Seesaw Counting Filter: An Efficient Guardian for Vulnerable Negative Keys During Dynamic Filtering

Meng Li
Deyi Chen
menson@smail.nju.edu.cn
chendeyi@smail.nju.edu.cn
Nanjing University
Nanjing, China

Haipeng Dai
Rongbiao Xie
haipengdai@nju.edu.cn
rongbiaoxie@smail.nju.edu.cn
Nanjing University
Nanjing, China

Siqiang Luo
siqiang.luo@ntu.edu.sg
Nanyang Technological University
Singapore

Rong Gu
gurong@nju.edu.cn
Nanjing University
Nanjing, China

Tong Yang
yang.tong@pku.edu.cn
Peaking University
Beijing, China

Guihai Chen
gchen@nju.edu.cn
Nanjing University
Nanjing, China

ABSTRACT

Bloom filter is an efficient data structure for filtering negative keys (keys not in a given set) with substantially small space. However, in real-world applications, there widely exist vulnerable negative keys, which will bring high costs if not being properly filtered, especially when positive keys are added/deleted dynamically. To address the problem, we propose SeeSaw Counting Filter (SSCF), which is innovated with encapsulating the vulnerable negative keys into a unified counter array named seesaw counter array, and dynamically modulating (or varying) the applied hash functions to guard the encapsulated keys from being misidentified. Moreover, we propose ada-SSCF to handle the scenarios where the vulnerable negative keys cannot be obtained in advance. We extensively evaluate our SSCF, which shows that SSCF outperforms the cutting-edge filters by 3× on averages regarding accuracy while ensuring a low operation latency. All source codes are in [2].

CCS CONCEPTS

• **Information systems** → *Web indexing*; *Point lookups*; *Spam detection*; • **Networks** → *Network monitoring*.

KEYWORDS

Bloom filter, Probabilistic Data Structure, Hash Modulation

ACM Reference Format:
Meng Li, Deyi Chen, Haipeng Dai, Rongbiao Xie, Siqiang Luo, Rong Gu, Tong Yang, and Guihai Chen. 2022. Seesaw Counting Filter: An Efficient Guardian for Vulnerable Negative Keys During Dynamic Filtering. In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3485447.3511996

WWW '22, April 25–29, 2022, Virtual Event, Lyon, France

ACM ISBN 978-1-4503-9096-5/22/04…$15.00
https://doi.org/10.1145/3485447.3511996

1 INTRODUCTION

Bloom filters are used widely. Designed for approximate membership testing problem, Bloom filter can reduce unnecessary access to the whole collection of keys, which is critical to systems like database and networking, especially considering the increasing data volume nowadays. With only a bit array as the underlying data structure, Bloom filter has been widely used due to its space efficiency and (insert/query) operation elegance, which only involves several times of hash function computation and memory accesses. Specifically, at insertion time, Bloom filter maps a positive key with k hash functions to k different bits and sets them to 1. At query time, a key is said to be in the filter if all its k mapped bits are 1. With such simple operations, Bloom filter provides one-side error probability. Specifically, only negative keys (*i.e.*, keys not in the given set) will be mistakenly identified to be (false) positive with bounded probability, while all positive keys (*i.e.*, keys in the given set) will be correctly identified. The probability of generating a false positive is denoted as false positive rate (FPR). Considering its performance and elegance, Bloom filters have been widely used in many applications, such as networking and database [11, 12, 14, 15, 18, 24].

Dynamicity is an emerging need. In the past decades, Bloom filter has been extensively studied for static set filtering, where the stored data is static and remains unchanged. For example, the Bloom filter is built for sorted string tables [13] in KV-stores like LevelDB [7]. However, there also exist important scenarios where data are inserted and deleted dynamically. To process dynamic set filtering, a variant named Counting Bloom filter (CBF) is proposed by replacing the underlying bit array with a counter array [12].

The costs of different filtering errors can be very skewed. One hidden assumption behind (Counting) Bloom filter is that all negative keys are treated identically [6], which implies that misidentifying of different items brings the same cost. However, the recent works indicate that the cost of filtering error of different negative keys can be significantly different or even very skewed [6, 10, 26]. For example, in the application of building a URL blacklist with Bloom filter to block malicious URLs, a given URL will first be checked if it is in the Bloom filter, and if yes, a new request is generated to validate its safety [10]. However, a significantly large cost will be brought when certain common URLs are misidentified.

Table 1: Comparison Among Filters

Filter Variants	Cost-efficiency	Dynamicity	One-pass Building	Prior Knowledge
Counting Bloom Filter [5, 12]	×	✓	✓	None
Learned Filter [9, 17, 25]	×	×	×	Positive & Negative keys
Weighted Bloom Filter [6]	✓	Partial	✓	Positive & Negative keys
Stacked Filter [10]	✓	Partial	×	Positive & Negative keys
HABF [26]	✓	×	×	Positive & Negative keys
SSCF/Ada-SSCF	✓	✓	✓	Vulnerable Negative Keys/None

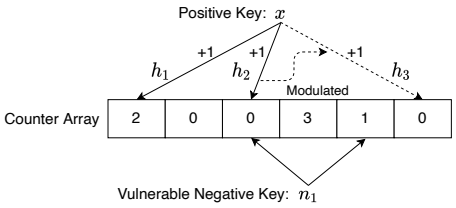


Figure 1: Hash Modulating

For instance, given a hotspot URL A (*e.g.*, outlook.com) and a rarely-visited URL B (*e.g.*, mail.21cn.com), it is better to distinguish URL A and URL B in building the filter, because treating A as a malicious URL brings significantly more safety validation than B. Another typical example is the Bitcoin Core whitelist [1], which is based on Bloom filter and designed to relieve the communication overhead by specifying which services should be provided to different IPs. Here, misidentifying service permission to malicious IPs [21] may raise the risks of being attacked. Similar examples include the access control list in the intrusion detection system [3].

We refer to these negative keys with high misidentifying costs as vulnerable negative keys. Note that these vulnerable negative key set can usually be obtained upfront, *e.g.*, the top popular websites (or URLs) worldwide. As far as we know, there have been few works regarding handling vulnerable negative keys during filtering, especially in dynamic scenarios. Table 1 outlines the comparison among different related filters, which can generally be divided into two types: passive filters and active filters. Passive filters include Counting Bloom filter and its variants [5, 12, 22], which can only improve its FPR by passively increasing their space usage. Recently, there have been a few works actively utilizing the prior knowledge of negative keys and are referred to as active filters [6, 9, 10, 17, 25, 26]. However, these active filters that are based on machine learning models suffer from several problems, including relying on semantic knowledge of data, prolonged construction and query latency, and no dynamicity [9, 17, 25]. Other active filters [6, 10, 26] are only designed to work in static offline scenarios[1], where positive and negative keys need to be known in advance. Besides, these filters [10, 26] need to scan keys multiple times, which makes them cannot be applied in dynamic scenarios.

