Bloom filters. However, Rosetta's FPR becomes higher than SuRF because it only queries the lowest level of Bloom filter and ignores the information stored in other Bloom filters. On the other hand, the filter throughput of SuRF, Rosetta and REncoder in point queries increase compared with range queries. For SuRF, compared with range queries, point queries perform much simpler traversal of its inner tries which greatly shortens the latency of queries. For Rosetta and REncoder, fewer queries to Bloom filters reduces computations for hash and raises overall performance. SNARF and Proteus perform similarly in point queries as they do in range queries because their structures are robust to different range sizes. REncoderSS and REncoderSE have the same performance

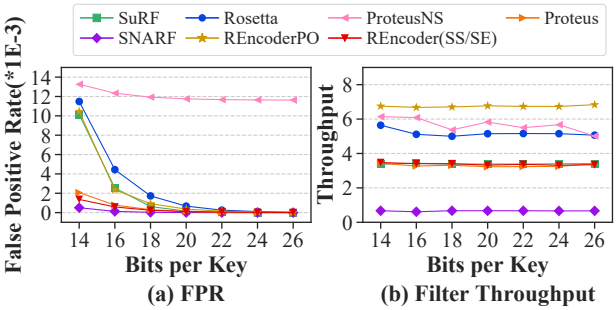

Fig. 9: Performance of point queries.

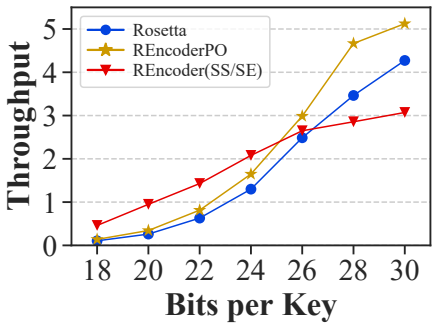

Fig. 10: Overall throughput of point queries.

as REncoder because higher levels and lower levels are equally important in point queries.

Optimization. Since Rosetta only queries the lowest level of Bloom filter, it has higher filter throughput than REncoder. Inspired by Rosetta, we propose an optimized version of **REncoder** for **PO**int queries, called **REncoderPO**. REncoderPO only queries the longest prefix of the key (*i.e.*, the key itself) for higher filter throughput at the cost of worse FPR. The overall throughput of Rosetta, REncoder and REncoderPO is shown in Figure 10. When BPK $<$ 26, all filters have relatively high FPRs, thus the overall throughput is dominated by queries in second-level storage. REncoder has the highest overall throughput because of its lowest FPR. When BPK $\geq$ 26, the FPRs of all filters are negligible, thus the overall throughput is dominated by queries in first-level storage (*i.e.*, filter throughput). REncoderPO has the highest overall throughput because of its highest filter throughput.

5.7 Experiments on Correlated Queries

In this section, we compare the performance of range filters in correlated queries using synthetic dataset.

FPR (Figure 11(a)). *REncoder(SE) remains low FPR in correlated queries for all BPK settings.* With the increase of BPK, the FPR of REncoder is changed from 0.027 to 0.00016, the FPR of Rosetta is changed from 0.029 to 0.00011, while the FPR of SuRF is always outrageous 1.

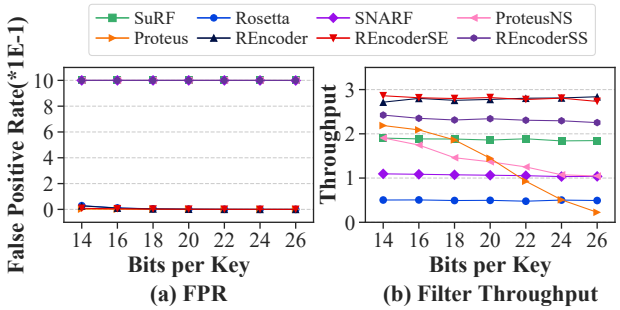

Fig. 11: Performance of correlated queries.

Filter Throughput (Figure 11(b)). *REncoder(SE) remains higher filter throughput than Rosetta in correlated queries for all BPK settings.* Specifically, the filter throughput of REncoder is between 5.39× and 5.88× higher than that of Rosetta.

Analysis. The FPR of SuRF reaches outrageous 1 even when BPK is 26. The reason is that SuRF truncates part of nodes in the lower levels, while the truncated nodes contain important information for distinguishing the queried key from the similar stored key. For Rosetta and REncoder, they are hardly affected by the distribution of the queries. The reason is that they both use Bloom filters to store the keys. Even if two keys are highly similar to each other, they are totally different after hash by Bloom filters. On the other hand, the filter throughput of SuRF decreases a little. When a correlated query coming, SuRF usually needs to traverse to the bottom level of the trie which is time consuming. Similar to FPR, the filter throughput of Rosetta and REncoder is also not affected. Note that REncoder still outperforms Rosetta. In addition to SuRF, the FPRs of SNARF, ProteusNS and REncoderSS also reach 1. The learned model of SNARF cannot distinguish between highly similar keys and queries. As for ProteusNS and REncoderSS, although both of them use Bloom filters, they do not store the lower levels of the segment tree. Therefore, they cannot distinguish between similar keys and queries either. Proteus remains low FPR, thanks to the appropriate design for correlated workload given by the CPFPR model. With the increase of BPK, the number of hash functions used by Proteus increases, leading to the decrease of its filter throughput. REncoderSE achieves the same performance as REncoder by selecting the end level (*i.e.*, storing the lower levels).

