Seesaw Counting Filter: A Dynamic Filtering Framework for Vulnerable Negative Keys

Meng Li, Deyi Chen, Haipeng Dai, Rongbiao Xie, Siqiang Luo, *Member, IEEE*, Rong Gu, *Member, IEEE*, Tong Yang, *Member, IEEE*, and Guihai Chen, *Fellow, IEEE*

***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. Such problem gets more severe when keys within one set are dynamically added or deleted. Recently, there are works focusing on handling such (vulnerable) negative keys by incorporating learning techniques. These learning-based filters fail to work as the learning techniques can hardly handle incremental insertions or deletions. 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 design 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 (SSCF-authors).**

***Index Terms*—Bloom filter, negaitve keys, query processing.**

I. 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. To be specific, 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 information retrieval [2], network applications [3], [4], database [5], [6], [7].

Dynamic 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 [15] in KV-stores like LevelDB [16]. However, there also exist important scenarios where data are inserted and deleted dynamically. To process dynamic set filtering, a Bloom filter variant named Counting Bloom filter (CBF) is proposed by replacing the underlying bit array with a counter array [3]. CBF has been widely applied in various applications, including web caching [3], transactional memory [17] and prefix matching [18].

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 [12], 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 [12], [13], [14]. 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 [13]. However, a significantly large cost will be brought when certain common URLs are misidentified. 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 [19], which is based on Bloom filter and designed to relieve communication overhead by

Manuscript received 2 May 2022; revised 8 December 2022; accepted 20 April 2023. Date of publication 8 May 2023; date of current version 8 November 2023. This work was supported in part by the Natural Science Foundation of Jiangsu Province under Grant BK20181251, in part by the Key Research and Development Project of Jiangsu Province under Grant BE2015154 and BE2016120, in part by the National Natural Science Foundation of China 61872178, 61832005, 61672276, 62072230, U1811461, 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. Recommended for acceptance by X. Xiao. *(Meng Li and Deyi Chen contributed equally to this work.) (Corresponding authors: Haipeng Dai; Guihai Chen.)*

Meng Li, Deyi Chen, Haipeng Dai, Rongbiao Xie, Rong Gu, and Guihai Chen are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210093, China (e-mail: menson@smail.nju.edu.cn; mf20330002@smail.nju.edu.cn; haipengdai@nju.edu.cn; rongbiaoxie@smail.nju.edu.cn; gurong@nju.edu.cn; gchen@nju.edu.cn).

Siqiang Luo is with the Nanyang Technological University, Singapore 639798 (e-mail: siqiang.luo@ntu.edu.sg).

Tong Yang is with the Peaking University, Beijing 100871, China (e-mail: yang.tong@pku.edu.cn).

Digital Object Identifier 10.1109/TKDE.2023.3273709

TABLE I
COMPARISON AMONG FILTERS

Filter Variants	Cost-efficiency	Dynamicity	One-pass Building	Prior Knowledge
Counting Bloom Filter [3], [8]	×	✓	✓	None
Learned Filter [9], [10], [11]	×	×	×	Positive & Negative keys
Weighted Bloom Filter [12]	✓	Partial	✓	Positive & Negative keys
Stacked Filter [13]	✓	Partial	×	Positive & Negative keys
HABF [14]	✓	×	×	Positive & Negative keys
SSCF/Ada-SSCF	✓	✓	✓	Vulnerable Negative Keys/None

specifying which services should be provided to different IPs. Here, misidentifying service permission to malicious IPs [20] may raise the risks of system being attacked.

We refer to these negative keys with high misidentifying cost 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 the dynamic scenarios. Table I outlines the comparison among different filters related, which can generally be divided into two types: passive filters and active filters. Passive filters include Counting Bloom filter and its variants [3], [8], [21], 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 [9], [10], [11], [12], [13], [14]. However, these active filters that are based on machine learning models suffer from several problems, including (1) relying on how well the used machine learning models can fit the underlying data distribution; (2) prolonged construction or query latency; (3) no incremental key insertion or deletion allowed [9], [10], [11]. Other active filters [12], [13], [14] are only designed to work in static offline scenarios[1], where positive and negative keys need to be known in advance. Besides, these filters [13,14] need to scan the full set of keys multiple times, which makes these filters cannot be applied in scenarios where incremental key insertions or deletions are required.

In this article, 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. Given vulnerable negative keys, SSCF encodes them into the seesaw counter array, which can also record positive keys at insertion time. 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 the 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 from being inseparable from positive keys. An example of hash modulating is shown in Fig. 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).

[1]Stacked filter [13] only allows minor insertion/deletion after building the filter with the prior knowledge of all positive/negative keys.

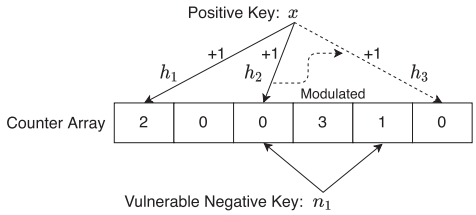


Fig. 1. Hash modulating.

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 under frequent key insertions/deletions. Compared with previous works, SSCF/ada-SSCF provide a lightweight and flexible way to be adaptive to dynamic workloads.

Challenges: In this article, 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 to mitigate the extra computation overhead brought by our proposed (hash modulating) operation, which involves extra hash function computation and memory accesses. 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 are the first to consider the dynamic cost-efficient filtering problem, where the vulnerable negative keys can be obtained and positive keys can be dynamically inserted/deleted.

TABLE II
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 of 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

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) *Theory:* We provide a theoretical analysis of SSCF regarding accuracy and parameter optimizations.
4) *Evaluation:* We evaluate SSCF on representative datasets and show that, with the same total memory space, SSCF achieves $3\times$ or even higher accuracy (i.e., cost-weighted FPR) with low operation latency comparable to Counting Bloom filter.

II. BACKGROUND

Counting Bloom Filter: Designed for approximate membership testing, Bloom filter [22] 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 (FPR) and zero false negative rate (FNR). To support deletion, Counting Bloom filter (CBF) [3] is proposed by replacing the bit array with a counter array. Meanwhile, the insertion operation increases the mapped counter by one while the deletion operation decreases the mapped counter by one. Frequently used notations used are listed in Table II.

Evaluation Metric. To measure the filtering error incurred by the misidentification of negative keys with different costs, inspired by [13], [14], we propose a new metric named cost-weighted FPR (CFPR for short), which 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.

III. SEESAW COUNTING FILTER

A. 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.*

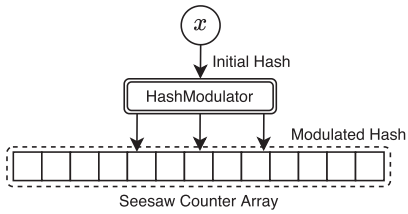


Fig. 2. Architecture of seesaw counting filter.

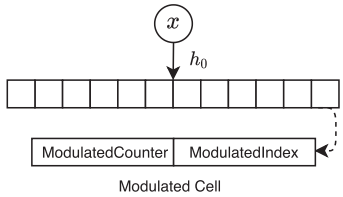


Fig. 3. Structure of hashModulator.

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

Fig. 4. Seesaw counter.

B. High-Level Idea

To handle the above DCFP, our proposed SSCF is shown in Fig. 2. 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 Fig. 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 Fig. 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 Fig. 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

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;

modulating is to spare space for vulnerable negative keys from unimportant negative keys.

Besides, the modulating may bring overhead due to hash 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 our one-modulating policy.