In this paper, to handle the vulnerable negative keys, we propose a filter named SeeSaw Counting Filter (SSCF for short). SSCF has two components, including an underlying seesaw counter array (SCA) and a lightweight hash table named HashModulator. Then at insertion time, SSCF applies k default (or initial) hash functions and increases the k mapped seesaw counters by one. Different from the Counting Bloom filter, the applied hash functions can be modulated (or varied) by SSCF during insertion time. As a classical technique in signal processing to safely mix signals to be separated later, the modulating is borrowed here by varying the properties (*i.e.*, hash functions) of positive keys to guard vulnerable negative keys against being inseparable from positive keys. An example of hash modulating is shown in Figure 1, in which the hash function h_2 is deprecated and modulated to h_3 to guard vulnerable negative key n_1 (*i.e.*, avoiding the third left counter being increased to > 0). The modulated hash function is then stored into HashModulator, and will be retrieved at query time. Moreover, we develop the corresponding modulated query and deletion procedures while ensuring one-side query error pattern as Counting Bloom filter does, *i.e.*, only false positives and no false negatives. Meanwhile, to handle the scenarios where vulnerable negative keys cannot be obtained in advance, we propose adaptive SSCF (*i.e.*, ada-SSCF), which takes vulnerable negative keys as input dynamically and obtain continuously improved performance with frequent key insertions/deletions.

Challenges. In this paper, we are mainly faced with three challenges. The first challenge is how to ensure one-side query error when the applied hash functions are modulated. To address this challenge, we design a two-round query procedure, in which a negative key is said to be not in the filter if and only if it is both rejected with initial hash functions in the first round and modulated hash functions in second round. The second challenge is how to mitigate the computation overhead of hash modulating since hash modulating will incur more hash function computation and memory accesses, which is a big concern for dynamic scenarios. To address the challenge, we propose a lightweight modulating scheme named one-modulating, which modulates at most one hash function but still achieves significant performance gain with low operation latency. The third challenge is, at deletion time, how to avoid the inconsistent deletion of modulated hash functions from HashModulator since the information about which keys use the stored modulated hash functions is not maintained. To address the challenge, a counter field named ModulatedCounter indicating the times of each modulated hash function being used is added and acts like a virtual lock preventing inconsistent deletion.

Contributions. The main contributions are as follows:

(1) **Problem formulation**: We propose the dynamic cost-efficient filtering problem, where the vulnerable negative keys can be obtained and positive keys are dynamically inserted/deleted.
(2) **Cost-efficient filtering framework**: We propose SSCF that allows the applied hash functions to be modulated to guard vulnerable negative keys, which can also be extended to work with dynamically obtained vulnerable negative keys.
(3) **Evaluation**: We evaluate SSCF on representative datasets and show that, with the same memory space, SSCF achieves 3× or even higher accuracy (*i.e.*, cost-weighted FPR) with low operation latency comparable to Counting Bloom filter.

[1]Stacked filter [10] only allows minor (or slow) insertion/deletion after building the filter with the prior knowledge of all positive/negative keys.

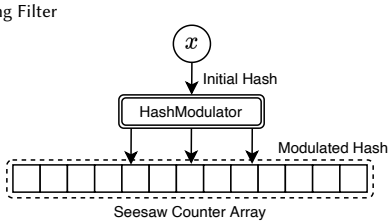


Figure 2: Architecture of Seesaw Counting Filter

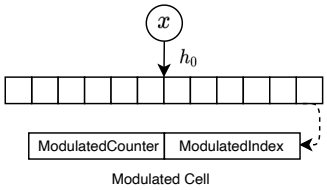


Figure 3: Structure of HashModulator

2 BACKGROUND

Counting Bloom filter. Designed for approximate membership testing, Bloom filter [4] aims to represent a set of keys, *e.g.*, $S = \{s_1, \cdots, s_n\}$, by encoding them into a bit array with k independent hash functions $\{h_1, \cdots, h_k\}$. To insert a key x, the array bits $h_i(x)$ are set to 1 for $1 \le i \le k$; to check whether x is in S, all mapped $h_i(x)$ bits are tested. If all $h_i(x)$ bits are 1, x is said to be in S with a small false positive rate and zero false negative rate. To support dynamic insertion and deletion, Counting Bloom filter (CBF) [12] is proposed by replacing the bit array with a (counter) cell array. Meanwhile, the insertion operation increases the mapped counter by one while the deletion operation decreases the mapped counter by one. Note that the underlying counter array may cause counter overflow, which, however, can be sufficiently addressed by allocating 4 bits for each counter [12]. Besides, the optimal hash function number $k = \lfloor B \ln 2 \rfloor$, where B is the number of bits (or counters) per key [4, 12]. All notations used are listed in Table 2.

3 SEESAW COUNTING FILTER

3.1 Problem Formulation

The problem studied is formalized as **Dynamic Cost-efficient Filtering Problem (DCFP)**, *i.e., given vulnerable negative keys $\mathcal{N}_n$ with their respective misidentifying costs, when dynamically inserting or deleting keys, how to minimize the overall cost incurred by misidentification of negative keys.*

To handle DCFP, we propose the seesaw counting filter (with HashModulator and seesaw counter array) as shown in Figure 2.

3.2 High-level Idea

We aim to encode vulnerable negative keys into the underlying seesaw counter array, each of which has two fields for recording negative keys and positive keys, as shown in Figure 4(a). At insertion or query time, similar to Counting Bloom filter, k hash functions are applied and map to the underling k seesaw counters. However, different from the standard Counting Bloom filter, an applied hash function may be modulated if it maps the inserted key to a seesaw counter that is already occupied by negative keys, which is shown in Figure 4(b). Specifically, if a seesaw counter is preoccupied by a negative key, the seesaw counter is said to lean to the negative cell and will be sealed to prevent positive keys from being inserted into the counter. If a hash function maps a positive key to a negative seesaw counter, the hash modulating is then activated to find a modulated hash function maps to another seesaw counter that is either empty or positive, which is shown in Figure 4(c). Note that the modulating may fail, and the initial hash functions will be applied if there is no such qualified modulated hash function that maps the inserted key to empty or positive seesaw counters. In that case, we will get a mixed seesaw counter with its negative and positive cells being both occupied. However, if the modulating succeeds, the modulated hash function is then stored into HashModulator and will be retrieved at query time. The key insight behind the modulating is to spare space for vulnerable negative keys from unimportant negative keys.