5.8 Experiments on Range Queries with Real Datasets

In this section, we compare the performance of range filters in range queries on real datasets.

FPR (Figure 12(a)-(d)). *REncoder(SS/SE) has the lowest or near-lowest FPR among all range filters on all datasets.* For the amzn dataset, the FPR of SuRF, Rosetta, and REncoder is respectively changed from 0.081 to 0.0042, 0.036 to 0.0001, and 0.023 to 0.0005. For the face dataset, the FPR of SuRF, Rosetta, and REncoder is respectively changed from 0.131 to 0.0055, 0.031 to 0.0001, and 0.021 to 0.0001. For the osmc dataset, the FPR of SuRF, Rosetta, and REncoder is respectively changed from 0.104 to 0.0021, 0.032 to 0.0001, and 0.028 to 0.0002. For the wiki dataset, the FPR of SuRF, Rosetta, and REncoder is respectively changed from 0.086 to 0.0786, 0.019 to 0.00006, and 0.011 to 0.00006.

Filter Throughput (Figure 12(e)-(f)). *REncoder(SS/SE) has higher filter throughput than that of Rosetta on all datasets.* Specifically, the filter throughput of REncoder is 4.7× faster than that of Rosetta on average.

Analysis. REncoder can adaptively choose the number of stored levels L_s of the segment tree, *i.e.*, make a space allocation, according to datasets. Therefore, it remains low FPR across all datasets. REncoderSS(SE) achieves lower FPR than REncoder, especially in relatively unskewed datasets (amzn and osmc). This is because in such datasets, keys and queries are nearly uniformly distributed, enabling REncoderSS(SE) to store higher levels (more significant information) than REncoder. The filter throughput of REncoder and REncoderSS(SE) is similar, and both decrease in relatively skewed datasets (face and wiki). This is because when keys and queries are similar, REncoder and REncoderSS(SE) need to query the Bloom filters more times to distinguish them. In summary, REncoder(SS/SE) remains great FPR and filter throughput across all real datasets.

5.9 Experiments on Range Queries with the Heavily Skewed Dataset

In this section, we compare the performance of range filters in range queries on a heavily skewed dataset. We use the open-source performance testing tool called Web Polygraph [48] to generate this Zipf dataset that follows the Zipf [44] distribution. In this dataset, 99% of the keys exist within 1% of the range ($0 \sim 2^{64}$), while the remaining 1% of the keys belong to the remaining 99% of the range.

FPR (Figure 13(a)). We find that REncoder has the lowest or near-lowest FPR among all range filters for all BPK settings on the Zipf dataset. Specifically, the FPR of SuRF, Rosetta, SNARF, and REncoder is changed from 0.058 to 0.00091, 0.044 to 0.00012, 0.096 to 0.0019, 0.019 to 0.00066, respectively; and the FPR of REn-

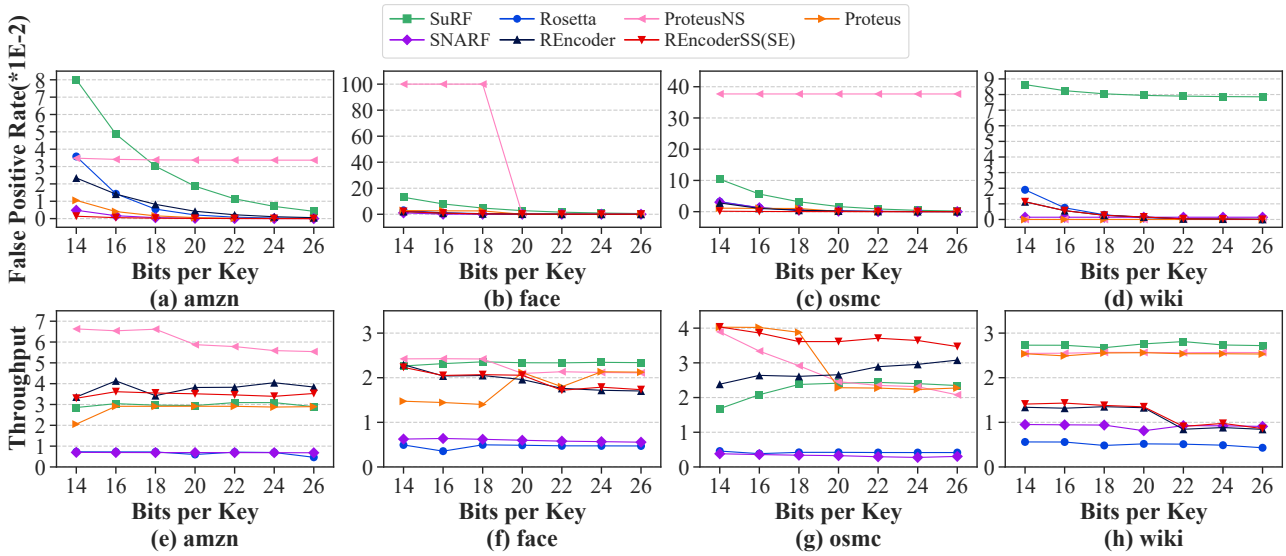

Fig. 12: Performance of range queries on real datasets.