C. Structure of SSCF

Seesaw Counting Array (SCA): As is shown in Fig. 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 Fig. 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 is 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 Fig. 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 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.

D. Negative Key Encoding

To construct SSCF, the vulnerable negative keys need to be encoded first. 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 article.

E. Modulated Insertion Procedure

After negative key encoding, as is shown in Fig. 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.

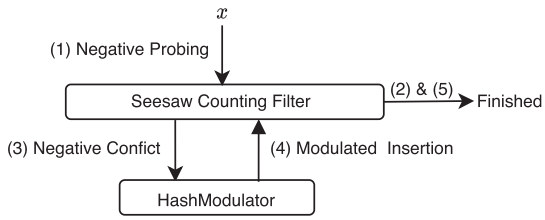


Fig. 5. Modulated insertion procedure.

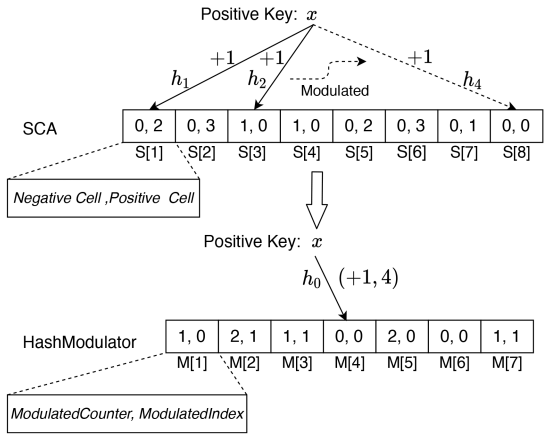


Fig. 6. Running example: Modulated insertion.

Running Example: As is shown in Fig. 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 Fig. 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 and is detailed in the following hash modulating step. The three key steps, including negative probing, hash modulating, and modulated insertion, are 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

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** *the negative cell of* $\mathcal{S}[j_{new}]$ *equals* 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 $M[j_{mod}].ModulatedIndex = 0$;
20 $j = H_{\mathcal{A}}[idx_{old}](x)$;
21 Increase the positive cell of counter $\mathcal{S}[j]$ by 1;
  else
22 $idx_{new} = \mathcal{M}[j_{mod}].ModulatedIndex$;
23 $j = H_{\mathcal{B}}[idx_{new}](x)$;
  if *the negative cell of* $\mathcal{S}[j]$ *equals* 0 **then**
24 Increase the positive cell of counter $\mathcal{S}[j]$ by 1;
  else
25 $j = H_{\mathcal{A}}[idx_{old}](x)$;
26 Increase the positive cell of counter $\mathcal{S}[j]$ by 1;
27 Increase $\mathcal{M}[j_{mod}]$.ModulatedCounter by 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).

First, 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, 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 17–19 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 20 21 of Algorithm 2.

Second, 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 24 of Algorithm 2). Otherwise, we turn to using the original initial hash function as shown by Steps $25-26$ 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 Fig. 5.

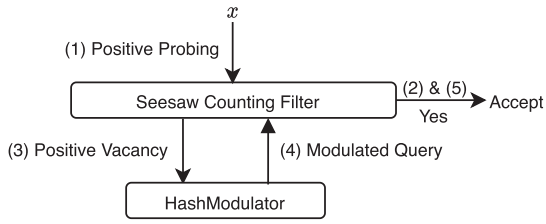


Fig. 7. Modulated query procedure.

F. Modulated Query Procedure

The query procedure is shown in Fig. 7 and Algorithm 3, which can be divided into two rounds as shown in Fig. 7. To ensure zero FNR[2], we restrict that a key is rejected if and only if it is rejected in both rounds. In the first round, the first Step (Positive Probing, Fig. 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 Fig. 7. Otherwise, the second round is activated if only one seesaw counter is probed to have an empty positive cell (Positive Vacancy, Fig. 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 $4-5$, Fig. 7). In other cases, the queried key is rejected.

Running Example: In Fig. 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.

The 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 first checks the k initial hash functions by probing their mapped seesaw counters. If all the probed positive cells are non-empty, the queried key 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 due to our one-modulating policy. Otherwise, if only one positive cell is probed to be empty,

[2]FNR is abbreviated from false negative rate, which refers to the probability rate of a positive key (keys in the given set) misidentified as a negative key (keys not in the given set).

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* ;

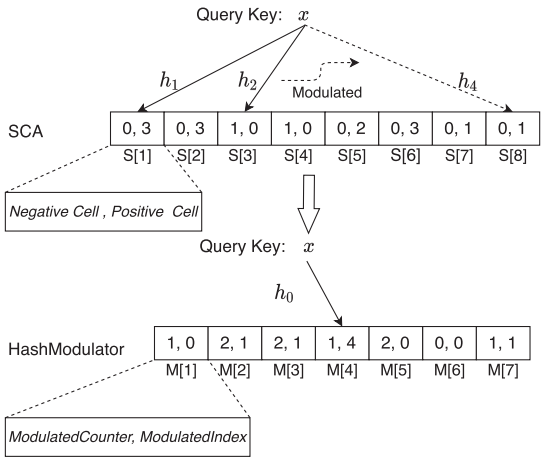


Fig. 8. Running example: Modulated query.

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 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).

G. Modulated Deletion Procedure

Another essential operation is deletion, which involves decreasing the mapped counters in SCA and removing the modulated hash function from HashModulator. However, the deletion procedure is non-trivial since whether the applied hash functions of a key are modulated is not recorded, which leads to FNR if the modulated hash functions are deleted mistakenly (or inconsistently). To handle the inconsistent deletion, we add a counter field named ModulatedCounter to record how many keys to be modulated are mapped to this cell and restrict that the stored modulated hash function can only be deleted when its ModulatedCounter is zero. By doing so, the inconsistent deletion will be addressed since the ModulatedCounter field indicates whether the stored modulated hash function in each cell is occupied by other keys. Besides, regarding the keys failing to be modulated, the HashedCounter field of their mapped cells will also be increased by one to prevent potential inconsistent deletion.

For a given key x to be deleted, how to delete it depends on whether its hash functions were modulated during its previous insertion procedure. In the first case that a key is not modulated, the deletion procedure is similar to that of Counting Bloom Filter, i.e., decreasing the positive cells of mapped seesaw counters in the underlying SCA. As for the second case that a key is modulated, the deletion procedure is slightly different and includes two parts: removing the modulated hash function in HashModulator, and then decreasing the counter fields of mapped cells.

The detailed pseudocode of deletion operation is illustrated in Algorithm 4, which includes two parts: (1) decreasing the seesaw counters from SCA (Steps $1-6$); and (2) removing modulated hash function from HashModulator if needed (Steps $7-18$).

In the first part, all initial k hash functions are applied to identify the hash function to be modulated, which is shown in Steps $2-6$ of Algorithm 4. If the hash function to be modulated is identified, its index is denoted as idx_{old} as shown by Step 6 of Algorithm 4. As for the other remaining hash functions, the positive cells of their mapped seesaw counters are decreased by one (Steps $4-5$, Algorithm 4).