Besides, the modulating may incur extra overhead from hash function computation and memory accesses, which is a big concern in dynamic scenarios. To reduce such overhead, we propose a lightweight modulating policy named one-modulating, *i.e.*, only the hash function with the smallest index will be modulated for each key However, even with the one-modulating policy, a significant performance gain can be observed since a negative key is misidentified if all its k mapped counters are occupied by positive keys, whose probability can already be greatly reduced by one-modulating.

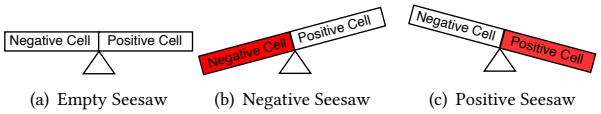
Table 2: Notations

Symbol	Description
N_p	Positive key capacity of SSCF
$\mathcal{N}_u$	The universe negative keys set
$\mathcal{N}_n$	The vulnerable negative keys set
N_n	Number of vulnerable negative keys, *i.e.* $N_n = \lvert\mathcal{N}_n\rvert$
B	Number of seesaw counters per positive key in SCA
m	Number of seesaw counters in SCA, $m = B \cdot N_p$
θ_1/θ_2	Size of negative/positive cell in bits per seesaw counter
$H_{\mathcal{A}}$	Initial hash function set of SSCF
$H_{\mathcal{B}}$	Backup modulated hash function set SSCF
k	Number of initial hash functions ($k = \lvert H_{\mathcal{A}}\rvert$)
$\widehat{k}$	Number of backup modulated hash functions ($\widehat{k} = \lvert H_{\mathcal{B}}\rvert$)
n	Number of cells in HashModulator
η_1	Size of *ModulatedCounter* field in bits in HashModulator
η_2	Size of *ModulatedIndex* field in bits in HashModulator
$C(x)$	The cost of filtering error for key x
α	The ratio of space allocated to HashModulator

3.3 Structure of SSCF

Seesaw Counting Array (SCA). As is shown in Figure 2, SCA is composed of an array of m seesaw counters, each of which has two fields: negative (counter) cell and positive (counter) cell as shown in Figure 4. The negative cell records whether this cell is mapped by vulnerable negative keys, which makes 1-bit space size is already enough. Specifically, when a negative key is mapped to a seesaw counter, its negative cell set to 1. As for the positive cell field, it is used to record the number of keys inserted into the cell. Similar to Counting Bloom filter, SSCF also has k accompanying hash functions, which is used to map inserted keys to the underlying counter array during insertion or query.

HashModulator. As shown in Figure 3, HashModulator consists of a cell array, each of which have two fields named *ModulatedCounter* and *ModulatedIndex*. The ModulatedIndex field records the index of the stored modulated hash function, whose times of



(a) Empty Seesaw (b) Negative Seesaw (c) Positive Seesaw

Figure 4: Seesaw Counter

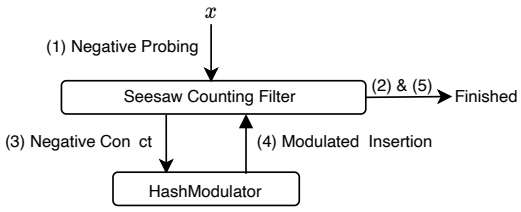


Figure 5: Modulated Insertion Procedure

being used is recorded in the ModulatedCounter field. For a given key x, it is mapped to HashModulator with a public hash function h_0.

3.4 Negative Key Encoding

To construct SSCF, the vulnerable negative keys need to be encoded firstly. The encoding procedure is similar to the insertion process of the standard Counting Bloom filter. Specifically, as shown by Steps 1 – 4 of Algorithm 1, for a given negative key x, the negative cells of the k seesaw counters in SCA mapped by x are set to 1. Here, the negative cell field is 1-bit by default in this paper.

Algorithm 1: Negative Key Encoding

Data: Negative key set $\mathcal{N}_n$, SSCF (SCA $\mathcal{S}$, HashModulator $\mathcal{M}$), initial hash functions $H_{\mathcal{A}} = \{h_1, \cdots, h_k\}$

1 **for** *negative key* $x \in \mathcal{N}_n$ **do**
2 **for** i_{th} *hash function* $h_i \in H_0$ **do**
3 $idx = h_i(x)$;
4 Set the negative cell of $\mathcal{S}[idx]$ to 1;

3.5 Modulated Insertion Procedure

After negative key encoding, as is shown in Figure 5, the main insertion procedure is innovated in hash modulating, including five steps: (1) negative probing; (2) if no negative seesaw counter probed, the insertion finishes; (3) negative conflict detected and hash modulating; (4) modulated insertion; (5) the insertion finishes.

Running Example. As is shown in Figure 6, when inserting positive key x, it is mapped by two hash functions h_1, h_2 to seesaw counters $S[1]$ and $S[3]$. However, the negative cell of $S[3]$ is probed to be non-empty, which then triggers the hash modulating to find a new hash function to redirect x to another seesaw counter with an empty negative cell. Suppose the qualified new hash function is h_4, as shown in the lower half of Figure 6, we need to record h_4 into the empty cell $M[4]$ (mapped by h_0) in HashModulator by increasing the ModulatorCounter by one and setting ModulatedIndex to 4. However, if the mapped cell is not empty, a trial of reusing the stored hash function for hash modulating is conducted instead of finding a new hash function and is detailed in the following hash modulating step. The three key steps, including negative probing, hash modulating, and modulated insertion, are explained as follows.

Negative Probing. The first step is negative probing, *i.e.*, testing whether there are any negative seesaw counters mapped by the initial hash functions. To be specific, given a key x to be inserted, we use the initial k hash functions $H_{\mathcal{A}}$ to probe its mapped seesaw counters to check whether there exist any seesaw counters with non-empty negative cells. If not, the positive cells of k mapped seesaw counters are increased by 1, and the modulated procedure ends. Otherwise, the hash modulating is triggered for the hash function that maps the x to the seesaw counter with a non-empty negative cell. There may be multiple hash functions probed to be negative conflict, but only the one with the smallest index will be marked as the candidate to be modulated. The probing procedure is shown in Steps 1 – 6 of Algorithm 2, in which the index of the hash function to be modulated is denoted as idx_{old} (Step 5).

Hash Modulating. With the obtained hash function to be modulated, the hash modulating is activated to find a qualified modulated hash function. However, we need to check whether the mapped cell in HashModulator is empty (Steps 7 – 8 of Algorithm 2).

