Published: 10 March 2022
Accepted: 01 November 2021
Revised: 01 September 2021
Received: 01 April 2021

RESEARCH-ARTICLE

BhBF: A Bloom Filter Using Bh Sequences for Multi-set Membership Query

SHUYU PEI, Hunan University, Changsha, Hunan, China

KUN XIE, Hunan University, Changsha, Hunan, China

XIN WANG, Stony Brook University, Stony Brook, NY, United States

GAOGANG XIE, Chinese Academy of Sciences, Beijing, Beijing, China

KENLI LI, Hunan University, Changsha, Hunan, China

WEI LI, Hunan University, Changsha, Hunan, China

View all

B_hBF: A Bloom Filter Using B_h Sequences for Multi-set Membership Query

SHUYU PEI and KUN XIE, the College of Computer Science and Electronic Engineering, Hunan University
XIN WANG, the Department of Electrical and Computer Engineering, the State University of New York at Stony Brook
GAOGANG XIE, the Computer Network Information Center, Chinese Academy of Sciences
KENLI LI and WEI LI, the College of Computer Science and Electronic Engineering, Hunan University
YANBIAO LI, the Computer Network Information Center, Chinese Academy of Sciences
JIGANG WEN, the Institute of Computing Technology, Chinese Academy of Sciences

Multi-set membership query is a fundamental issue for network functions such as packet processing and state machines monitoring. Given the rigid query speed and memory requirements, it would be promising if a multi-set query algorithm can be designed based on Bloom filter (BF), a space-efficient probabilistic data structure. However, existing efforts on multi-set query based on BF suffer from at least one of the following drawbacks: low query speed, low query accuracy, limitation in only supporting insertion and query operations, or limitation in the set size. To address the issues, we design a novel B_h sequence-based Bloom filter (B_hBF) for multi-set query, which supports four operations: insertion, query, deletion, and update. In B_hBF, the set ID is encoded as a code in a B_h sequence. Exploiting good properties of B_h sequences, we can correctly decode the BF cells to obtain the set IDs even when the number of hash collisions is high, which brings high query accuracy. In B_hBF, we propose two strategies to further speed up the query speed and increase the query accuracy. On the theoretical side, we analyze the false positive and classification failure rate of our B_hBF. Our results from extensive experiments over two real datasets demonstrate that B_hBF significantly advances state-of-the-art multi-set query algorithms.

CCS Concepts: • **Theory of computation** → *Data structures design and analysis;*

Additional Key Words and Phrases: Multi-set membership query, bloom filter

This work was supported in part by the National Natural Science Foundation of China under Grant 62025201, Grant 61972144; in part by the NSF Electrical, Communications and Cyber Systems (ECCS) under Grant 1731238 and Grant 2030063; in part by the NSF Communication and Information Foundations (CIF) under Grant 2007313; and in part by the Hunan Provincial Innovation Foundation for Postgraduate Studies under Grant CX20200437.

Authors' addresses: S. Pei and K. Xie (corresponding author), the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, Hunan Province, China; emails: {peishuyu, xiekun}@hnu.edu.cn; X. Wang, the Department of Electrical and Computer Engineering, the State University of New York at Stony Brook, Stony Brook, NY 11794, USA; email: x.wang@stonybrook.edu; G. Xie and Y. Li, the Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China; emails: xie@ict.ac.cn, lybmath@cnic.cn; K. Li and W. Li, the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, Hunan Province, China; emails: {lkl, rj_wli}@hnu.edu.cn; J. Wen, the Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China; email: wenjigang@ict.ac.cn.

1556-4681/2022/03-ART89 $15.00
https://doi.org/10.1145/3502735

ACM Reference format:
Shuyu Pei, Kun Xie, Xin Wang, Gaogang Xie, Kenli Li, Wei Li, Yanbiao Li, and Jigang Wen. 2022. B_hBF: A Bloom Filter Using B_h Sequences for Multi-set Membership Query. *ACM Trans. Knowl. Discov. Data.* 16, 5, Article 89 (March 2022), 26 pages.
https://doi.org/10.1145/3502735

1 INTRODUCTION

1.1 Problem and Motivation

Multi-set membership query is a fundamental operation for network applications and computing systems. Network applications, such as network packet classification, forming approximate state machines and distributed caching, often involve multi-set membership query. Given L independent sets $S = \{S_1, S_2, \ldots, S_L\}$ with $S_i \cap S_j = \emptyset$ where $1 \leq i < j \leq L$, multi-set membership query replies which set a queried element **e** belongs to or **e** does not belong to any set. An element **e** can be a string, an integer, or an IP address, and S_i can be a category, a port, and so on. Following are some typical examples.