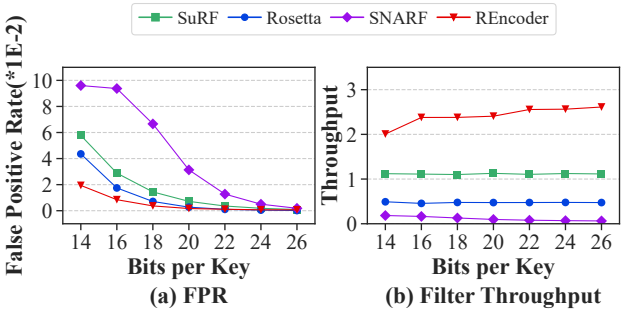

Fig. 13: Performance of range queries on the Zipf dataset.

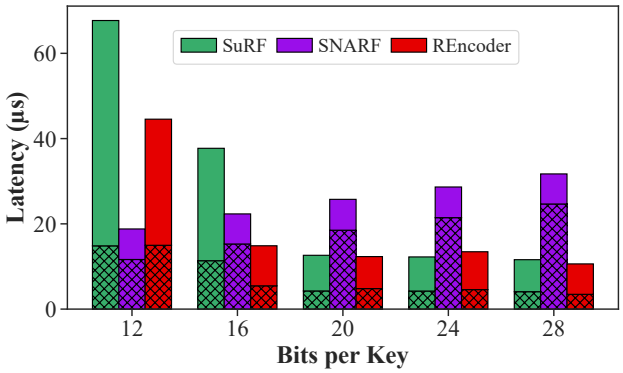

Fig. 14: The latency of range queries on RocksDB.

coder is 3.21×, 2.03×, and 8.61× lower than that of SuRF, Rosetta, and SNARF on average, respectively.

Filter Throughput (Figure 13(b)). We find that REncoder has significantly the highest filter throughput for all BPK settings on the Zipf dataset. Specifically, the filter throughput of REncoder is 2.17×, 5.09×, and 21.54× higher than that of SuRF, Rosetta, and SNARF on average, respectively.

Analysis. Compared to those on uniform distributions, the performance of Rencoder, SuRF, and Rosetta only slightly degrades on highly skewed distributions. While SNARF's degrades significantly, mainly because its learning model clearly degrades on highly skewed distributions.

5.10 System Experiments for Integration into RocksDB

In this section, we deploy the sampling-free REncoder along with SOTA SuRF and SNARF to the SOTA LSM-tree engine RocksDB [6], and conduct range query and point query experiments on real datasets and mixed workloads, respectively. We use a RocksDB setup that includes an LSM-tree with a size ratio of 10 and a lazy leveling compaction policy. We directly measure the latency as the evaluation metric.

5.10.1 Experiments on real datasets

In these experiments, we sample 10M keys and 1M queries from the osmc dataset, ensuring that all queries are empty, and the query range is $2 \sim 64$. For the experimental results in Figures 14-15, the shaded in each bar is the latency of the range filters, while the rest can be approximated as the latency of actually accessing the hard disk with false positives, *i.e.*, latency of the range filters (RFL) + latency due to false positives (FPL) ≈ actual overall latency on the system (SOL), for ease of analysis.

Range Query Latency (Figure 14): We find that the SOL and RFL of REncoder on RocksDB are both optimal or sub-optimal among all range filters for range

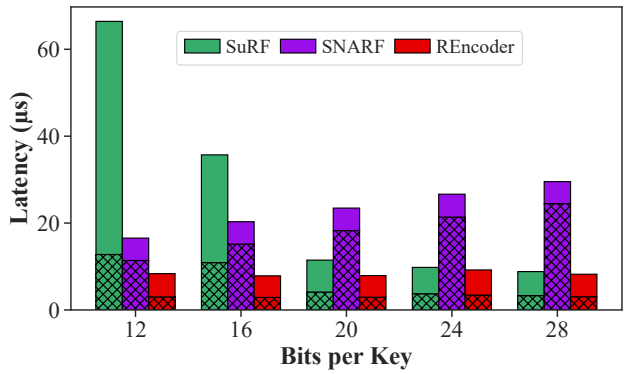

Fig. 15: The latency of point queries on RocksDB.

query task. Specifically, the SOL (or RFL) of SuRF, SNARF, and REncoder is changed from 67.70 to 11.59 (14.84 to 4.08), 18.80 to 31.71 (11.64 to 24.66), and 44.55 to 10.60 (14.97 to 3.47), respectively; and the SOL (or RFL) of REncoder is 1.48× (1.16×) and 1.33× (2.75×) lower than that of SuRF and SNARF on average, respectively.