Firstly, if the mapped cell in HashModulator is empty (*i.e.*, ModulatedCounter is 0), we then proceed to find a qualified modulated hash function from hash function candidate set $H_{\mathcal{B}}$ (Steps 9 – 14 of Algorithm 2). A qualified modulated hash function is found if it maps the inserted key x to an empty or positive seesaw counter (Steps 12 – 14, Algorithm 2). If a qualified hash function is found,

Algorithm 2: Modulated Insertion

Data: Key x, SSCF (SCA $\mathcal{S}$, HashModulator $\mathcal{M}$), initial hash functions $H_{\mathcal{A}}$, modulated hash functions $H_{\mathcal{B}}$

1 Set the hash function index to be modulated $idx_{old} = -1$ and the modulated hash function index $idx_{new} = -1$;
2 **for** *the* i_{th} *hash function* $h_i \in H_{\mathcal{A}}$ **do**
3 $j = h_i(x)$;
4 **if** $idx_{old} < 0$ *and negative cell of* $\mathcal{S}[j]$ *larger than* 0 **then**
5 $idx_{old} = i$;
  else
6 Increase the positive cell of counter $\mathcal{S}[j]$ by 1;
  if $idx_{old} > 0$ **then**
7 $j_{mod} = h_0(x)$;
8 **if** $\mathcal{M}[j_{mod}]$.*ModulatedCounter == 0* **then**
9 idx_{new} = -1 ;
10 **for** *the* i_{th} *hash function* $h_i \in H_{\mathcal{B}}$ **do**
11 $j_{new} = h_i(x)$;
12 **if** $\mathcal{S}[j_{new}].nc == 0$ **then**
13 $idx_{new} = i$;
14 **break**;
15 **if** $idx_{new} > 0$ **then**
16 $M[j_{mod}].ModulatedIndex = idx_{new}$;
17 $j = H_{\mathcal{B}}[idx_{new}](x)$;
18 Increase the positive cell of counter $\mathcal{S}[j]$ by 1;
  else
19 $j = H_{\mathcal{A}}[idx_{old}](x)$;
20 Increase the positive cell of counter $\mathcal{S}[j]$ by 1;
  else
21 $idx_{new} = \mathcal{M}[j_{mod}].ModulatedIndex$;
22 $j = H_{\mathcal{B}}[idx_{new}](x)$;
  if $\mathcal{S}[j].nc == 0$ **then**
23 Increase the positive cell of counter $\mathcal{S}[j]$ by 1;
  else
24 $j = H_{\mathcal{A}}[idx_{old}](x)$;
25 Increase the positive cell of counter $\mathcal{S}[j]$ by 1;
26 Increase $\mathcal{M}[j_{mod}]$.ModulatedCounter by one;

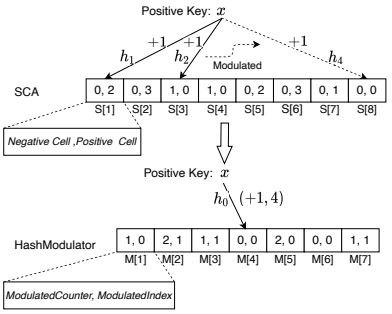


Figure 6: Running Example: Modulated Insertion

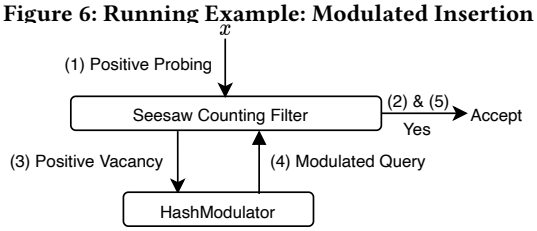


Figure 7: Modulated Query Procedure

its index denoted as idx_{new} will be stored into HashModulator (Steps 16, Algorithm 2). Then the modulated hash function comes into use, which will increase the positive (counter) cell of its mapped seesaw counter by one as shown by Steps 17 – 18 of Algorithm 2. However, if no qualified modulated hash function is found, the hash modulating fails, followed by which the original initial hash function is used, which is shown in Steps 19 – 20 of Algorithm 2.

Secondly, if the mapped cell in HashModulator is occupied, we will first check whether the stored hash function stored in the occupied cell can be reused. If the hash function can be reused, we increase the positive (counter) cell of seesaw counter mapped by the reused hash function by one (Steps 21 – 23 of Algorithm 2). Otherwise, we turn to using the original initial hash function as shown by Steps 24 – 25 of Algorithm 2.

Modulated Insertion. Finally, we need to increase the ModulatedCounter field mapped by the key x by one, named Modulated Insertion as shown in Step 26. After the modulated insertion step, the whole insertion process ends as shown in Figure 5.

3.6 Modulated Query Procedure

The query procedure is shown in Figure 7 and Algorithm 3, which can be divided into two rounds as shown in Figure 7. To ensure zero FNR, we restrict that a key is rejected if it is rejected in both rounds. In the first round, the first Step (Positive Probing, Figure 7) is positive-probing, which probes whether all mapped seesaw counters have non-empty positive cells. If yes, the queried key is accepted by SSCF as shown by Step 2 in Figure 7. Otherwise, the second round is activated if only one seesaw counter is probed to have an empty positive cell (Positive Vacancy, Figure 7). In the second round, we retrieve a modulated hash function and accept x if the positive cell of the new mapped seesaw counters is non-empty (Steps 4 – 5, Figure 7). In other cases, the queried key is rejected.

Running Example. In Figure 8, when querying key x, it is mapped by two hash functions h_1, h_2 to seesaw counters $S[1]$ and $S[3]$. However, the positive cell of $S[3]$ is probed to be empty, which makes key x is rejected in the first round and triggers the procedure of retrieving the modulated hash function stored in HashModulator, *i.e.*, h_4 in $M[4]$. With the retrieved h_4, x is redirected and mapped to $S[8]$, which then makes x accepted in the second round.