- **Packet processing** [12, 27, 29]. A router classifies packets it receives before forwarding at the network layer. After parsing the packet, it quickly finds the corresponding forwarding port and forwards the packet out. In this case, the data packet is the element to query, and the correct port needs to be returned from the set of possible ports.
- **State machines monitoring** [3, 25]. When monitoring the connection states of TCP flows for attack detection, a specified TCP flag may indicate the potential problem. For example, when a large number of TCP flows on the server are in the semi-connected state of SYN-RECEIVED, it may indicate potential SYN attack. Here, the TCP flow is the element and the states represented by flags (such as ACK, RST, SYN, and FIN) form the set.

The advance of high-speed networks and applications bring the following requirements, which makes the design of an algorithm for multi-set membership query challenging.

- **Supporting high speed and accurate query with low memory overhead.** (a) The algorithm processing speed needs to keep up with the speed of the network. (b) The data structure used by the algorithm should be compact to fit in the fast but small on-chip memory.
- **Supporting four operations including insertion, query, deletion, and update.** Besides basic operations of insertion and query, update and deletion are required to handle changing network states (e.g., TCP flow states) and flow handling rules (e.g., in SDN).

1.2 Prior Arts and Limitations

Existing multi-set membership query algorithms can not satisfy above two requirements. As a straightforward solution for the multi-set query, hash table can be used. With the elements to query as the hash keys and the set IDs to return as the hash values, we can build a hash table for these key-value pairs. Hash-table-based solutions are accurate but not memory efficient. Take per-flow state monitoring as an example. With the 5-tuple flow key <src IP (32bit), dest IP (32bit), src port (16bit), dest port (16bit), protocol (8bit)> used as the hash key and the state value as the hash value, we can build a hash table. It is accurate with high speed, however, they need to store element keys to reduce hash collision rate. As a 5-tuple flow key has 104 bits, the memory space is proportional to the actual number of entries and a large memory is needed [22].

Bloom filter (**BF**) is a space-efficient probabilistic data structure that supports membership testing. With low false positive rate and $O(k)$ time complexity [13–15, 18, 23], BF-based approaches are used in many network applications to reduce the information processing and memory/bandwidth consumption. However, designed for managing the set membership to determine if a queried element exists in a set, conventional BFs are generally difficult to apply for multi-set query.

Recently, a few efforts [8, 11, 12, 24] have sought to design multi-set query algorithms based on BF to well utilize the good properties of BF. However, they suffer as least one of the following drawbacks and can not satisfy the requirements of many network functions: (1) low query processing speed due to large memory access; (2) relative high error rate because of hashing collisions; (3) limitation in only supporting insertion and query operations; and (4) limitation in the set size to support.

1.3 Proposed Approach

To address the issues with existing multi-set query algorithms and satisfy the requirements of the network functions simultaneously, we design a novel **B_h sequence-based Bloom filter** (**B_hBF**) for multi-set queries. A B_h sequence is a set of integers with the property that for any $h^{'} \leq h$, all the sums of $h^{'}$ integers from the B_h sequence are distinct. Therefore, given a sum of $h^{'}$ integers, we can determine which $h^{'}$ integers add up to the sum.

In B_hBF, a set ID is encoded as a code in a B_h sequence. B_hBF utilizes a cell array instead of a bit array to store the set IDs encoded by the B_h sequence. Exploiting good properties of B_h sequences, we can correctly decode the cell to obtain the set IDs inserted previously even when the number of hash collisions is high, which brings high query accuracy. To the best of our knowledge, this is the first time that B_h sequences are used in BF design to support multi-set queries. The technique contributions in B_hBF are summarized as follows:

- We design B_hBF, a compact probabilistic data structure to support multi-set membership query with four basic operations: insertion, query, deletion, and update.
- We propose two strategies to speed up the query speed and increase the query accuracy.
- We present theoretical analysis on B_hBF for its false positive rate and classification failure.
- We have conducted extensive experiments by using two real datasets, and the results demonstrate that our B_hBF outperforms the state-of-the-art algorithms for multi-set test with the capability of supporting all four operations, and it can achieve a high correctness rate with low space cost and low memory access overhead.

Paper organization. The rest of the article is organized as follows. We introduce the related work in Section 2. In Sections 3 and 4, we give the preliminaries and an overview of our scheme and then present its details. In Section 5, we provide the mathematical expressions of two types of errors and conduct evaluation in Section 6. Finally, we conclude the work in Section 7.