In the second part, if no hash function identified in the first part, the deletion finishes directly (Steps $7-8$, Algorithm 4). Otherwise, with identified hash function to be modulated (i.e., indexed by idx_{old}), we need to retrieve its corresponding modulated hash function (indexed by idx_{new}) from HashModulator as shown in Steps $9-10$ of Algorithm 4. Afterwards, we need to check whether the retrieved hash function belongs to the key to be deleted, i.e., whether the retrieved hash function maps the key to a seesaw counter with empty negative cell and non-empty positive cell (Steps $11-12$, Algorithm 4). If yes, the positive cell of the mapped seesaw counter is decreased by one (Step 13, Algorithm 4), otherwise the positive cell of seesaw counter mapped by the identified hash function to be modulated (indexed by idx_{old}) is decreased by one (Steps $14-15$, Algorithm 4). Finally, if the ModulatedCounter field is zero, the modulated hash function stored in this mapped cell is removed since no key is occupying the cell, which is shown in Steps $16-18$.

Algorithm 4: Modulated Deletion(x).

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** $idx_{old} > 0$ *or the negative cell of counter* $\mathcal{S}[j]$ *is empty* **then**
5 Decrease the positive cell of counter $\mathcal{S}[j]$ by 1;
  else
6 $idx_{old} = i$; `// one-modulating policy`
7 **if** $idx_{old} == -1$ **then**
8 **return**;
  else
9 $j_{mod} = h_0(x)$;
10 $idx_{new} = \mathcal{M}[j_{mod}].ModulatedIndex$;
11 $j_{new} = H_{\mathcal{B}}[idx_{new}](x)$;
12 **if** *the seesaw counter* $\mathcal{S}[j_{new}]$ *has empty negative cell and non-empty positive cell* **then**
13 Decrease the positive cell of counter $\mathcal{S}[j_{new}]$ by 1;
  else
14 $j_{old} = H_A[idx_{old}](x)$;
15 Decrease the positive cell of counter $\mathcal{S}[j_{old}]$ by 1;
16 Decrease $\mathcal{M}[j_{mod}]$.ModulatedCounter by 1;
17 **if** $\mathcal{M}[j_{mod}]$.*ModulatedCounter* $== 0$ **then**
18 Remove $\mathcal{M}[j_{mod}]$.ModulatedIndex field;

Meanwhile, we denote the retrieved the index of the modulated hash function as idx_{new} as well as the index of its mapped cell as j_{new}. After that, the deletion procedure mainly goes through all hash functions one by one, and decreases the counter field of mapped cell by one, which is as shown by Steps $7-16$ of Algorithm 4. For each encountered hash function during the loop, it may be non-modulated or modulated. If the hash function is non-modulated, we directly decrease the counter field of its mapped cell by one, which is shown by Steps $9-10$ of Algorithm 4. As for the modulated hash function, the situation is a little different and needs to be handled according to whether the cells mapped are occupied, which can be divided into the following three cases.

Case 1. The counter field of the cell mapped by the raw hash function is zero: When the counter field of cell mapped by the raw hash function is probed to be zero, then raw hash function (indexed by idx_{old}) must be modulated since x is inserted. Therefore, we decrease the counter field of cell mapped by modulated hash function by one (Steps $11-13$, Algorithm 4).

Case 2. The counter field of the cell mapped by the modulated hash function is zero: When the counter field of cell mapped by the modulated hash function is probed to be zero, then raw hash function (indexed by idx_{old}) is modulated and the counter field of mapped cell by the raw hash function is then decreased by one (Steps $14-15$, Algorithm 4).

Case 3. Both of the counter fields of the cells mapped by the raw and modulated hash functions are non-zero: Concerning Case 3, there is no way to distinguish whether the modulated hash function is applied or not and thus lazy deletion (skip decrement step) is conducted (Step 16, Algorithm 4) to ensure zero false negative.

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

Fig. 9. Adaptive seesaw counter.

H. 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 at runtime. These newly inserted negative keys will be first stored into the adaptive-bits but will not be used during query time, which thus will not affect the cost-weighted FPR as shown in Fig. 9(a). Particularly, as shown in Fig. 9(b), the adaptive-bit of a seesaw counters will then be safely transported (or moved) to the negative cell once the positive cell (of the same seesaw counter) is empty because the empty seesaw counter is not being used by any positive key. These transported negative cell will be used during hash modulating process to achieve lower cost-weighted FPR, which is similar to our raw version SSCF as shown in Fig. 1. Besides, such transport may be blocked if the positive cell is not empty and will not be carried out until the positive cell decreases to zero due to deletion operations, which are frequent in dynamic scenarios. In other words, the performance of ada-SSCF improves over time. As shown in Fig. 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.

All operations of Ada-SSCF are the same as that of SSCF except the extra operation of adaptive encoding of vulnerable negative keys. 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. First, as shown in Fig. 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. Second, 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 Fig. 9(c), direct transportation is not feasible and will be blocked if the positive cell is non-empty. However, in dynamic scenarios, the key deletion operations are frequent, which involve decreasing the positive cells. With

enough key deletions, the positive cell of one seesaw counter will decrease to zero. Once the positive cell becomes empty, the key deletion operation needs to check the Adaptive-bit and resume the previously blocked transport procedure, i.e., setting the negative cell to 1 if the Adaptive-bit is 1.

IV. Theoretical Analysis

A. Ratio of Non-Empty Negative Cells

Lemma IV.1. The ratio of (non-empty negative cell) seesaw counters is $\mu \approx 1 - e^{-\frac{k}{B}\frac{N_n}{N_p}}$ in expectation, where k is the number of hash functions, B is the average number of counters per positive key in SCA, N_n is the number of negative keys to be encoded, and N_p is the capacity of positive keys in SSCF.

Proof. Considering that there are N_n negative keys, the probability that a seesaw counter is not mapped by all negative keys is $1-(1-\frac{1}{m})^{k\cdot N_n}$. Therefore, the ratio of negative seesaw counters is $\mu = 1-(1-\frac{1}{m})^{k\cdot N_n} \approx 1 - e^{-\frac{kN_n}{m}}$ in expectation. Besides, considering that the counter per key is B in SCA, then $\mu \approx 1 - e^{-\frac{k\cdot N_n}{B\cdot N_p}}$. □

B. Hash Modulating

Lemma IV.2. After inserting N_p positive keys, the expected number of triggered hash modulating is $\varphi = N_p \cdot (1-(1-\mu)^k)$.

Proof. With one positive key to be inserted, the hash function modulating is triggered if one or more applied hash functions map the inserted key to a negative seesaw counter, whose probability is $1-(1-\mu)^k$. By the linearity of expectation, with N_p positive keys, the expected number of triggered hash modulating is $\varphi = N_p \cdot (1-(1-\mu)^k)$. □

Lemma IV.3. Suppose there are φ times of hash modulating and $n = s\cdot\varphi$ cells in HashModulator, the load factor of HashModulator is $\mathcal{L} = (1-\mathrm{e}^{-\frac{1}{s}})$, where s is a constant scale factor that varies the space size of HashModulator.

Proof. Considering that each key is mapped to HashModulator with a public hash function (i.e., h_0), the load factor of HashModulator is then equivalent to the number of cells being occupied. We start with considering probability of the complementary case, i.e., a cell is not mapped by all keys to be modulated, which is given by $(1-\frac{1}{s\cdot\varphi})^{\varphi}$. That is to say, a cell will be occupied with probability $1-(1-\frac{1}{s\cdot\varphi})^{\varphi}$. By the linearity of expectation, the load factor $\mathcal{L}$ is $(1-(1-\frac{1}{s\cdot\varphi})^{\varphi}) \approx (1-\mathrm{e}^{-\frac{\varphi}{s\cdot\varphi}}) = (1-\mathrm{e}^{-\frac{1}{s}})$.