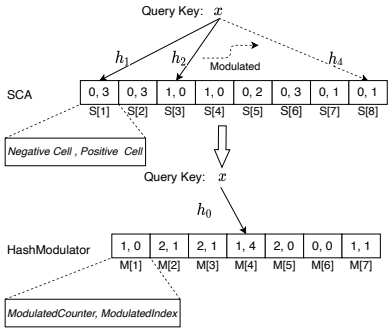


Figure 8: Running Example: Modulated Query

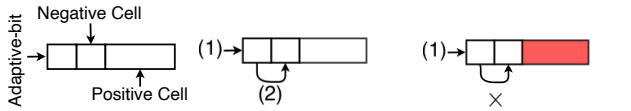
The detailed procedure of modulated query is presented in Algorithm 3, which includes the first query round (Steps 2 – 6) and the second round (Steps 7 – 14). In the first round, SSCF firstly checks the k initial hash functions by probing their mapped seesaw counters. If all the probed positive cells are non-empty, the queried key x is accepted directly (Step 15, Algorithm 3). If more than one positive cell is empty (Steps 4 – 5, Algorithm 3), the queried key x is rejected in the first round due to our one-modulating policy. Otherwise, if only one positive cell is probed to be empty, the second round is activated to retrieve the modulated hash function.

However, the retrieval may fail if the mapped cell of the inserted key in HashModulator is empty (Steps 9 – 10, Algorithm 3); otherwise, a modulated hash function is retrieved with index denoted as idx_{new} (Step 11, Algorithm 3). The retrieved modulated hash function is used by probing its mapped seesaw counters (Steps 12 – 14, Algorithm 3). Considering that a modulated hash function will be adopted if it maps to a seesaw counter with an empty negative cell during insertion, the queried key x is rejected when the negative

Algorithm 3: Modulated Query (x)

Data: Key x, SSCF (SCA $\mathcal{S}$, HashModulator $\mathcal{M}$), initial hash functions $H_{\mathcal{A}}$, modulated hash functions $H_{\mathcal{B}}$
Result: Whether x is in SSCF

1 Set the hash function index to be modulated $idx_{old} = -1$ and the modulated hash function index $idx_{new} = -1$;
2 **for** *the* i_{th} *hash function* $h_i \in H_{\mathcal{A}}$ **do**
3 $j = h_i(x)$;
4 **if** *positive cell of* $\mathcal{S}[j]$ *is empty* **then**
5 **if** $idx_{old} > 0$ **then return** *false*;
6 **else** $idx_{old} = i$;
7 **if** $idx_{old} > 0$ **then**
8 $j_{mod} = h_0(x)$;
9 **if** $\mathcal{M}[j_{mod}].ModulatedCounter == 0$ **then**
10 **return** *false*;
11 **else** $idx_{new} = \mathcal{M}[j_{mod}].ModulatedIndex$; ;
12 $j_{new} = H_{\mathcal{B}}[idx_{new}](x)$; ;
13 **if** $\mathcal{S}[j_{new}]$ *has non-empty negative cell or empty positive cell* **then**
14 **return** *false*;
15 **return** *true* ;



(a) Structure (b) Adaptive Encoding (c) Blocked Adaptive Encoding

Figure 9: Adaptive Seesaw Counter

cell is non-empty (Step 14, Algorithm 3). Similarly, the queried key x is rejected if the positive cell of the mapped seesaw counter is empty (Step 14, Algorithm 3). In other cases, the queried key is accepted directly (Step 15).

3.7 Modulated Deletion Procedure

Another essential operation is deletion, which involves decreasing the mapped counters in SCA and removing the modulated hash function from HashModulator. We omit it here for space saving.

3.8 Extension: Adaptive SSCF

When the vulnerable negative keys may not be obtained in advance, we propose adaptive SSCF (ada-SSCF for short), which can take (or absorb) the vulnerable keys as input dynamically. Note that these absorbed vulnerable negative keys may not bring immediate decreased cost-weighted FPR but will contribute to continuously improved cost-weighted FPR when keys are deleted and inserted. In other words, the performance of ada-SSCF improves over time. As shown in Figure 9(a), ada-SSCF also has HashModulator but is equipped with a new adaptive seesaw counter array, each of which has an extra cell named Adaptive-bit recording the vulnerable negative keys at runtime.

Now we introduce how to encode vulnerable negative keys at runtime. Given vulnerable negative key x, x is mapped with the k initial hash functions to the underlying adaptive seesaw counters. For each mapped seesaw counter, we set its adaptive-bit field to 1. However, with only Adaptive-bit cell being set, the hash modulating will not be triggered even if this cell is mapped by positive keys since the hash modulating only relies on the negative cell. To address this problem, we need to transport this encoded Adaptive-bit to the negative cell field of the same cell. According to whether the positive cell is empty in the same counter, the transportation can be divided into two cases. Firstly, as shown in Figure 9(b), if the positive cell is empty, the corresponding negative cell in the same cell can be set to 1 safely since this mapped positive cell is not occupied by any positive keys. Secondly, if the positive cell is not empty, the transportation is then blocked since directly setting the negative cell may make it hard to distinguish whether the applied hash functions of positive keys mapped to this cell are modulated or not, which may further lead to the inconsistent deletion or even FNR. Therefore, as shown in Figure 9(c), direct transportation is not feasible and will be blocked when the positive cell is non-empty. However, in the dynamic scenarios, deletions or insertions are frequent, and thus makes the blocked transportation can be resumed and carried out once the positive cell is decreased to 0.

4 EXPERIMENT EVALUATION

4.1 Experiment Setup

4.1.1 ***Comparison Filter Implementation.*** The first filter is Counting Bloom filter (CBF) [12]. By inheriting the optimal parameter setting of CBF, we set the number of hash functions $k = \lfloor ln2 \cdot B \rfloor$ to minimize the FPR for CBF, where B is the number of counters per key. Besides, each counter cell is set to 4-bit according to [12], which is also inherited by SSCF. The second comparison filter is Weighted Bloom filter (WBF) [6]. However, considering that deletion is not supported by WBF, we adapt WBF to our problem by replacing its bit array with counter array and name the adapted version as Weighted Counting Bloom filter (WCBF for short). The third Filter is Stacked Filter (SF) [10]. However, SF cannot be borrowed directly here since the construction of SF relies on the prior knowledge of positive and negative keys, which cannot be obtained in advance in dynamic scenarios. To adapt SF to our problem, we build a three-layer SF with the second layer (*i.e.*, negative filter layer) storing a default portion about 5% (same as SSCF) of negative keys with the highest cost. Besides, inspired by [10], the space size allocated for each filter layer to make each Filter have roughly the same FPR. Moreover, to make SF support deletion, we replace the Bloom filter in SF framework with Counting Bloom filter. To achieve a head-to-head comparison, we require the space consumption of all filters evaluated to be the same. The memory space consumption is indicated by the metric named bits-per-key (*i.e.*, bits per positive key), which equals the total memory space size in bits divided by the positive key number.