2 RELATED WORK

2.1 Standard BF

A standard BF [2] is a space-efficient data structure for representing a single set S in order to support membership testing. When given a queried element $\mathbf{e}$, it is used to answer whether it belongs to the set. A standard BF constructs an array of m bits, initially all bits are set to 0. In the phase of insertion, it uses k independent hash functions $h_1(\cdot), \ldots, h_k(\cdot)$ to calculate the hashing positions of the array $h_1(\mathbf{e})\%m, \ldots, h_k(\mathbf{e})\%m$ and set them to 1. In the phase of query, we check whether all $h_i(\mathbf{e})\%m$ for $1 \leq \mathrm{i} \leq \mathrm{k}$ are set to 1s. If yes, $\mathbf{e}$ is considered to be a member of the set S; otherwise, it is not.

2.2 Multiple BFs-Based Structure

Using multiple BFs is a straightforward solution for multi-set membership query [6, 7, 10, 29]. Each BF vector corresponds to a set, and a query needs to be performed against all BF vectors. Following [6, 10, 29], variants [12, 28] have been proposed. Hao et al. put forward **combinatorial Bloom filters** (**COMB**) [12], multiple groups of hash functions are used to differentiate set IDs in one BF. Yoon et al.[28] design a Bloom Tree, where each node is a BF vector and each leaf node represents a set ID. To improve the search efficiency of the schemes using multiple BFs, Crainiceanu et al. [7] propose FlatBF, which packs every 64 BFs of length m into m 64-bit integers so that it can exploit parallel processing at the bit-level.

All these algorithms require performing multiple groups of hash functions to find the set ID of a queried element. Depending on the groups of hash functions, their memory accesses are generally several times those of standard BFs that only involve one group of hash functions, which results in much lower query processing speed.

In contrast, our proposed B_hBF only needs one group of hash functions, and has low memory access cost thus high speed.

2.3 Modified Structure Based on BF

To support multi-set query operations, studies [11, 24, 25] propose to modify the conventional BF to use cells instead of bits to hold the set IDs directly.

Goodrich et al. [11] proposed **invertible Bloom filter** (**IBF**), which directly adds the value of set ID to the corresponding k cells and records the hash check value of the element. Obviously, in the query phase, if there is one cell that records the information of a single element and the hash check value is correct, the value of the set ID can be returned correctly. Otherwise *"not found"* is returned, which means if hashing collisions occur more than once in a cell, the cell cannot be decoded.

Xiong et al. [25] proposed a **key-value Bloom filter** (**KBF**), which assigns a unique string to each set ID with any two different strings having different XOR results. Obviously, when hash conflicts occur more than twice in a cell, the cell cannot be decoded. Since the XOR result of two same strings is 0, it is impossible to decode even the number of hashing collisions does not exceed twice.

Besides cell-based BF solutions [11, 25], studies [8, 17, 26] use bit vectors like standard BF to store the information of set IDs of elements. Dai et al. [8] proposed a **noisy Bloom filter** (**NBF**), which encodes the set IDs of elements into a bit vector in a noisy way. The OMASS scheme [17] uses block BF to track the set IDs of elements. Obviously, a disadvantage inherited from standard BF, NBF, and OMASS do not support the operation of deletion and update. Since a shifting framework [26] can handle update operations by increasing bits for counting, the probability that it gives an uncertain answer increases when the number of sets increases.

There are also some schemes [19, 24] that combine two or more data structures including variants of BF to test multi-set membership. Yang et al. [24] proposed coloring embedder consisting of a variant of BF and an adjacency list. However, the operations of insertion and update are expensive, since the color problem is well known to be NP-hard, which limits the number of set IDs to support thus its applications in the network field. Qiao et al. [19] proposed iSet, which consists of two data structures including an index filter and a set-ID table. The index filter is a BF, while the set-ID table is a multi-hash table where the set ID is stored in one of the candidate table entries calculated by the multiple hash functions. However, if all candidate entries have already stored the set IDs of other elements inserted, an insertion-failure occurs and the non-negligible probability of failing effects the query accuracy.

To solve the problem faced in current studies, we propose a novel B_hBF, which supports four operations, including insertion, query, deletion, and update. It uses a cell array to store the set IDs

encoded with the numbers from a B_h sequence. Exploiting good properties of B_h sequences, the cell can be correctly decoded even when the number of hash collisions is high, which brings high query accuracy. We further propose two strategies to speed up the query speed and increase the query accuracy.

3 PRELIMINARIES AND SOLUTION OVERVIEW

The goal of this work is to novelly exploit BF for the quick query of multi-set membership to enable fast network functions. In a multi-set membership query, we need to determine which set among multiple sets a queried element belongs to. If the queried element does not belong to any set, NULL will be returned.