Lemma IV.4. When the load factor of HashModulator is $\mathcal{L}$, the probability that one hash function modulating is successful is $P_{ms} = \mathcal{L}(1-\mu) + (1-\mathcal{L})(1-\mu^{2^{\eta_2}})$, where μ is the ratio of negative seesaw counters in SCA and η_2 is the ModulatedIndex field size in HashModulator.

Proof. The hash modulating is successful if it switches one initial hash to a modulated hash function that maps to an empty or positive seesaw counter in SCA. According to the whether the key mapped by the key to insert, the hash modulating can be divided into the following two cases.

In the first case, if the mapped cell is empty in HashModulator (with probability $1-\mathcal{L}$), there will be at most 2^{η_2} backup modulated hash functions to be used since the ModulatedIndex is η_2-bit in size. Then the probability that the hash modulating is successful, i.e., at least one modulated hash function maps to empty or positive seesaw counter, is given by $P_{ms1} = (1-\mathcal{L})(1-\mu^{2^{\eta_2}})$.

In the second case, if the mapped cell is occupied in HashModulator (with probability $\mathcal{L}$), only modulated hash function stored in this cell can be used. Similar to the first case, the probability of a successful hash modulating is $P_{ms2} = \mathcal{L}(1-\mu)$. □

By combing P_{ms1} and P_{ms2}, the probability of a successful hash modulating is $P_{ms} = \mathcal{L}(1-\mu) + (1-\mathcal{L})(1-\mu^{2^{\eta_2}}) = (1-\mu^{2^{\eta_2}}) - L(\mu - u^{2^{\eta_2}})$.

Lemma IV.5. After inserting N_p positive keys, there is $\psi \approx 1 - e^{-\frac{k(1-\tau\cdot P_{ms})}{B}}$ ratio of negative seesaw counters with positive cell being non-empty in expectation, where τ is a constant coefficient.

Proof. When inserting a positive key, the positive cell of a negative seesaw counter is occupied if the hash modulating fails or more than one hash function maps the inserted key to negative seesaw counter at the same time during the insertion of one single key. First, as per Lemma IV.4, the probability of one unsuccessful modulating is $1-P_{ms}$. Second, we analyze the probability of more than one negative seesaw counter being mapped at the insertion of one single key. Without loss of generality, the applied k hash functions are assumed to be uniformly random, which indicates the number of negative seesaw counters being mapped during one positive key insertion obeys the Binomial distribution $B(k,\mu)$. Therefore, the number of negative seesaw counters mapped during the insertion of one single positive key, which is denoted as ψ_1 and is given by

$$
\begin{aligned}
\psi_1 &= \sum_{i=1}^{k}\binom{k}{i}\mu^i(1-\mu)^{k-i}((1-P_{ms})\cdot i + P_{ms}\cdot(i-1)) \\
&= \sum_{i=1}^{k}\binom{k}{i}\mu^i(1-\mu)^{k-i}(i-P_{ms}) \\
&= \sum_{i=0}^{k}\binom{k}{i}\mu^i(1-\mu)^{k-i}(i-P_{ms}) \\
&\quad - \binom{k}{0}\mu^0(1-\mu)^k(0-P_{ms}) \\
&= k\mu - P_{ms} + (1-\mu)^k P_{ms} \\
&= k\mu - P_{ms}(1-(1-\mu)^k). \qquad (1)
\end{aligned}
$$

Note that the obtained ψ_1 depends on P_{ms} that further depends on the number of inserted keys. Therefore, the number of hash functions mapped to negative seesaw counters can be formulated as $\sum_{i=1}^{N_p} i * \psi_1(i)$, where i denotes the number of positive keys already being inserted and $\psi_1(0) \le \psi_1(i) \le \psi_1(N_p)$. Then, the ratio of negative seesaw counters with positive cells also being occupied can be bounded by $\psi \le 1-(1-\frac{1}{m\mu})^{N_p\cdot\psi_1}$, where ψ_1 is short for $\psi_1(N_p)$ for the sake of concision. Moreover,

the value of $\sum_{i=1}^{N_p} i * \psi_1(i)$ can be explicitly calculated and thus denoted as $\tau \cdot \psi_1 \cdot N_p$ for simplicity, where τ is a constant coefficient. Then, the ratio of negative seesaw counters with positive cells also being occupied is $\psi = 1 - (1 - \frac{1}{m\mu})^{\tau \cdot N_p \cdot \psi_1} = 1 - e^{-\frac{k(1-\tau \cdot P_{ms})}{B}}$.

This completes the proof. □

Lemma IV.6. After inserting N_p positive keys, $\lambda = 1 - e^{-\frac{k}{B}(1+\frac{\mu}{1-\mu}P_{ms})}$ ratio of non-negative seesaw counters are with non-empty positive cell in expectation.

Proof. For a given empty seesaw counter in SCA, it is set either if mapped by the initial hash functions or if mapped by modulated hash functions. Meanwhile, during the insertion of N_p positive keys, the number of applied initial hash functions is $(1-\mu)N_pk$ and the number of applied modulated hash functions is $\varphi \cdot P_{ms}$ as per Lemma IV.1 and IV.4. Therefore, the probability of a non-negative seesaw counter with non-empty positive cell is

$$\lambda = 1 - \left(1 - \frac{1}{(1-\mu)m}\right)^{(1-\mu)N_pk+\varphi \cdot P_{ms}} \approx 1 - e^{-\frac{k+\frac{k\mu}{1-\mu}P_{ms}}{B}} = 1 - e^{-\frac{k}{B}(1+\frac{\mu}{1-\mu}P_{ms})}.$$

This completes the proof. □

C. Cost-Weighted FPR of SSCF

Theorem IV.1. When querying a vulnerable negative key, its FPR is $FP_{\mathcal{V}} = \psi^{k-1}(\psi + (1-\psi)k\lambda(1-\mathrm{e}^{-\frac{1}{s}})(1-\mu))$.

Proof. When querying a vulnerable negative key, it is identified to be positive if it is misidentified with initial hash functions or modulated hash functions. First, with initial hash functions, a false positive occurs when all the mapped seesaw counters are with non-empty positive cells, whose probability is given by $FP_{\mathcal{V}_1} = \psi^k$, where ψ is the ratio of seesaw counters with non-empty negative and positive cell as shown in Lemma IV.5. Second, if a vulnerable negative key is misidentified, there should be only one initial hash function mapping to a seesaw counter with empty positive cell, whose probability is given by $FP_{\mathcal{V}_2} = k\psi^{k-1}(1-\psi) \cdot (\mathcal{L}\lambda)(1-\mu)$. By combing $FP_{\mathcal{V}_1}$ and $FP_{\mathcal{V}_2}$, we have

$$\begin{aligned} FP_{\mathcal{V}} &= FP_{\mathcal{V}_1} + FP_{\mathcal{V}_2} = \psi^k + k\psi^{k-1}(1-\psi) \cdot (\mathcal{L}\lambda) \\ &= \psi^{k-1}\left(\psi + (1-\psi)(1-\mu)k\lambda\left(1-\mathrm{e}^{-\frac{1}{s}}\right)\right). \end{aligned} \quad (2)$$

This completes the proof. □

Theorem IV.2. When querying a non-vulnerable negative key, its FPR is $FP_{\mathcal{N}} = (\mu\psi + (1-\mu)\lambda)^{k-1}((\mu\psi + (1-\mu)\lambda) + (1 - (\mu\psi + (1-\mu)\lambda))k(1-\mu)\lambda(1-\mathrm{e}^{-\frac{1}{s}}))$.