4.1.2 ***Evaluation Metrics.*** We use the following metrics: (1) cost-weighted FPR; (2) insertion latency; (3) query latency; and (4) deletion latency. The first metric, *i.e.*, cost-weighted FPR, is a variant of standard FPR that takes key costs as weighted factors. Specifically, the cost-weighted FPR is defined as $\frac{\sum_{x \in \mathcal{N}_u} F(x) \cdot C(x)}{\sum_{x \in \mathcal{N}_u} C(x)}$, where $\mathcal{N}_u$ is the negative key set, $F(x) \in [0, 1]$ is the queried result of key x in filter F, and $C(x)$ denotes the cost of key x.

4.1.3 ***Data Sets.*** To validate the effectiveness of SSCF, the following two data sets are used:

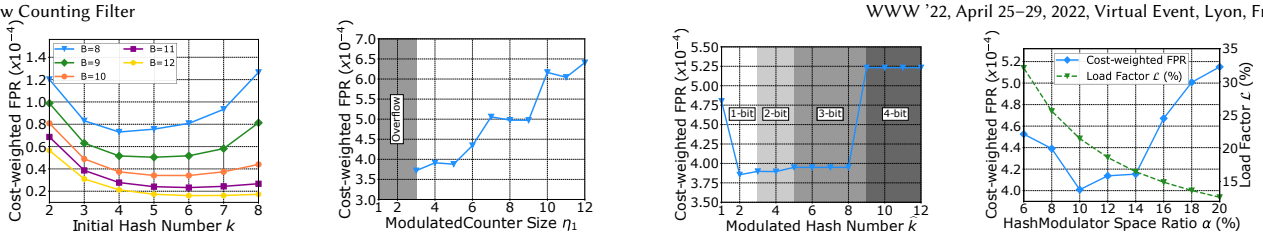
(1) Shalla's Blacklists: Shalla's Blacklists [16] (abbreviated as Shalla) is a URL dataset, which contains malicious URLs to be blocked and used here to simulate building dynamic white-list URLs that can be accessed safely.
(2) YCSB: YCSB is a benchmark [8] designed by Yahoo for performance evaluating of key-value stores. We use YCSB here to simulate network cache applications, where data are dynamically added or deleted. In such applications, filtering the frequently missed access requests is important.

4.1.4 ***Cost Distribution.*** Considering that the keys from Shalla and YCSB have no default cost, a skewed cost distribution (*i.e.*, Zipf distribution [23]) is generated. To reveal the correlation between the cost skewness and filtering performance, various skewness parameters (from 1.0 to 2.5) are generated.

4.2 Parameter Evaluation

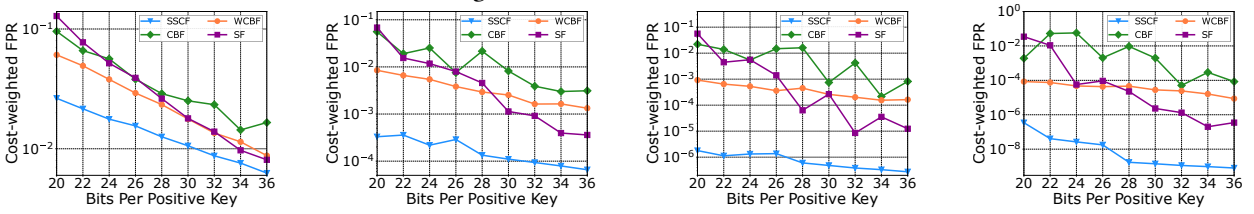
Given the total memory space budget, the performance of SSCF depends on the following parameters: (1) the number of initial hash functions k; (2) the size of ModulatedCounter field η_1; (3) the number of backup modulated hash functions $\widehat{k}$ and ModulatedIndex field size η_2; and (4) HashModulator space ratio α. We evaluate these parameters on Shalla with 1.5 Zipf skewness to study their effects.

(1) *Number of Initial Hash Functions k.* In Figure 10(a), we vary the initial hash function number against different counters per key. Overall, the optimal k is 4 for $B = 8$ and increases to 8 for



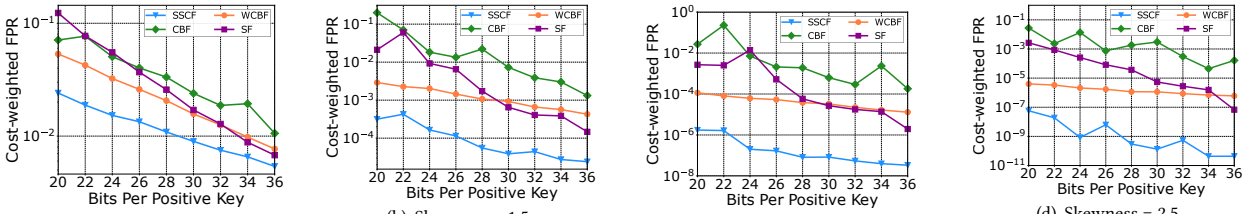
(a) Cost-weighted FPR vs. k (b) Cost-weighted FPR vs. η_1 (c) Cost-weighted FPR vs. $\widehat{k}$ & η_2 (d) Cost-weighted FPR vs. α

Figure 10: Parameter Evaluation



(a) Skewness = 1 (b) Skewness = 1.5 (c) Skewness = 2 (d) Skewness = 2.5

Figure 11: Cost-weighted FPR on Shalla vs. Skewness under the Same Memory Space (Varying from 3.5MB to 6.4MB)



(a) Skewness = 1 (b) Skewness = 1.5 (c) Skewness = 2 (d) Skewness = 2.5

Figure 12: Cost-weighted FPR on YCSB vs. Skewness under the Same Memory Space (Varying from 29.8MB to 53.6MB)

$B = 12$, which roughly agrees with the optimal hash function number setting $k = \lfloor B \cdot \ln(2) \rfloor$ from CBF. Moreover, given the number of seesaw counters per key B, the cost-weighted FPR drops at first but then increases as B increases.

(2) *ModulatedCounter Size η_1.* As is shown in Figure 10(b), the ModulatedCounter size η_1 is varied from 1 to 12. The results show that when the η_1 is small (≤ 2), the ModulatedCounter field size is too small to record the times of stored modulated hash function being used, and thus overflows; when η_1 increases from 3 to 12, the cost-weighted increases gradually since the ModulatedCounter field occupies more space but brings no benefits except the increased capacity of each modulated hash function being used.

(3) *Number of Modulated Hash Functions $\widehat{k}$ and Modulated Field Size η_2.* The number of modulated hash functions $\widehat{k}$ is constrained by the size η_2 of ModulatedIndex field, *i.e.* $k \leq 2^{\eta_2}$. In Figure 10(c), with $\widehat{k}$ increased from 1 to 12, the optimal $\widehat{k}$ is 2 and can be covered by 1-bit ModulatedIndex. For $\widehat{k} < 2$, the number of modulated hash functions is limited, which brings down the successful probability of hash modulating. As for $\widehat{k} > 2$, the ModulatedIndex field consumes more space to cover the modulated hash index (up to $\widehat{k}$), which leads to cost-weighted FPR deterioration.