More formally, we divide elements $\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$ into L mutually independent sets $S_1, S_2, \ldots, S_L$ with $S_i \cap S_j = \emptyset$ where $1 \leq i < j \leq L$. The set ID of S_i is denoted by $\mathbf{v_i}$. If a queried element $\mathbf{e} \in S_i$, the multi-set membership query returns the set ID of S_i, otherwise $\mathbf{e} \notin \{S_1 \cup S_2 \ldots \cup S_L\}$ and returns NULL. An element $\mathbf{e_i}$ can be a string, an integer, or an IP address.

In order to support multi-set queries, we propose to use a B_h sequence to encode the set IDs, taking advantage of the mathematical properties of the B_h sequence. We design a novel B_hBF to store the element-set pairs. In this section, we first introduce the definition of B_h sequences and its properties we use, then provide an overview of our algorithm.

3.1 B_h Sequences

B_h sequences were first studied by Sidon [21] in connection with the theory of Fourier series, and a B_2 sequence was also called Sidon sequence. The construction of B_h sequences was provided in [4, 9].

Definition 1 (B_h Sequences). A sequence of $\{d_1, d_2, \ldots, d_L\}$ is called a B_h sequence if all

$$d_{i_1} + d_{i_2} + \cdots + d_{i_h}, 1 \leq i_1 \leq i_2 \leq \ldots \leq i_h \leq L,$$

are distinct.

As the sum of any h integers is unique in a B_h sequence, if the sum number is given, we can determine which h integers add up to the total.

Example 1. The set of integers $\{1, 22, 55, 72\}$ is a B_3 sequence because the sum of any three integers (repetitions allowed) is different from any other three integers. It has 20 combinations: $1+1+1=3$, $1+1+22=24$, $1+1+55=57$, $1+1+72=74$, $1+22+22=45$, $1+22+55=78$, $1+22+72=95$, $1+55+55=111$, $1+55+72=128$, $1+72+72=145$, $22+22+22=66$, $22+22+55=99$, $22+22+72=116$, $22+55+55=132$, $22+55+72=149$, $22+72+72=166$, $55+55+55=165$, $55+55+72=182$, $55+72+72=199$, $72+72+72=216$. When given a sum number 24, we have $24 = 1+1+22$. That is, the three integers 1, 1, and 22, can be separated from the sum 24.

Theorem 1. *If a sequence* S *is a B_h sequence,* S *is also a $B_{h'}$ sequence where $h' \in [1, h]$.*

Proof. We prove the theorem by the way of contradiction. When $h' = h$, S is a B_h sequence according to Definition 1. When $h' \in [1, h-1]$, if S is not a $B_{h'}$ sequence, we can find at least two groups of h' integers in S with the same sum. If we add the same $h - h'$ integers to the two groups, the two groups should still get the same sum. This means that we have two groups of h integers to obtain the same sum, which is inconsistent with the Definition 1. So S is a $B_{h'}$ sequence. □

Example 2. For the same set $S = \{1, 22, 55, 72\}$ in Example 1 which is a B_3 sequence, we obtain that the sum of any two integers in S is different from the sum of any other two integers: $1+1=2$,

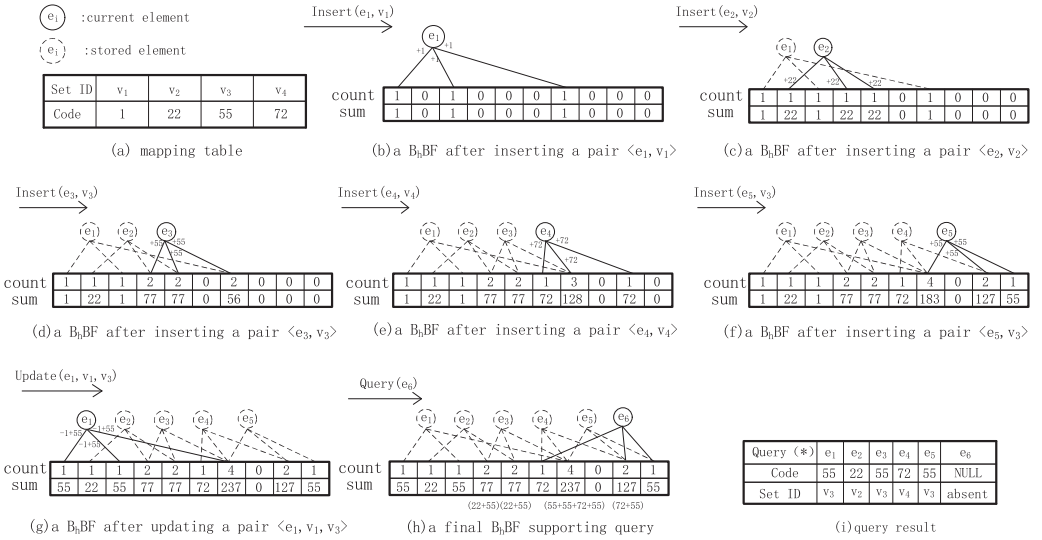


Fig. 1. Example of four operations.