Point Query Latency (Figure 15): We find that the SOL and RFL of REncoder on RocksDB are both optimal among all range filters for point query task. Specifically, the SOL (or RFL) of SuRF, SNARF, and REncoder is changed from 66.42 to 8.83 (12.77 to 3.30), 16.54 to 29.54 (11.39 to 24.47), and 8.36 to 8.22 (3.00 to 3.04), respectively; and the SOL (or RFL) of REncoder is 3.18× (2.28×) and 2.80× (5.93×) lower than that of SuRF and SNARF on average, respectively.

Analysis. For range queries on RocksDB, the filter speed (shaded part) of REncoder is much faster than that of SNARF, with the larger BPK being more obvious; compared to that of SuRF, the FPR of REncoder is lower, so there are fewer redundant I/O queries, with the smaller BPK being more obvious; In short, REncoder performs best overall. For point queries on RocksDB, REncoder adapts better to point queries than the other two algorithms, and the advantage is more significant when the BPK is small.

5.10.2 Experiments on mixed workloads

In these experiments, we sample 10M keys from the osmc dataset and insert them in advance, then sample 9M queries and 1M keys, and randomly perform insertion query operations, *i.e.*, with an insertion query ratio of 1:9, and the query range is $2 \sim 64$.

Total Latency (Figure 16(a)): We find that the total latency of REncoder on mixed workloads for RocksDB is either optimal or sub-optimal among all range filters for all BPK settings. Specifically, the total latency of SuRF, SNARF, and REncoder is changed from 32.79 to 11.75, 14.99 to 15.68, and 22.32 to 8.71, respectively; and the total latency of REncoder is 1.42× and 1.31× lower than that of SuRF and SNARF on average, respectively.

Insertion Latency (Figure 16(b)): We find that the insertion latency of REncoder on mixed workloads for RocksDB can remain sub-optimal among all range filters when BPK is high. Specifically, the insertion latency of SuRF, SNARF, and REncoder is changed from 22.70 to 19.21, 20.13 to 20.34, and 26.36 to 20.25, respectively; and the insertion latency of REncoder is 1.08× and 1.07× higher than that of SuRF and SNARF on average, respectively.

Query Latency (Figure 16(c)): We find that the query latency of REncoder on mixed workloads for RocksDB is either optimal or sub-optimal among all range filters for all BPK settings. Specifically, the query latency of SuRF, SNARF, and REncoder is changed from 33.92 to 10.92, 14.41 to 15.16, and 21.88 to 7.42, respectively; and the query latency of REncoder is 1.52× and 1.39× lower than that of SuRF and SNARF on average, respectively.

Analysis. In terms of insertion speed, when the BPK is small, in order to ensure the FPR, REncoder performs layer-by-layer insertion to realize the optimal memory allocation, thus the insertion latency is larger; when the BPK is large, REncoder can insert multiple layers at a time, greatly improving the insertion speed, which is almost consistent with that of the other two algorithms. In terms of query, SNARF basically reaches the optimal FPR when BPK=12, and the latency bottleneck lies in the filter speed. As BPK increases, the FPR of REncoder and SuRF decreases, and the query latency decreases accordingly. Meanwhile, since the filter speed of REncoder is faster, it remains optimal for the same FPR. Taken together, except for the case of limited memory (BPK=12), the total latency of REncoder is always optimal.

5.11 Experiments on Blacklist Evaluation

In this section, we conduct experiments to evaluate the combined effects of the blacklist (BL) optimization proposed in Section 3.4. Although the proposed BL occupies a small amount of memory, it still cannot be ignored for a fair comparison. Therefore, the 3 BL-optimized schemes (REncoder-BL, REncoderSS-BL and REncoderSE-BL) will pre-run one round separately to obtain the size of the BL used by each. Here, we provide the specific values of the BL sizes used by the three BL versions under different BPKs, as shown in Table 5. These will be subtracted from the preset memory sizes of the 3 original REncoder versions (REncoder, REncoderSS and REncoderSE), respectively, to ensure