Proof. Similar to that of querying a vulnerable negative key, a non-vulnerable negative key is misidentified if it is either misidentified with initial or modulated hash functions. For the initial hash functions, a false positive occurs when all mapped cells are occupied, which has probability $FP_{\mathcal{N}_1} = (\mu\psi + (1-\mu)\lambda)^k$. As for the modulated hash functions, its probability is $FP_{\mathcal{N}_2} = k(\mu\psi + (1-\mu)\lambda)^{k-1}(1 - (\mu\psi + (1-\mu)\lambda))\mathcal{L}(1-\mu)\lambda$. Therefore, the FPR of querying non-vulnerable negative keys can be derived as

$$\begin{aligned} FP_{\mathcal{N}} &= FP_{\mathcal{N}_1} + FP_{\mathcal{N}_2} \\ &= (\mu\psi + (1-\mu)\lambda)^{k-1}\left((\mu\psi + (1-\mu)\lambda)\right. \\ &\quad \left. + (1 - (\mu\psi + (1-\mu)\lambda)) \cdot k(1-\mu)\lambda\left(1-\mathrm{e}^{-\frac{1}{s}}\right)\right). \end{aligned} \quad (3)$$

This completes the proof. □

Theorem IV.3. Suppose that the encoded N_n negative keys account for ϵ ratio of all costs, the cost-weighted FPR of SSCF is $CFPR(B,s) = \epsilon \cdot FP_{\mathcal{V}} + (1-\epsilon) \cdot FP_{\mathcal{N}}$, where B is the seesaw counters per key in SCA and s is the scale factor of HashModulator.

Proof. The cost-weighted FPR is composed of two parts: the false positive of vulnerable negative keys and non-vulnerable keys. By the linearity of expectation, the cost-weighted FPR arising from vulnerable keys is $\epsilon \cdot FP_{\mathcal{V}}$, where ϵ is the ratio of costs of the encoded vulnerable negative keys. Similarly, as per Theorem IV.2, the cost-weighted FPR of non-vulnerable key is $(1-\epsilon) \cdot FP_{\mathcal{N}}$. Therefore, the cost-weighted FPR of SSCF is $\epsilon \cdot FP_{\mathcal{V}} + (1-\epsilon) \cdot FP_{\mathcal{N}}$. □

D. Parameter Optimization

Suppose the memory space budget is M, the HashModulator is allocated with $\alpha \cdot M$ space and SCA is allocated with $(1-\alpha)M$ ($\alpha \in [0,1]$). Meanwhile, there are m seesaw counters in SCA of SSCF and each seesaw counter is composed of two fields, including θ_1-bit negative cell and θ_2-bit positive cell. Besides, the positive key capacity of SSCF is N_p and each positive key is allocated with B seesaw counters cells. Therefore, we have $(1-\alpha)M = m \cdot (\theta_1 + \theta_2) = N_p \cdot B \cdot (\theta_1 + \theta_2)$. Then α can be formulated as

$$\alpha = 1 - \frac{BN_p(\theta_1 + \theta_2)}{M}. \quad (4)$$

Concerning HashModulator, it is allocated with $(1-\alpha)M$ space with n cells, each of which has two fields: η_1-bit *ModulatedCounter* and η_2-bit *ModulatedIndex*. As per Lemma IV.3, we can reformulate n as

$$n = sN_p\left(1 - e^{-\frac{k^2}{B}\frac{N_n}{N_p}}\right) \approx \frac{s \cdot k^2 \cdot N_n}{B}. \quad (5)$$

Besides, the space of HashModulator is αM, which consists of n ($(\eta_1+\eta_2)$-bit in size) cells of size. Therefore, we have $\alpha M = n(\eta_1 + \eta_2)$ and s can be reformulated as $s = \frac{\alpha M}{(\eta_1+\eta_2)(\frac{k^2 \cdot N_n}{B})} = \frac{MB - B^2N_p(\theta_1+\theta_2)}{(\eta_1+\eta_2)(k^2 \cdot N_n)}$. Finally, as per Theorem IV.3, the problem of how to obtain an optimal cost-weighted FPR regarding space ratio allocated to HashModulator can be formulated as follows:

$$\begin{aligned} &\min CFPR(B,s) = \epsilon \cdot FP_{\mathcal{V}} + (1-\epsilon) \cdot FP_{\mathcal{N}} \\ &s.t. s = \frac{MB - B^2N_p(\theta_1+\theta_2)}{(\eta_1+\eta_2)(k^2 \cdot N_n)}, \quad B \in \left[1, \frac{M}{(\theta_1+\theta_2)N_p}\right]. \end{aligned} \quad (6)$$

Note that similar to standard Counting Bloom filter, we set $\theta_1 = 1$ and $\theta_2 = 4$ by default in this article. How to set k, η_1, η_2 will be discussed in our evaluations (Section V-B) and we focus on how to obtain an optimized B. Considering that (6) can be evaluated with $O(1)$ time complexity, the parameter B is an integer within range $(1, \frac{M}{(\theta_1+\theta_2)N_p})$. Therefore, the optimal B can be found within $o(log_2^{\frac{M}{(\theta_1+\theta_2)N_p}})$ time with binary search. Finally, with the obtained optimal B, the optimal ratio α of space allocated to HashModulator can be derived by plugging B into (4).

V. Experiment Evaluation

In this section, extensive experiments are conducted to validate the effectiveness of SSCF. Overall, we want to answer the five key questions: (1) how do the parameters affect the performance of SSCF? (2) when to adopt SSCF? (3) how does SSCF 's performance (i.e., accuracy and operation latency) compared with other filters? and (4) how does the performance of ada-SSCF evolves in the dynamic scenarios?

A. Experimental Setup

1) Comparison Filter Implementation: There are three comparison filters, whose implementation details are as follows. The first filter is Counting Bloom filter (CBF) [3]. By inheriting optimal the parameter setting of CBF, we set the number of hash functions $k = \lfloor \ln 2 \cdot B \rfloor$ 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 [3], which is also inherited by SSCF. The second comparison filter is Weighted Bloom filter (WBF) [12]. 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). Besides, as WCBF takes in the costs of queried keys as input, a default portion (about 0.005%) of keys with the highest cost are stored. Note that storing more keys consumes too much space since the keys (or its ID like URL) may be very long in bits. The third filter is Stacked Filter (SF) [13]. 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 the 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 [13], 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 (including the auxiliary data structures) 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 to the total memory space size in bits divided by the positive key number.

2) Experimental Infrastructure: We evaluate all the filters on the same Linux server equipped with Intel Xeon Gold 5218R (2.10GH with 80 cores), 125 GB RAM. All filters are implemented with C++ and compiled with the same configurations. The results reported are obtained with 10 repetitions.

TABLE III
Dataset Summary

Data Set	# Keys	# Positive Keys	# Negative Keys
Shalla	2,927,472	1,491,178	1,435,527
YCSB	24,074,812	12,500,611	11,574,201
CAIDA	10,607,107	5,603,095	5,004,012

3) 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 $C(x)$ is the cost of key x.