$1 + 22 = 23$, $1 + 55 = 56$, $1 + 72 = 73$, $22 + 22 = 44$, $22 + 55 = 77$, $22 + 72 = 94$, $55 + 55 = 110$, $55 + 72 = 127$, $72 + 72 = 144$. So the set $\mathbf{S}$ is a B_2 sequence and also is a B_1 sequence obviously.

Representing each element of a set by a group of B_h codes, VI-CBF [20] (a variant of counting BF) exploits the good properties of B_h sequences to reduce the false positive rate for membership query, which determines whether an element is in a set. In contrast, B_hBF is designed for multi-set query, which determines which set an element belongs to.

3.2 Basic Idea of B_hBF

Instead of using a bit array to represent the traditional BF, B_hBF uses a cell array to store element-set pairs. As shown in Figure 1(b), B_hBF consists of m cells $C[i]$ for $0 \le i \le m-1$, each of which contains two fields: a count field and a sum field. A count field denoted as $C[i].count$ represents the number of element-set pairs stored, and a sum field denoted as $C[i].sum$ is the sum of the $C[i].count$ codes of the set IDs. Initially, we set $C[i].count = 0$ and $C[i].sum = 0$ for all cells with $0 \le i \le m-1$.

In the current cell-based BF [11, 25], high collisions result in a high query error rate, as the set IDs inserted into the cell cannot be correctly identified when the number of hash collisions is more than one [11] or two [25]. To solve the problem, B_hBF utilizes a B_h sequence to encode the set IDs. When an element $\mathbf{e_i}$ with its set ID $\mathbf{v_i}$ needs to be stored, a code $d_j \in \mathbf{S}$ is assigned to $\mathbf{v_i}$ where $\mathbf{S} = \{d_1, d_2, \ldots, d_L\}$ is a B_h sequence. The relation between $\mathbf{v_i}$ and d_j is one-to-one mapping, as shown in Figure 1(a).

After hashing $\mathbf{e}$ to k cells, its encoded set ID d_j will be inserted to the k cells by adding the cell's sum value with the newly added set ID code d_j. As the codes of set IDs are from a B_h sequence, if the count value of a cell $C[i].count$ does not exceed h, we can determine which $C[i].count$ codes add up to $C[i].sum$. Therefore, given an element to query, if k cells are confirmed to have a common code d_j, it is the code corresponding to the set ID of the queried element.

As only one group of hash functions is applied in B_hBF, the memory access cost is the same as that of the traditional BF, but is much lower compared with the one using multiple BFs [10, 12, 28].

ALGORITHM 1: Insert Operation

Input: $\langle \mathbf{e}, \mathbf{v} \rangle$, $\mathbf{e}$ is an element with its set ID $\mathbf{v}$

```
1 d ← Encode(v);
2 for 1 ≤ i ≤ k do
3     C[h_i(e)%m].count + 1;
4     C[h_i(e)%m].sum + d;
```

4 DETAILED SOLUTION

Exploiting a B_h sequence to encode the set ID, B_hBF supports following four operations:

- Insert($\mathbf{e_i}, \mathbf{v_i}$): insert an element-set pair $\langle \mathbf{e_i}, \mathbf{v_i} \rangle$ into a B_hBF.
- Query($\mathbf{e_i}$): query the set ID $\mathbf{v_i}$ for an element $\mathbf{e_i}$ in a B_hBF;
- Delete($\mathbf{e_i}, \mathbf{v_i}$): delete an element $\mathbf{e_i}$ and its associated set ID $\mathbf{v_i}$ from a B_hBF;
- Update ($\mathbf{e_i}, \mathbf{v_{old}}, \mathbf{v_{new}}$): update the set ID of an element $\mathbf{e_i}$ from old set ID $\mathbf{v_{old}}$ to new set ID $\mathbf{v_{new}}$ in a B_hBF.

Following, we introduce these operations in details.

4.1 Insert

In Algorithm 1, inserting a pair $\langle \mathbf{e}, \mathbf{v} \rangle$ into a B_hBF takes two steps. In the first step, the element $\mathbf{e}$'s value $\mathbf{v}$ is mapped to a code d in a B_h sequence S with an **Encode** process. In the second step, perform $C[h_i(\mathbf{e})\%m].count + 1$ and $C[h_i(\mathbf{e})\%m].sum + d$ for $1 \le i \le k$.