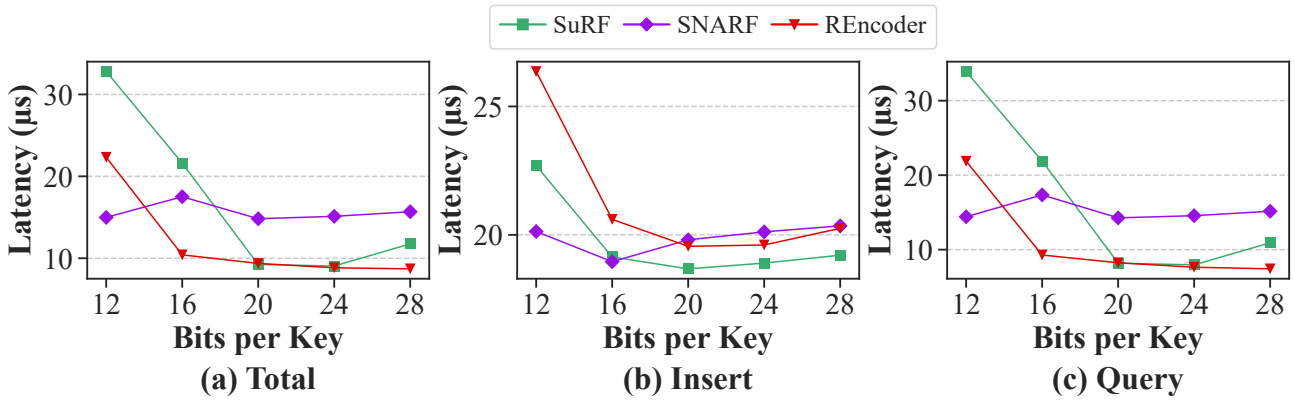

Fig. 16: The latency of mixed workloads on RocksDB.

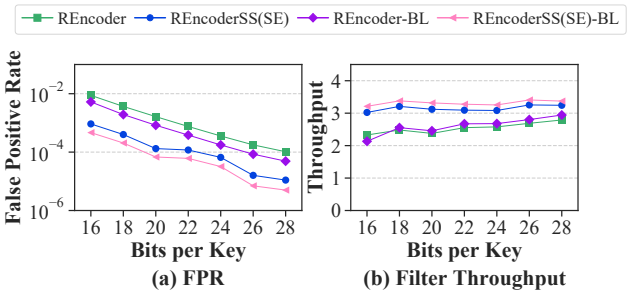

Fig. 17: Performance of range queries with BL optimization (overall).

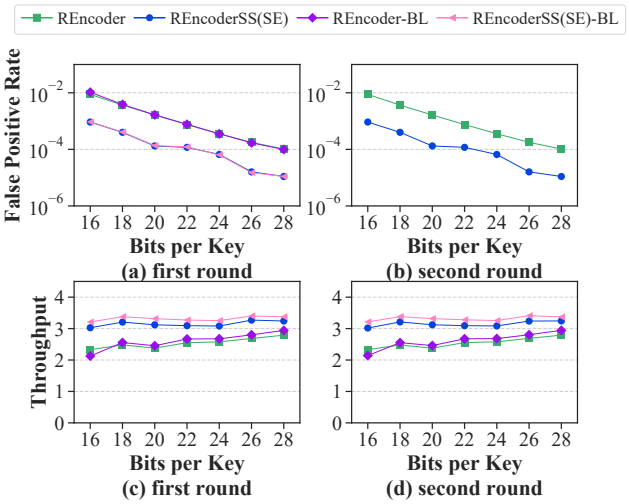

Fig. 18: Performance of range queries with BL optimization (separate rounds).

the same memory space before and after. In addition, except for the correlated query experiments in Section 5.11.6, REncoderSS-BL and REncoderSE-BL are exactly the same in the uniformly distributed experiments in Sections 5.11.1 to 5.11.5, so they are uniformly represented as REncoderSS(SE)-BL.

5.11.1 Overall evaluation of range queries

Similar to Section 5.5, but we perform 2-round queries under the same workload and take the average to compare the performance of all REncoder versions in $2 \sim 32$ range queries using the synthetic dataset.

FPR (Figure 17(a)): *No matter how the BPK changes, the FPR of the 3 BL-optimized versions is obviously lower than that of their respective BL-free versions, respectively.* Specifically, the FPR of REncoder-BL and REncoderSS(SE)-BL is on average 1.78× and 1.98× lower than that of REncoder and REncoderSS(SE), respectively.

Filter Throughput (Figure 17(b)): *The filter throughput of the 3 BL-optimized versions is slightly higher than that of their respective BL-free versions for all BPK settings, respectively.* Specifically, the filter throughput of REncoder-BL and REncoderSS(SE)-BL is on average 1.03× and 1.05× higher than that of REncoder and REncoderSS(SE), respectively.

Analysis. After being optimized by the BL, the FPR of the 3 BL-optimized versions is nearly 50% lower than the corresponding versions without the BL, and their throughput is also slightly improved because the lower FPR means that existence can be determined earlier.

5.11.2 Separate rounds of range queries

The setup is similar to the above Section 5.11.1, but the experimental results of the 2-round queries are shown separately to observe the effects of the BL itself more intuitively.

FPR (Figure 18(a)-(b)): *We find that all 3 BL-optimized versions have no advantage in FPR in the first round of queries but a significant advantage in FPR in the second round of queries compared to the FPR of their respective BL-free versions, respectively.* Specifically, the FPR of REncoder-BL and

Table 5: Blacklist memory overhead (KB) for 3 versions of REncoder.