4) Data Sets: To validate the effectiveness of SSCF, the following two data sets (summarized in Table III) are used:

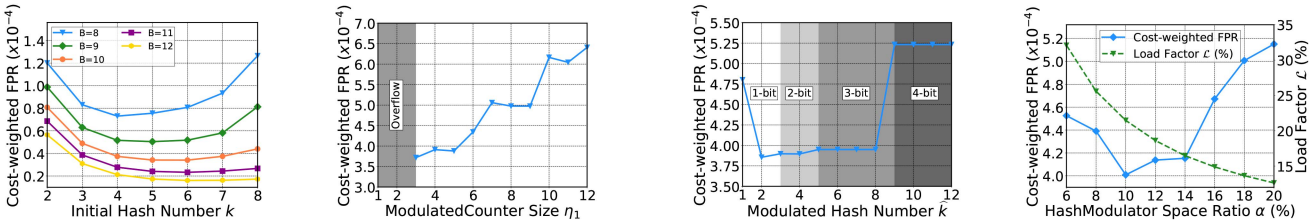
1) *Shalla's Blacklists.* Shalla's Blacklists [23] (abbreviated as Shalla) is a URL dataset, which contains malicious URLs to be blocked and used here to simulate building a dynamic white-list URLs that can be accessed safely.
2) *YCSB.* YCSB is a benchmark [24] designed by Yahoo for performance evaluating of key-value stores.
3) *CAIDA.* CAIDA [25] is a set of collected Internet traffics and hosted by University of California's San Diego Supercomputer Center.

5) Cost Distribution: Considering that the keys from Shalla and YCSB have no default cost, a skewed cost distribution (i.e., Zipf distribution [26]) is generated. To reveal the correlation between the cost skewness and filtering performance, various skewness parameters (from 1.0 to 2.5) are generated.

B. 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. Besides, a portion (5%) of negative keys with the highest costs are selected as the vulnerable negative keys.

1) *Number of Initial Hash Functions k:* In Fig. 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 $B = 12$, 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. This is because a modestly large k increases number of cells being checked and thus reduces cost-weighted FPR while an oversized k leads to more occupied counters and thus higher cost-weighted FPR. Considering that the optimal hash function configuration



(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

Fig. 10. Parameter evaluation.

is similar to that of CBF, we directly inherit the optimal hash function number setting from CBF by setting, where the B is the counter per positive key in SCA.

2) *ModulatedCounter Size η_1:* As is shown in Fig. 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 Fig. 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. Therefore, we set $\widehat{k} = 2$ and $\eta_2 = 1$ in the following evaluations.

4) *HashModulator Space Ratio α:* As is shown in Fig. 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.

C. When to Adopt SSCF?

In this subsection, we study when to adopt SSCF or more exactly when will SSCF surpass other filters (CBF as an example), which mainly depends on the cost distribution (i.e., skewness) and the encoded negative key number N_n.

SSCF Benefits from Large Skewness: First, to explore how does the skewness affect SSCF, we take the Zipf distribution as an example and vary its skewness from 0.2 to 2.6. As shownin Fig. 11, for the skewness ≤ 0.6, CBF shows better cost-weighted FPR as the encoded negative keys do not account for a large portion of cost. As for the skewness larger than 0.6, SSCF shows

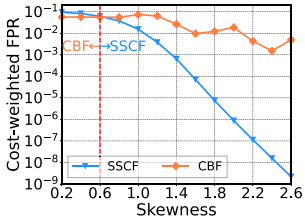


Fig. 11. Cost-weighted FPR versus skewness.

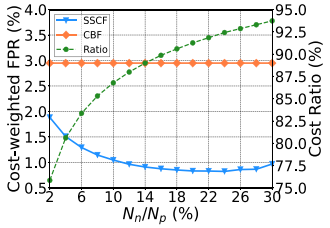


Fig. 12. Cost-weighted FPR versus $\frac{N_n}{N_p}$ with 1.0 skewness.

lower cost-weighted FPR and the performance gain continues to enlarge exponentially. This is because for large skewness, these encoded vulnerable negative keys accounts for the major costs, which are actually taken special care of by SSCF.

SSCF Handles A Considerable Number of Vulnerable Negative Keys: Second, we focus on the parameter N_n, i.e., the number of vulnerable negative keys. For the sake of clarity, instead of considering N_n directly, we turn to its normalized formulation $\frac{N_n}{N_p}$, where N_p is the positive key capacity in SSCF. As shown in Fig. 12, we vary $\frac{N_n}{N_p}$ from 2% to 30%. The optimal cost-weighted FPR is achieved when $\frac{N_n}{N_p} = 22\%$. For $\frac{N_n}{N_p} < 22\%$, the encoded negative keys do not cover the major cost and lead to worse cost-weighted FPR. For $\frac{N_n}{N_p} > 22\%$, the increased number of encoded negative keys requires more space for HashModulator, which leads to worse cost-weighted FPR. In summary, with larger cost encapsulated by smaller (i.e., larger skewness) $\frac{N_n}{N_p}$ ratio of negative keys, SSCF is preferred.

D. Overall Filtering Performance

In this experiment, we evaluate all filters by varying the space size (indicated by the bits per positive key), and Zipf cost distribution skewness. Specifically, the bits per positive key is increased from 20 to 36 and the skewness is increased from 1.0 to 2.5.

SSCF always has the smallest cost-weighted FPR under all space settings and outperforms all the comparison filters at least by $1.55\times$ and up to two orders of magnitude on skewed data: For Shalla with skewness 1.0, as shown in Fig. 13(a), the cost-weighted FPR of SSCF decreases from 2.99% to 0.57%. Among other filters, the SF shows the best performance with

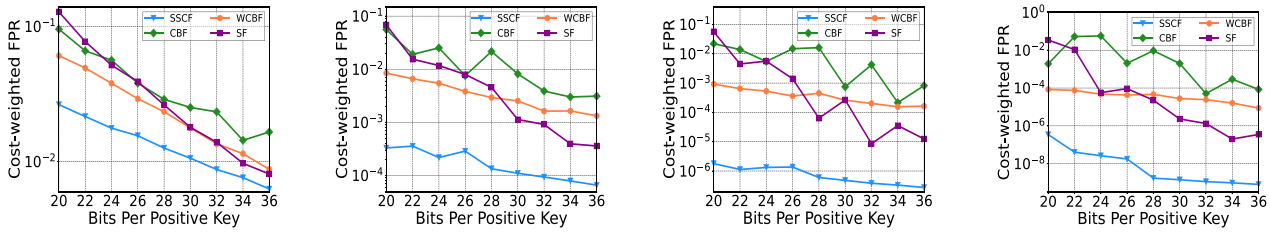


Fig. 13. Cost-weighted FPR on shalla versus skewness under the same memory space (varying from 3.5MB to 6.4MB).

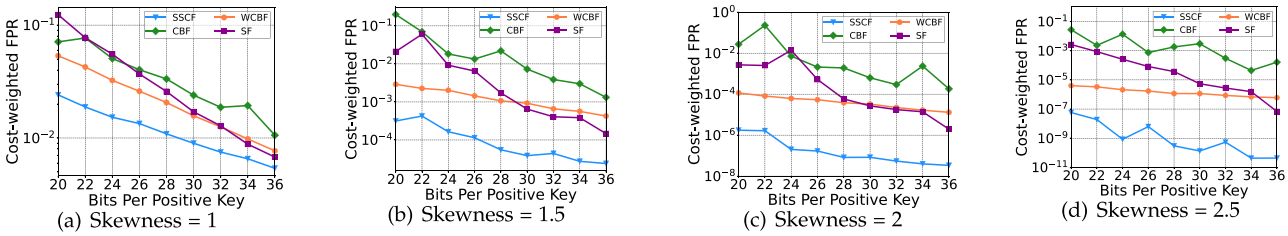


Fig. 14. Cost-weighted FPR on YCSB versus skewness under the same memory space (varying from 29.8MB to 53.6MB).