Example 3. In Figure 1(b)–(f), five element-set pairs are sequentially inserted into a B_hBF: $\langle \mathbf{e_1}, \mathbf{v_1} \rangle$, $\langle \mathbf{e_2}, \mathbf{v_2} \rangle$, $\langle \mathbf{e_3}, \mathbf{v_3} \rangle$, $\langle \mathbf{e_4}, \mathbf{v_4} \rangle$, $\langle \mathbf{e_5}, \mathbf{v_3} \rangle$. The B_hBF is initialized to 0 for each cell. When $\langle \mathbf{e_1}, \mathbf{v_1} \rangle$ arrives, the set ID $\mathbf{v_1}$ is encoded to 1 according to the mapping table in Figure 1(a). For the cells $C[0]$, $C[2]$, and $C[6]$ in which the element $\mathbf{e_1}$ is hashed to, perform $C[0].count + 1$ and $C[0].sum + 1$, $C[2].count + 1$ and $C[2].sum + 1$, and $C[6].count + 1$, and $C[6].sum + 1$. After inserting $\langle \mathbf{e_1}, \mathbf{v_1} \rangle$, the B_hBF is shown in Figure 1(b). Similar operations are performed when following element-set pairs come. After five pair insertions are completed, B_hBF is shown in Figure 1(f).

4.2 Query

As the inserted set ID is encoded as an integer in a B_h sequence, we can separate the inserted codes from $C[i].sum$ when $C[i].count$ is less than or equal to h. We call the process to separate the codes from a sum as **Decode**.

In a B_h sequence, the sum of any h integers is unique, thus the sum and the group of codes added up to the sum have a one-to-one mapping. Using the mapping, it can easily decode a sum to obtain the codes inserted, which helps B_hBF to achieve a high query accuracy even when the number of hash collisions is high. The decoding can be performed with a low cost of $O(1)$ by storing the one-to-one mapping with a hash table.

Therefore, to query an element $\mathbf{e}$'s set ID, we can first decode all its k cells $C[h_1(\mathbf{e})\%m]$, $C[h_2(\mathbf{e})\%m], \ldots, C[h_k(\mathbf{e})\%m]$. Then $\mathbf{e}$'s set ID is the one that corresponds to the common code of k cells.

Example 4. Query element $\mathbf{e_2}$'s set ID. In Figure 1(h), the element $\mathbf{e_2}$ is hashed to $C[1], C[3]$, and $C[4]$. After decoding these cells, we have code lists $\{22\}$, $\{22, 55\}$, and $\{22, 55\}$, which are inserted into $C[1]$, $C[3]$, and $C[4]$, respectively. As the common code among these lists is 22, and it corresponds to $\mathbf{v_2}$ according to the mapping table in Figure 1(a), the set ID of $\mathbf{e_2}$ is $\mathbf{v_2}$.

ALGORITHM 2: Query Operation

Input: element **e**
Output: NULL, a set ID **v**, or UNKNOWN
1 sort $C[h_i(\mathbf{e})\%m]$ for $1 \le i \le k$ according to the *count* field in an ascending order, the resulted ordered cells are $C_1, C_2, \ldots, C_k$;
2 **for** $1 \le i \le k$ **do**
3 **if** $C_i.count == 0$ **then**
4 return NULL;
5 **else if** $0 < C_i.count \le h$ **then**
6 $C_i.CodeList \leftarrow$ Decode($C_i.sum$);
7 **if** $i == 1$ **then**
8 $ComCode \leftarrow C_i.CodeList$;
9 **else**
10 $ComCode \leftarrow$ Intersection ($C_i.CodeList, ComCode$);
11 **if** *size* ($ComCode$) == 0 **then**
12 return NULL;
13 **if** *size* ($ComCode$) == 1 **then**
14 return set ID **v** that maps the code $\in ComCode$;
15 **else**
16 return UNKNOWN;

The above straightforward solution, however, may suffer from the problems of **redundant computation** and **unknown query result**. To address the issues, we further propose two strategies in the following sub-sections.

4.2.1 Avoiding Redundant Computation. In Algorithm 2, $C_i.CodeList$ represents the code list of cell C_i, and $ComCode$ denotes the common code list of multiple cells. If any $C_i.CodeList$ has no common code with the intersection of other code lists, we know that the queried element has not been inserted into a B_hBF. Enlightened by this observation, instead of decoding all cells and calculating the interaction among all code lists, we propose Strategy 1 to speed up the query procedure by avoiding redundant decoding and calculating the list intersection. The detailed Strategy 1 is:

(1) Sort the cells in the ascending order according to their count fields (on line 1 in Algorithm 2).