BL-Optimized Version	BPK						
	16	18	20	22	24	26	28
REncoder-BL	907.1	379.3	169.9	78.6	37.0	18.2	10.7
REncoderSS-BL	327.2	190.2	12.8	11.5	6.5	1.6	1.1
REncoderSE-BL	327.2	190.2	12.8	11.5	6.5	1.6	1.1

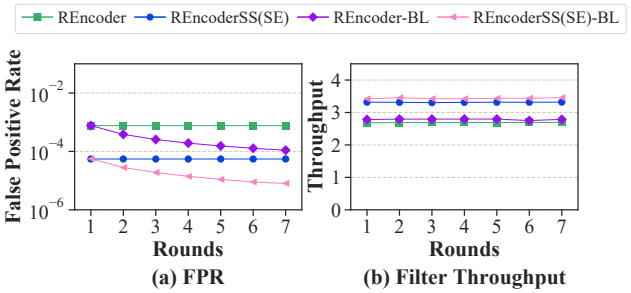

Fig. 19: Performance of range queries with BL optimization (multiple rounds).

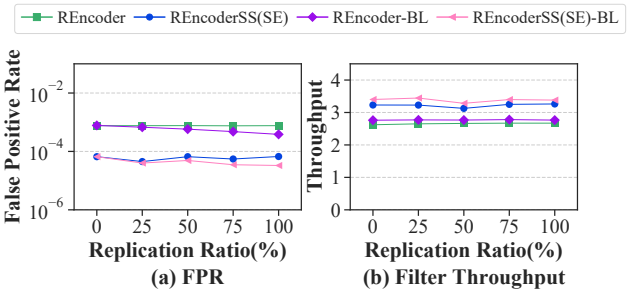

Fig. 20: Performance of range queries with BL optimization (replication query ratio).

REncoderSS(SE)-BL in the second round of queries is all 0, which is significantly better than that of REncoder and REncoderSS(SE), with average values of 0.0022 and 0.00024, respectively.

Filter Throughput (Figure 18(c)-(d)): *The filter throughputs of REncoder-BL and REncoderSS(SE)-BL are both slightly higher than those of REncoder and REncoderSS(SE), respectively, with indistinguishable differences in the first and second rounds of queries.*

Analysis. The BL does not work in the first round of queries, and all false positives are subsequently inserted into the BL, so there are no more false positives in the second round of queries.

5.11.3 Multiple rounds of range queries

The setup is similar to Section 5.11.1, but with the BPK fixed to 22 and the number of rounds in the workload set as the independent variable.

FPR (Figure 19(a)): *The FPR of REncoder-BL and REncoderSS(SE)-BL gradually decreases as the number of rounds increases, while that of REncoder and REncoderSS(SE) remains unchanged.* Specifically, the FPR of REncoder-BL and REncoderSS(SE)-BL in the seventh round of queries is 6.98× and 7× lower than that in the first round of queries, respectively.

Filter Throughput (Figure 19(b)): *There is no observable change in the filter throughput of the six algorithms as the number of rounds in the workload changes.*

Analysis. For multi-round query scenarios, as the number of rounds increases, the improvement of FPR by the BL becomes more significant, as expected.

5.11.4 Replication query ratio of range queries

The setup is similar to Section 5.11.1, but with BPK fixed at 22, and the workload of the second round is not exactly the same as that of the first round, with the replication query ratio set as the independent variable. Here, the replication query ratio refers to the proportion of queries in the second round of queries that are the same as those in the first round.

FPR (Figure 20(a)): *The FPR of REncoderSS(SE) fluctuates with the increase of the replication ratio of the second round workload, and that of REncoder remains almost unchanged, while the overall FPR of REncoder-BL and REncoderSS(SE)-BL shows a downward trend.*

Filter Throughput (Figure 20(b)): *The filter throughput of REncoder and REncoder-BL has almost no change as the replication ratio of the second round workload increases, while the overall filter throughput of REncoderSS(SE) and REncoderSS(SE)-BL shows slight fluctuations.*

Analysis. In actual scenarios, it is always unlikely that the workload of the second round will be completely consistent with that of the first round (*i.e.*, Replication Ratio = 100%), and may only be partially consistent. If this ratio is 100%, the BL optimizes FPR optimally. Even if the ratio is only 25%, the BL can still work.

5.11.5 Empty query ratio of range queries

The setup is similar to Section 5.11.1, but with BPK fixed at 22 and the empty query ratio in the same 2-round workload is set as the independent variable.

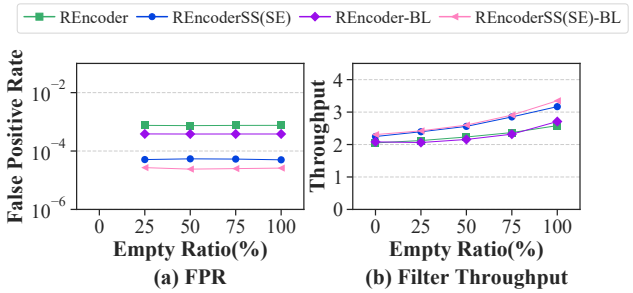

Fig. 21: Performance of range queries with BL optimization (empty query ratio).