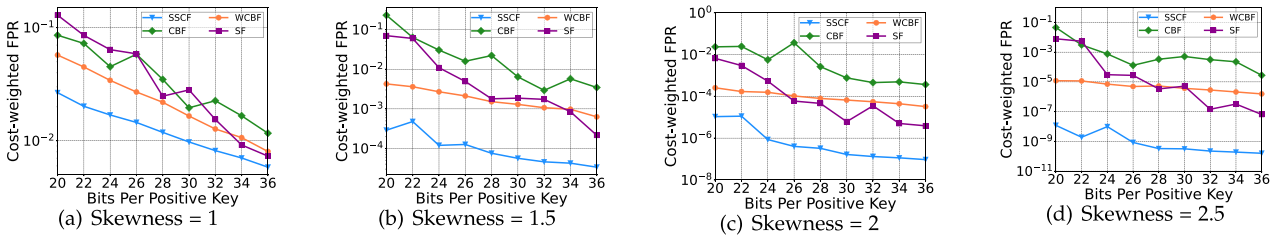


Fig. 15. Cost-weighted FPR on CAIDA versus skewness under the same memory space (varying form 12MB to 22MB).

cost-weighted FPR decreasing from 5.62% to 0.63%. Even compared with SF, SSCF shows over $1.55\times$ performance improvement and obtains larger performance gain under larger skewness, which is as shown in Fig. 13(b), (c), and (d). Particularly, SSCF outperforms SF over two orders of magnitude with 2.5 Zipf skewness, which is as shown in Fig. 13(d).

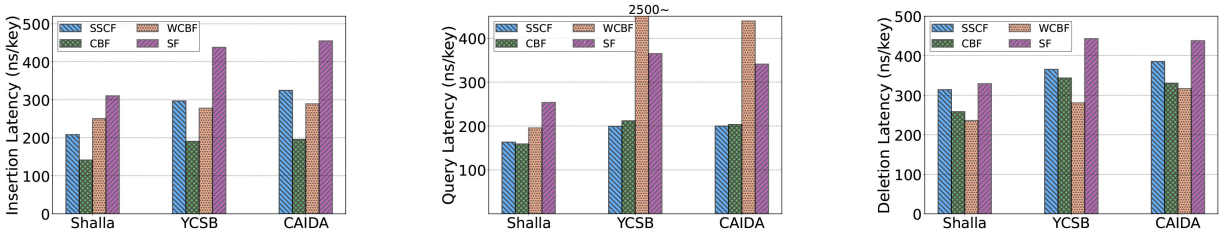
As is shown in Fig. 14(a), for YCSB with skewness 1.0, the cost-weighted FPR of SSCF decreases from 2.35% to 0.51%. Similar to that on Shalla, SF also shows the best performance with cost-weighted FPR decreasing from 4.80% to 0.54% on YCSB with Zipf skewness 1.0. Meanwhile, SSCF outperforms all the other comparison filters by at least $1.45\times$. With the increased cost distribution skewness, the performance gap enlarges as shown in Fig. 14(b), (c), and (d). As is shown in Fig. 14(d), on YCSB with 2.5 Zipf skewness, SSCF also outperforms all other comparison filters by over two orders of magnitude.

As for CAIDA, the evaluation results are similar, which is reported in Fig. 15(a), (b), (c), and (d). Specifically, with the increased cost distribution skewness, our proposed SSCF consistly achieves the best weighted-FPR with enlarged performance gaps compared with other filters.

E. Operation Latency

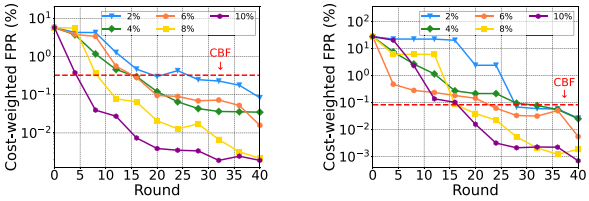
1) Insertion Latency: The insertion latency of SSCF is about $1.47\times$ *the latency of CBF:* As shown in Fig. 16(a), on Shalla, the insertion latency per key is 208ns for SSCF, 141ns for CBF, 250ns for WCBF, and 310ns for SF. As for YCSB, the insertion latency is 296ns for SSCF, 190ns for CBF, 277ns for WCBF, and 438ns for SF. on CAIDA, the insertion latency is 326ns for SSCF, 197ns for CBF, 283ns for WCBF, and 458ns for SF. The insertion latency of SSCF is comparable to CBF since the insertion procedure of SSCF is very similar to that of CBF except the extra hash modulating procedure, which involves very limited times of hash function computation and memory access.

2) Query Latency: The query latency of SSCF is similar to the latency of CBF. The query latency per key on Shalla, as shown in Fig. 16(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. As for CAIDA, the insertion latency for SSCF, CBF, WCBF and SF is 195ns, 209ns, 448ns and 334ns, respectively. At query time, SSCF, even with modulated query procedure, only applies the roughly at



(a) Insertion latency (b) Query latency (c) Deletion latency

Fig. 16. Operation latency.



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

Fig. 17. Ada-SSCF versus cost-weighted FPR.

most one more hash function, which makes the query latency of be comparable to CBF's query latency. WCBF suffers from prolonged 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. Moreover, SF shows large query latency since at least two or three layers of filters need to be queried for both positive and negative keys.

3) Deletion Latency: The deletion latency of SSCF is about $1.21\times$ *of CBF's deletion latency:* As shown in Fig. 16(c), the deletion latency on Shalla is 314 ns, 258 ns, 236 ns, and 329 ns for SSCF, CBF, WCBF and SF, respectively. On YCSB, the latency is 354ns for SSCF, 345 ns for CBF, 280 ns for WCBF and 443 ns for SF. As for CAIDA, the deletion latency is 385 ns for SSCF, 320 ns for CBF, 315 ns for WCBF, and 430 ns for SF. We note that benefited from fewer (1 less) applied hash functions, SSCF achieves smaller latency compared with CBF. Besides, similarly, the large deletion latency of SF comes from the overhead of handling multiple filters of different layers.

F. 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 Fig. 17. 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 Fig. 17(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 Fig. 17(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 can even improve much faster.

VI. Related Work

Filters That are Cost-Aware: The standard Bloom filter [22] and its variants do not take into account the key costs [3], [8], [27], [28], which makes all negative keys are treated identically. Particularly, the CQF (counting quotient filter) [28], although being more space-efficient compared with CBF (counting Bloom filter), also ignores the negative keys as well as the key costs during construction time, which makes CQF achieves suboptimal performance even when vulnerable negative keys can be obtained. 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) [12]. 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 [29], which can only work offline. The recent proposed stacked filter framework [13] 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 [30] and SuRF [31] are designed for range query.

Filters That are Learning-Based: Recently, there have been several works that propose to utilize machine learning model in the filter design. With an elaborately trained learned model, existing learning-based works could achieve beyond the

theoretical limit performance in terms of FPR and space efficiency [9], [32]. Kraska et al. proposed a learned Bloom filter [9] to obtain optimized space efficiency by incorporating machine techniques that can capture data distribution information within a small learned model. However, the high space efficiency achieved by the learning-based filter is at the sacrifice of construction and query latency, which is unacceptable in the dynamic scenarios. To reduce query latency, Mitzenmacher proposed to add an initial Bloom filter on top of the learned Bloom filter to reject the frequent queried negative keys as early as possible [32]. 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 [11]. 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 [14].

VII. Conclusion

In this article, 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 and other variants on the whole regarding accuracy, construction time, query latency and filter size.