(2) Decode the cells and calculate the list intersection operations one by one. If any cell's $CodeList$ has no common code with $ComCode$ calculated before, stop the process and return NULL, that means the queried element is not inserted into the B_hBF. The detailed operations can be found on lines 2–12 in Algorithm 2.

In our design, we use sorting as the first step to facilitate the quickly finding of NULL intersection and stop the query process in advance. This is because that the interaction of code lists of small count cells has high probability of being NULL.

Example 5. Query element $\mathbf{e_6}$'s set ID with Strategy 1. In Figure 1(h), element $\mathbf{e_6}$ is hashed to $C[5]$, $C[8]$, and $C[9]$. Two query processes are performed. (a) **Without sorting**. If we first decode $C[5]$ and $C[8]$, as the list intersection of these two cells $C[5]$ and $C[8]$ is not NULL, we need to further decode the third cell $C[9]$ and calculate the list intersection again. (b) **With sorting.** Sorting these three cells in the ascending order according to their count values, we have $C[5].count \le C[9].count \le C[8].count$. After decoding the first two cells $C[5]$ and $C[9]$, we find

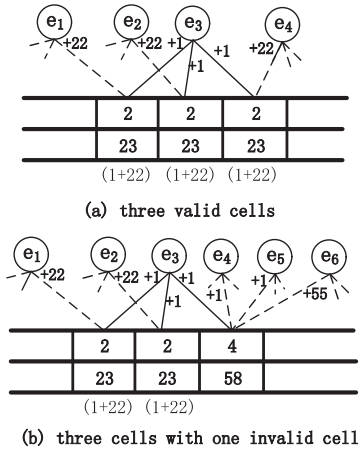


Fig. 2. Two cases of query algorithm returning UNKNOWN.

that the intersection of the code lists of $C[5]$ and $C[9]$ is NULL. Without further decoding the third cell $C[8]$ and calculating the intersection again on the code list of $C[8]$, we can confirm that the element $\mathbf{e_6}$ is not inserted into B_hBF yet and quickly return NULL. The computation cost under (b) is largely reduced.

4.2.2 Alleviating the Problem of Unknown Query Result. According to the size of common code list, the query algorithm returns three types of results, first two are certainty results, the third is an uncertain result. (a) When $size(ComCode)$=0, return NULL, which means that the queried element was not inserted into B_hBF. (b) When $size(ComCode)$=1, return the set ID that corresponds to the unique code in $ComCode$ according to the mapping table. (c) When $size(ComCode) \geq 2$, as we do not know among the list of $ComCode$, which one the queried element belongs to, return an uncertain result UNKNOWN.

If we can provide a solution to avoid the UNKNOWN result, we can increase the query accuracy. Following, we first present how UNKNOWN happens, then provide our Strategy 2 to relieve the problem.

In B_hBF, according to a cell's count value, we divide the cells into two categories

$$C[i] = \begin{cases} valid & C[i].count \leq h \\ invalid & otherwise. \end{cases} \tag{1}$$

According to the definition of B_h sequences, we can decode a valid cell to obtain all the codes inserted to it, while can not decode an invalid cell to get the codes of set IDs in it.

If an element-set pair $\langle \mathbf{e_i}, \mathbf{v_i} \rangle$ is inserted into a B_hBF, Figure 2 presents two cases under which UNKNOWN happens. In the case of Figure 2 (a), the set ID code of the element $\mathbf{e_3}$ is 1. However, the set ID code of $\mathbf{e_1}$, $\mathbf{e_2}$, and $\mathbf{e_4}$ is 22. Hashing $\mathbf{e_1}$, $\mathbf{e_2}$, and $\mathbf{e_4}$ will make $\mathbf{e_3}$'s cells being inserted with a common code 22. As a result, the sum values of these cells are all $23 = 1 + 22$. As the $ComCode$={1, 22}, UNKNOWN is returned. In this case, there exists one cell corresponding to the queried element is polluted by other elements that hold the same set ID. We can not avoid the problem due to the random feature of hash functions. In Section 5.2, we provide theoretical analysis on the probability, and moreover, the experiment results also show that this case seldom happens.

In the case of Figure 2(b), $h = 3$ in the example. Among the three cells of $\mathbf{e_3}$, one cell is invalid because the cell count is 4, which is larger than $h = 3$ and equals to $h + 1$, we can not separate the codes that are added up to the sum (58). However, the opportunity for B_hBF to return a certainty

result increases when more cells are decoded. If we can provide a strategy to well utilize the invalid cell, we can alleviate the UNKNOWN problem in this case.