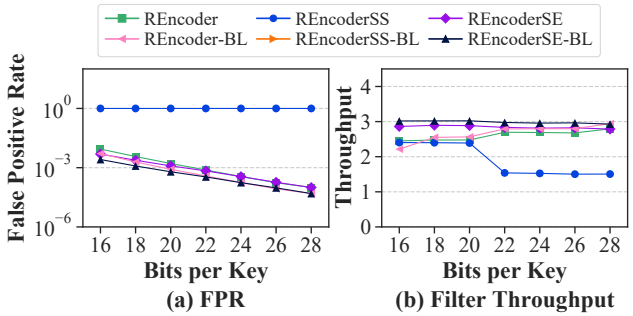

Fig. 22: Performance of correlated queries with BL optimization.

FPR (Figure 21(a): *The FPR of these six algorithms does not change significantly with the increase of the empty ratio in the second round workload.*
Filter Throughput (Figure 21(b): *The filter throughput of the six algorithms increases with the empty ratio in the second round of workload.*
Analysis. This experiment proves the stability of the BL for optimizing FPR. The filter throughput trends are explained as follows. When the ratio is 0, it means that all the keys to be checked exist, and the REncoder series needs to be queried the most times: the existence can only be determined by checking the bottom layer. On the contrary, if the ratio is 100%, the number of queries required is the least and the query speed is the fastest.

5.11.6 Correlated queries

The setup is similar to Section 5.7, except 2-round queries are run under the same workload. Note that REncoderSS-BL cannot work because the FPR of REncoderSS in correlated queries is close to 1 (see Section 5.7 and Figure 11(a)).
FPR (Figure 22(a): *Except for REncoderSS and its corresponding REncoderSS-BL, which do not function properly in this scenario, the FPR of the remaining 2 BL-optimized versions is lower than that of their respective BL-free versions, respectively.* Specifically, the FPR of REncoder-BL and REncoderSE-BL is on average 1.78× and 1.93× lower than that of REncoder and REncoderSE, respectively.
Filter Throughput (Figure 22(b): *The filter throughput of the remaining 2 BL-optimized versions is slightly higher than that of their respective BL-free versions, respectively.* Specifically, the filter throughput of REncoder-BL and REncoderSE-BL is on average 1.02× and 1.05× higher than that of REncoder and REncoderSE, respectively.
Analysis. This experiment proves that BL is also effective/robust in different distributions, and further highlights the role and significance of the proposed REncoderSE, because REncoderSS is unstable in this scenario and causes BL to shut down.

6 Conclusion

In this paper, we introduce REncoder, a novel range filter with great space-time efficiency and accuracy. The key idea is taking advantage of the locality to accelerate queries without affecting accuracy. It has theoretical error bound and supports various workloads. The experimental results show the superiority of REncoder compared with the state-of-the-arts. Finally, we specifically propose a general blacklist optimization framework for three original REncoder versions to cope with multi-round range queries, and experimentally validate that the three blacklist optimization versions can achieve a substantial improvement in FPR with even a slight increase in throughput.

Acknowledgements This work was supported in part by the National Key R&D Program of China (No. 2022YFB2901504), in part by the China Postdoctoral Science Foundation (No. 2023TQ0010, GZC20230055, 2024M750102), and in part by the National Natural Science Foundation of China (NSFC) (No. U20A20179, 62372009).

References

1. Apache. Accumulo. `https://accumulo.apache.org`.
2. BOB Hash website. `http://burtleburtle.net/bob/hash/evahash.html`.
3. CockroachLabs. CockroachDB. `https://github.com/cockroachdb/cockroach`.
4. Dgraph. Badger Key-value DB in Go. `https://github.com/dgraphio/badger`.
5. Facebook. MyRocks. `http://myrocks.io`.
6. Facebook. RocksDB. `https://github.com/facebook/rocksdb/`.
7. Google LevelDB. `https://github.com/google/leveldb`.
8. Intel instructions. `https://software.intel.com/sites/landingpage/IntrinsicsGuide`.
9. Source code related to REncoder. `https://github.com/Range-Filter/REncoder`.
10. D. Abadi, P. Boncz, S. H. Amiato, S. Idreos, and S. Madden. *The design and implementation of modern column-oriented database systems.* Now Hanover, Mass., 2013.