References

[1] SSCF authors, "SSCF source code," 2021. [Online]. Available: https://github.com/njulimn/SSCF

[2] B. Goodwin et al., "Bitfunnel: Revisiting signatures for search," in *Proc. Int. Conf. Res. Develop. Inf. Retrieval*, 2017, pp. 605–614.

[3] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *Trans. Netw.*, vol. 8, no. 3, pp. 281–293, 2000.

[4] Y. Lu, B. Prabhakar, and F. Bonomi, "Perfect hashing for network applications," in *Proc. IEEE Int. Symp. Inf. Theory*, 2006, pp. 2774–2778.

[5] D. L. Quoc et al., "ApproxJoin: Approximate distributed joins," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 426–438.

[6] Google, "Leveldb a fast and lightweight key/value database library," 2011. [Online]. Available: http://code.google.com/p/leveldb/

[7] Facebook, "A facebook fork of leveldb which is optimized for flash and big memory machines," 2013. [Online]. Available: https://rocksdb.org/

[8] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *Proc. Eur. Symp. Algorithms*, 2006, pp. 684–695.

[9] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proc. Int. Conf. Manage. Data*, 2018, pp. 489–504.

[10] K. Singhal and P. Weiss, "DeepBloom," 2020. [Online]. Available: https://github.com/karan1149/DeepBloom/tree/master/data

[11] Z. Dai and A. Shrivastava, "Adaptive learned Bloom filter (Ada-BF): Efficient utilization of the classifier," 2019, *arXiv:1910.09131*.

[12] J. Bruck, J. Gao, and A. Jiang, "Weighted Bloom filter," in *Proc. IEEE Int. Symp. Inf. Theory*, 2006, pp. 2304–2308.

[13] K. Deeds, B. Hentschel, and S. Idreos, "Stacked filters: Learning to filter by structure," in *Proc. Int. Conf. Very Large Data Bases*, 2020, pp. 600–612.

[14] R. Xie et al., "Hash adaptive bloom filter," in *Proc. IEEE Int. Conf. Data Eng.*, 2021, pp. 636–647.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, pp. 29–43, 2003.

[16] D. Charles and K. Chellapilla, "Bloomier filters: A second look," in *Proc. Eur. Symp. Algorithms*, 2008, pp. 259–270.

[17] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Proc. IEEE Int. Symp. Comput. Architecture*, 2005, pp. 494–505.

[18] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," *Trans. Netw.*, vol. 14, no. 2, pp. 397–409, 2006.

[19] Bitcoin Optech, "Bitcoin optech newsletter #60," 2019. [Online]. Available: https://bitcoinops.org/en/newsletters/2019/08/21/#bitcoin-core-16248

[20] P. A. Networks, "Palo alto networks malicious IP address feeds," 2019. [Online]. Available: 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

[21] S. Pontarelli, P. Reviriego, and J. A. Maestro, "Improving counting bloom filter performance with fingerprints," *Inf. Process. Lett.*, vol. 116, no. 4, pp. 304–309, 2016.

[22] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[23] S. S. S. KG, "Shalla's blacklists," 2022. [Online]. Available: http://mirror.netlinux.cl/shallalist/

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. Symp. Cloud Comput.*, 2010, pp. 143–154.

[25] University of California's San Diego Supercomputer Center, "Caida dataset," 2021. [Online]. Available: https://www.caida.org/

[26] D. M. Powers, "Applications and explanations of Zipf's law," in *New Methods in Language Processing and Computational Natural Language Learning*. Toronto, Canada: Association for Computational Linguistics, 1998.

[27] H. Dai, Y. Zhong, A. X. Liu, W. Wang, and M. Li, "Noisy bloom filters for multi-set membership testing," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Sci.*, 2016, pp. 139–151.

[28] P. Pandey, M. A. Bender, R. Johnson, and R. Patro, "A general-purpose counting filter: Making every bit count," in *Proc. ACM Int. Conf. Manage. Data*, Chicago Illinois USA: ACM, 2017, pp. 775–787.

[29] M. Zhong, P. Lu, K. Shen, and J. Seiferas, "Optimizing data popularity conscious Bloom filters," in *Proc. Symp. Princ. Distrib. Comput.*, 2008, pp. 355–364.

[30] 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 *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Portland OR USA: ACM, 2020, pp. 2071–2086.

[31] H. Zhang et al., "SuRF: Practical range query filtering with fast succinct tries," in *Proc. Int. Conf. Manage. Data*, Houston TX USA: ACM, 2018, pp. 323–336.

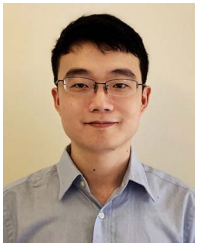
[32] M. Mitzenmacher, "A model for learned Bloom filters and optimizing by sandwiching," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 464–473.

Meng Li received the BS degree in computer science from Nanjing University, Jiangsu, China, in 2016. He is currently working toward the PhD degree with Nanjing University. His research interests are in the area of high-performance query processing in databases.

Deyi Chen received the bachelor's degree from the Nanjing University of Science and Technology, Jiangsu, China, in 2020. He is currently working toward the master's degree in Nanjing University. His reaserch interests include sketches and data stream processing.

Haipeng Dai received the BS degree in the Department of Electronic Engineering from Shanghai Jiao Tong University, Shanghai, China, in 2010, and the PhD degree in the Department of Computer Science and Technology in Nanjing University, Nanjing, China, in 2014. His research interests are mainly in the areas of data mining, Internet of Things, and mobile computing. He is an associate professor in the Department of Computer Science and Technology in Nanjing University.

Rongbiao Xie received the BS degree in the Department of Automation, Xiamen University. He is currently working toward the master's degree in computer science from Nanjing University, Jiangsu, China. His research interests are in the optimization of database index and the acceleration of storage Engine.



Siqiang Luo (Member, IEEE) received the bachelor's and master's degrees from Fudan University, in 2010 and 2013, respectively, and the PhD degree in computer science from the University of Hong Kong, in 2019. He is currently an assistant professor with the School of Computer Science and Engineering, Nanyang Technological University. He was a postdoc with Harvard University from 2019 to 2020. His research interests include data management and data mining.

Rong Gu (Member, IEEE) received the PhD degree from Nanjing University, China, in 2016. He is an associate research professor in Nanjing University. His research interests include parallel and distributed computing, big data systems. His research papers have been published in many conference and journals, including *IEEE Transactions on Parallel and Distributed Systems*, IEEE ICDE, WWW, IEEE IPDPS, IEEE ICPP, JSA, Parallel Computing, and JPDC.

Tong Yang (Member, IEEE) received the PhD degree in computer science from Tsinghua University, in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently a associate professor in School of Computer Science, Peking University. He published articles in SIGCOMM, SIGKDD, SIGMOD, SIGCOMM CCR, VLDB, ATC, ToN, ICDE, and INFOCOM. His research interests include network measurements, sketches, IP lookups, Bloom filters, sketches, and KV stores.

Guihai Chen (Fellow, IEEE) received the BS degree in computer software from Nanjing University, in 1984, and the ME degree in computer applications from Southeast University, in 1987, and the PhD degree in computer science from the University of Hong Kong, in 1997. He is a professor and deputy chair of the Department of Computer Science, Nanjing University, China. He had been invited as a visiting professor by many foreign universities, including Kyushu Institute of Technology, Japan, in 1998, University of Queensland, Australia, in 2000, and Wayne State University, USA from Sep. 2001 to Aug. 2003. He has a wide range of research interests focusing on sensor networks, peer-to-peer computing, high-performance computer architecture, and combinatorics.