(4) *HashModulator Space Ratio α.* As is shown in Figure 10(d), we vary α from 6% to 20%. The optimal cost-weighted FPR is achieved when $\alpha = 10\%$. For $\alpha < 10\%$, the load factor $\mathcal{L}$ of HashModulator is high, which leads to more hash collisions in HashModulator, and thus more misidentification in the second query round of modulated query procedure. For $\alpha > 10\%$, the space allocated for SCA becomes too small, which leads to more misidentification in the first query round, and thus higher cost-weighted FPR.

4.3 Overall Filtering Performance

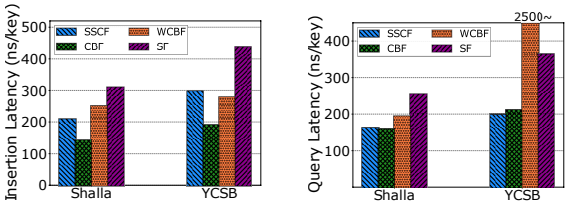
In this subsection, we evaluate all filters by varying the space size, and Zipf cost distribution skewness.

SSCF always has the smallest cost-weighted FPR under all space settings and outperforms all the comparison filters at least by 1.55× *and up to two orders of magnitude on skewed data.* For Shalla with skewness 1.0, as shown in Figure 11(a), the cost-weighted FPR of SSCF decreases from 2.99% to 0.57%. Among other filters, the SF shows the best performance with cost-weighted FPR decreasing from 5.62% to 0.63%. Even compared with SF, SSCF shows over 1.55× performance improvement and obtains larger performance gain under larger skewness, which is as shown in Figure 11(b), Figure 11(c) and Figure 11(d). Particularly, SSCF outperforms SF over two orders of magnitude with 2.5 Zipf skewness, which is as shown in Figure 11(d).

As is shown in Figure 12(a), for YCSB with skewness 1.0, the cost-weighted FPR of SSCF decreases from 2.35% to 0.51%. Similar to that on Shalla, SF also shows the best performance with cost-weighted FPR decreasing from 4.80% to 0.54% on YCSB with Zipf skewness 1.0. Meanwhile, SSCF outperforms all the other comparison filters by at least 1.45×. With the increased cost distribution skewness, the performance gap enlarges as shown in Figure 12(b), Figure 12(c) and Figure 12(d). As is shown in Figure 12(d), on YCSB with 2.5 Zipf skewness, SSCF also outperforms all other comparison filters by over two orders of magnitude.

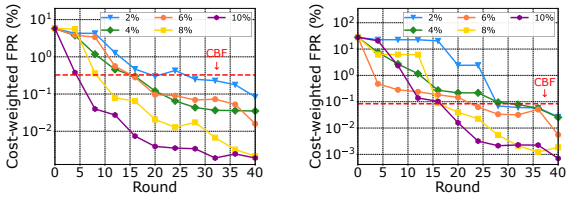
4.4 Operation Latency

4.4.1 ***Insertion Latency.*** *The insertion latency of SSCF is about* 1.47× *the latency of CBF.* As shown in Figure 13(a), on Shalla, the insertion latency per key is 208ns for SSCF, 141ns for CBF, 250ns



(a) Insertion latency (b) Query latency

Figure 13: Operation Latency



(a) Ada-SSCF on Shalla (b) Ada-SSCF on YCSB

Figure 14: Ada-SSCF vs. Cost-weighted FPR

for WCBF, and 310ns for SF. As for YCSB, the insertion latency is 296ns for SSCF, 190ns for CBF, 277ns for WCBF, and 438ns for SF.

4.4.2 ***Query Latency****. The query latency of SSCF similar to the latency of CBF.* The query latency per key on shalla, as shown in Figure 13(b), is 163ns for SSCF; for CBF, it is 158ns; for WCBF and SF, it is 195ns and 253ns, respectively. On YCSB, the insertion latency for SSCF, CBF, WCBF and SF is 199ns, 211ns, 2548ns and 364ns, respectively. WCBF suffers from high query latency as it needs to check the costs of queried keys, and particularly, the computation overhead of the checking process is non-negligible when the number of keys stored is large.

4.5 Ada-SSCF is Robust in Dynamic Scenarios.

In this subsection, we evaluate ada-SSCF in the dynamic scenarios, where the vulnerable negative keys cannot be obtained in advance. To be specific, at first, we only insert positive keys into ada-SSCF and feed 5% the vulnerable negative keys with the highest costs to ada-SSCF dynamically. Then, we randomly delete and insert keys from ada-SSCF round by round with a fixed ratio, which is as shown in Figure 14. For example, a round with 2% ratio means that in one single round, we randomly delete 2% keys from SSCF and then randomly insert 2% new keys into SSCF. It can be observed that with more and more deletions and insertions, the cost-weighted FPR of ada-SSCF improves gradually and finally surpasses the CBF. Particularly, as shown in Figure 14(a), the cost-weighted FPR of ada-SSCF decreases from 5.7% to 0.08% with 2% ratio in 40 rounds and can even reach up to 0.0019% with 10% ratio, which is about two orders of magnitude improvement compared with CBF. Similarly, as shown in Figure 14(b), the cost-weighted FPR of ada-SSCF decreases from 27% to 0.02% with 2% ratio in 40 rounds and even to 0.00038% with 10% ratio. Particularly, in the scenarios where deletions are more than insertions, the performance improves much faster.

5 RELATED WORK

Filters that are cost-aware. The standard Bloom filter and its variants do not take into account the costs of keys [5, 12], which makes all negative keys are treated identically. To handle keys with different costs, Bruck *et al.* proposed to vary the hash functions of each key according to its cost and formalized a new filter named Weighted Bloom filter (WBF) [6]. Nonetheless, the problem is that WBF needs to calculate the number of applied hash functions for each key at query time, which thus requires the storage of key cost, and then leads to high space usage and query latency at query time. Considering that the varied hash function approach proposed by WBF remains heuristic, Zhong *et al.* adopted a similar idea but posed it as a constrained nonlinear integer programming problem [28], which can only work offline. The recent proposed stacked filter framework [10] proposes to learn from the workload in a structured way, *i.e.*, stacking the filters one by one to filter keys progressively. However, one important problem for the stacked filter is that it needs prior knowledge of both positive and negative keys, making it cannot be applied in dynamic scenarios. Other filters like Rosetta [19] and SuRF [27] are designed for range query problem.