11. K. Alexiou, D. Kossmann, and P.-Å. Larson. Adaptive range filters for cold data: Avoiding trips to siberia. *Proceedings of the VLDB Endowment*, 6(14):1714–1725, 2013.
12. S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in asterixdb. *Proceedings of the VLDB Endowment*, 7(10):841–852, 2014.
13. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.
14. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
15. A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.
16. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.
17. D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, jun 1979.
18. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
19. A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-ring: an efficient and robust p2p range index structure. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 223–234.
20. N. Dayan, M. Athanassoulis, and S. Idreos. Optimal bloom filters and adaptive merging for lsm-trees. *ACM Transactions on Database Systems (TODS)*, 43(4):1–48, 2018.
21. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *IEEE Computer Society Non-profit Org. US Postage PAID Silver Spring, MD*, 2007.
22. S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 201–212, 2003.
23. J. Gao and P. Steenkiste. An adaptive protocol for efficient support of range queries in dht-based systems. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP)*, pages 239–250, 2004.
24. M. Goswami, A. Grønlund, K. G. Larsen, and R. Pagh. Approximate range emptiness in constant time and optimal space. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 769–775. SIAM, 2014.
25. G. Graefe and H. Kuno. Modern b-tree techniques. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1370–1373. IEEE, 2011.
26. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.
27. G. Jacobson. Space-efficient static trees and graphs. In *30th annual symposium on foundations of computer science*, pages 549–554. IEEE Computer Society, 1989.
28. T. Kahveci and A. Singh. Variable length queries for time series data. In *Proceedings 17th International Conference on Data Engineering*, pages 273–282. IEEE, 2001.
29. A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. Sosd: A benchmark for learned indexes. *arXiv preprint arXiv:1911.13014*, 2019.
30. A. Kirsch, M. Mitzenmacher, and G. Varghese. Hash-based techniques for high-speed packet processing. In *Algorithms for Next Generation Networks*, pages 181–218. Springer, 2010.
31. E. R. Knorr, B. Lemaire, A. Lim, S. Luo, H. Zhang, S. Idreos, and M. Mitzenmacher. Proteus: A self-designing range filter. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1670–1684, 2022.
32. H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas. Coconut palm: Static and streaming data series exploration now in your palm. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1941–1944, 2019.
33. A. Kyrola and C. Guestrin. Graphchi-db: Simple design for a scalable graph database system–on just a pc. *arXiv preprint arXiv:1403.0701*, 2014.
34. A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
35. K. Li and G. Li. Approximate query processing: What is new and where to go? a survey on approximate query processing. *Data Science and Engineering*, 3:379–397, 2018.
36. Q. Liu, L. Zheng, Y. Shen, and L. Chen. Stable learned bloom filters for data streams. *Proceedings of the VLDB Endowment*, 13(12):2355–2367, 2020.
37. C. Luo and M. J. Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.
38. S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, and S. 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*, pages 2071–2086, 2020.
39. D. P. Mehta and S. Sahni. *Handbook of data structures and applications*. Chapman and Hall/CRC, 2004.
40. G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. 1966.
41. B. Mößner, C. Riegger, A. Bernhardt, and I. Petrov. bloomrf: On performing range-queries in bloom-filters with piecewise-monotone hash functions and prefix hashing. In *Proceedings of the 26th International Conference on Extending database Technology (EDBT)*, volume 26, pages 131–143, 2023.
42. P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.
43. Y. Peng, J. Guo, F. Li, W. Qian, and A. Zhou. Persistent bloom filter: Membership testing for the entire history. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1037–1052, 2018.
44. D. M. Powers. Applications and explanations of zipf's law. In *Proceedings of the Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning (NeMLaP3/CoNLL '98)*, pages 151–160, 1998.
45. F. P. Preparata and M. I. Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.
46. K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. Slimdb: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment*, 10(13):2037–2048, 2017.
47. J. Roozenburg. A literature survey on bloom filters. *Research Assignment, November*, 2005.
48. A. Rousskov and D. Wessels. High-performance benchmarking with web polygraph. *Software: Practice and Experience*, 34(2):187–211, 2004.
49. R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. *Proceedings of the VLDB Endowment*, 1(1):526–537, 2008.
50. T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. Technical report, University of Maryland, 1987.
51. S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys & Tutorials*, 14(1):131–155, 2011.
52. K. Vaidya, S. Chatterjee, E. Knorr, M. Mitzenmacher, S. Idreos, and T. Kraska. Snarf: a learning-enhanced range filter. *Proceedings of the VLDB Endowment*, 15(8):1632–1644, 2022.
53. P. K. Vairam, P. Kumar, C. Rebeiro, and K. Veezhinathan. Fadingbf: A bloom filter with consistent guarantees for online applications. *IEEE Transactions on Computers*, 2020.
54. H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas. Coconut: sortable summarizations for scalable indexes over static and streaming data series. *The VLDB Journal*, 28(6):847–869, 2019.
55. Z. Wang, Z. Zhong, J. Guo, Y. Wu, H. Li, T. Yang, Y. Tu, H. Zhang, and B. Cui. Rencoder: A space-time efficient range filter with local encoder. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 2036–2049, 2023.
56. Y. Wu, J. He, S. Yan, J. Wu, T. Yang, O. Ruas, G. Zhang, and B. Cui. Elastic bloom filter: Deletable and expandablefilter using elastic fingerprints. *IEEE Transactions on Computers*, 2021.
57. R. Xie, M. Li, Z. Miao, R. Gu, H. Huang, H. Dai, and G. Chen. Hash adaptive bloom filter. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 636–647. IEEE, 2021.
58. T. Yang, A. X. Liu, M. Shahzad, D. Yang, Q. Fu, G. Xie, and X. Li. A shifting framework for set queries. *IEEE/ACM Transactions on Networking*, 25(5):3116–3131, 2017.
59. Y. Yi, R. Li, F. Chen, A. X. Liu, and Y. Lin. A digital watermarking approach to secure and precise range query processing in sensor networks. In *2013 Proceedings IEEE INFOCOM*, pages 1950–1958, 2013.
60. 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 *Proceedings of the 2018 International Conference on Management of Data*, pages 323–336, 2018.