Filters that are learning-based. With an elaborately trained learned model, existing learning-based works could achieve beyond the theoretical limit performance in terms of FPR and space efficiency [17, 20]. Kraska *et al.* proposed a learned Bloom filter [17] to obtain optimized space efficiency by incorporating machine techniques that can capture data distribution information within a small learned model. Adaptive Learned Bloom filter was proposed to use machine learning technique to measure the probability of whether a key in the set and adaptively decides the number of hash functions applied [9]. However, despite the remarkable space efficiency, existing learning-based filters all suffer from prolonged training and query latency. Besides, they cannot be adapted to dynamic workloads since the learned models need to be repeatedly retrained on new data, which is unacceptable in dynamic scenarios. Based on customizing the hash function in an offline setting, HABF is designed for static set filtering, which does not support dynamic insertions or deletions [26]. Besides, HABF needs to scan keys repeatedly to obtain an optimized hashing schema, which incurs heavy memory overhead from storing all keys during construction.

6 CONCLUSION

In this paper, we have studied the proposed dynamic cost-efficient filtering problem. Targeting at such problem, we propose a new filter named (ada-)SSCF, which is innovated in a lightweight negative key encoding mechanism and dynamic hash method named hash modulating. With hash modulating, SSCF provides the adaptivity of choosing applied hash functions dynamically to prevent vulnerable negative keys from being misidentified. To validate the performance, SSCF is extensively evaluated on several representative data sets and outperforms the standard Counting Bloom filter, stacked filter and other variants on the whole regarding accuracy, construction time/memory, query latency and filter size.

ACKNOWLEDGMENTS

This work was supported in part by the Natural Science Foundation of Jiangsu Province under Grant No. BK20181251, in part by the Key Research and Development Project of Jiangsu Province under Grant No. BE2015154 and BE2016120, in part by the National Natural Science Foundation of China (No. 61872178, No. 61832005, No. 61672276, NO. 62072230, U1811461), in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization, Nanjing University, in part by the Jiangsu High-level

Innovation and Entrepreneurship (Shuangchuang) Program, in part by Alibaba Innovative Research Project, in part by Singapore MOE AcRF Tier 1 (RG18/21), and NTU Startup Grant. Haipeng Dai and Guihai Chen are the corresponding authors.

REFERENCES

[1] [n.d.]. https://bitcoinops.org/en/newsletters/2019/08/21/#bitcoin-core-16248.
[2] 2021. SSCF Source Code. https://anonymous.4open.science/r/SSCF-7505.
[3] Parvez Anandam. 2019. Network Access Control using Bloom filters. https://courses.cs.washington.edu/courses/csep521/07wi/prj/parvez.pdf.
[4] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM* 13, 7 (1970), 422–426.
[5] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An improved construction for counting bloom filters. In *European Symposium on Algorithms*. Springer, 684–695.
[6] Jehoshua Bruck, Jie Gao, and Anxiao Jiang. 2006. Weighted Bloom filter. In *International Symposium on Information Theory*. IEEE, 2304–2308.
[7] Denis Charles and Kumar Chellapilla. 2008. Bloomier filters: A second look. In *European Symposium on Algorithms*. Springer, 259–270.
[8] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of symposium on Cloud computing*. ACM, 143–154.
[9] Zhenwei Dai and Anshumali Shrivastava. 2019. Adaptive learned Bloom filter (Ada-BF): Efficient utilization of the classifier. *arXiv preprint* (2019).
[10] Kyle Deeds, Brian Hentschel, and Stratos Idreos. 2020. Stacked filters: learning to filter by structure. In *Proceedings of International Conference on Very Large Data Bases*, Vol. 14. VLDB Endowment, 600–612.
[11] Facebook. 2013. A facebook fork of leveldb which is optimized for flash and big memory machines. https://rocksdb.org/.
[12] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *Transactions on Networking* 8, 3 (2000), 281–293.
[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. ACM, 29–43.
[14] Bob Goodwin, Michael Hopcroft, Dan Luu, Alex Clemmer, Mihaela Curmei, Sameh Elnikety, and Yuxiong He. 2017. BitFunnel: Revisiting signatures for search. In *Proceedings of International Conference on Research and Development in Information Retrieval*. ACM, 605–614.
[15] Google. 2011. LevelDB. A fast and lightweight key/value database library. http://code.google.com/p/leveldb/.
[16] Shalla Secure Services KG. 2021. Shalla's Blacklists. http://www.shallalist.de/index.html.
[17] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the International Conference on Management of Data*. ACM, 489–504.
[18] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. 2006. Perfect hashing for network applications. In *International Symposium on Information Theory*. IEEE, 2774–2778.
[19] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. 2020. Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, Portland OR USA, 2071–2086. https://doi.org/10/gpcb5b
[20] Michael Mitzenmacher. 2018. A model for learned Bloom filters and optimizing by sandwiching. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 464–473.
[21] Palo Alto Networks. 2019. Palo Alto Networks Malicious IP Address Feeds. https://docs.paloaltonetworks.com/pan-os/8-1/pan-os-admin/policy/use-an-external-dynamic-list-in-policy/palo-alto-networks-malicious-ip-address-feeds.
[22] Salvatore Pontarelli, Pedro Reviriego, and Juan Antonio Maestro. 2016. Improving counting Bloom filter performance with fingerprints. *Inform. Process. Lett.* 116, 4 (2016), 304–309.
[23] David MW Powers. 1998. Applications and explanations of Zipf's law. In *New methods in language processing and computational natural language learning*. Association for Computational Linguistics.
[24] Do Le Quoc, Istemi Ekin Akkus, Pramod Bhatotia, Spyros Blanas, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. 2018. Approxjoin: Approximate distributed joins. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 426–438.
[25] Karan Singhal and Philip Weiss. 2020. DeepBloom. https://github.com/karan1149/DeepBloom/tree/master/data.
[26] Rongbiao Xie, Meng Li, Zheyu Miao, Rong Gu, Huang He, Haipeng Dai, and Guihai Chen. 2021. Hash Adaptive Bloom filter. In *Proceedings of International Conference on Data Engineering*. IEEE.
[27] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, Houston TX USA, 323–336. https://doi.org/10/gg224v
[28] Ming Zhong, Pin Lu, Kai Shen, and Joel Seiferas. 2008. Optimizing data popularity conscious Bloom filters. In *Proceedings of symposium on Principles of distributed computing*. ACM, 355–364.