As $23 = 1 + 22$, 1 and 22 are the candidate codes for $\mathbf{e_3}$. If 1 is the set code of $\mathbf{e_3}$, 1 should also be included in other cells. Therefore, we use the sum 58 of the invalid cell to subtract the candidate code 1 and have $58 - 1 = 57$. Obviously, the result 57 is a sum of three codes of the B_3 sequence in Example 1. It indicates that candidate code 1 may be inserted to the cell and included in sum 57 with count = 4. Similarly, we can also test the candidate code 22. As $57 - 22 = 25$ is not a sum of three codes of the B_3 sequence, 22 is definitely not the set code of $\mathbf{e_3}$.

Therefore, to reduce the number of UNKNOWN results in Case 2, we propose Strategy 2 with the following steps:

(1) Decode and calculate the intersection of all valid cells. The codes in the intersection are the candidate ones of the queried set ID code.

(2) Using the sum of an invalid cell whose count is $h+1$ to subtract a candidate code, if the result is a sum of codes in the B_h sequence, the candidate code may have been inserted into the cell and included in the sum.

(3) If all invalid cells whose count are $h + 1$ pass the similar testing in (2), we can conclude that this candidate code is the set ID code of the queried element.

Although the above procedures help alleviate the UNKNOWN problem, there is a small chance that the return is error. As the code we find is in a B_h sequence not in a B_{h+1} sequence, we cannot guarantee that the sum of $h + 1$ codes is distinct. Nevertheless, our experiment results demonstrate that our Strategy 2 is a very effective in relieving the UNKNOWN problem, and helps our B_hBF to achieve high query accuracy.

4.3 Delete

Deleting an element-set pair $\langle \mathbf{e_i}, \mathbf{v_i} \rangle$ can be easily realized in B_hBF. Let d be the code of $\mathbf{v_i}$. After finding all k cells $C[h_i(\mathbf{e})\%m]$ for $i = 1, 2, \ldots, k$ of the element $\mathbf{e_i}$, $C[h_i(\mathbf{e})\%m].count - 1$ and $C[h_i(\mathbf{e})\%m].sum - d$ are performed.

4.4 Update

In the update process, for the element $\mathbf{e}$ with an old set ID $\mathbf{v_{old}}$ and a new set ID $\mathbf{v_{new}}$, it performs $C[h_i(\mathbf{e})\%m].sum\text{-}d_{old}\text{+}d_{new}$ for $i = 1, 2, .., k$ where d_{old} is the code of $\mathbf{v_{old}}$ and d_{new} is the code of $\mathbf{v_{new}}$. Now the element has been updated with the new set ID in B_hBF.

Example 6. In Figure 1(g), element $\mathbf{e_1}$ is hashed to $C[0], C[2]$ and $C[6]$. Updating element $\mathbf{e_1}$'s set ID from $\mathbf{v_1}$ to $\mathbf{v_3}$, it performs $C[0].sum - 1 + 55$, $C[2].sum - 1 + 55$, and $C[6].sum - 1 + 55$ where 1 and 55 are the codes of $\mathbf{v_1}$ and $\mathbf{v_3}$, respectively.

5 ANALYSIS

Two types of errors can occur, which are false positive and classification failure. We present the mathematical expressions of these two types of error in this section.

5.1 False Positive Rate

False positive: As a probabilistic data structure, the B_hBF query may yield a false positive, that is, it falsely identifies that an element-set pair is present and has been inserted into B_hBF, while it is not.

As shown in Figure 3, suppose $\mathbf{e_2}$ is absent, but $\mathbf{e_1}$ and $\mathbf{e_3}$ are present and have already inserted their set ID codes into the corresponding cells. The query result indicates that $\mathbf{e_2}$'s set ID code is 14. Thus, a false positive occurs.

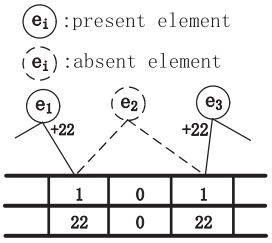


Fig. 3. False positive: $\mathbf{e_2}$ is absent but the query returns 22.

In our B_hBF, given an absent element $\mathbf{e_i}$, a false positive happens when the element's all valid cells have only one common set ID code. Before we provide false positive rate expression in Theorem 2, we first provide five lemmas.

To find the false positive rate, we need to calculate the probability that two valid cells have one or more common set ID codes. However, this is challenging as it involves a mathematically and computationally intractable combinatorics calculation.

We take an example to explain the problem. Given two cells $C[i]$ and $C[j]$, the cell $C[i]$ is inserted $r_1 = 3$ codes (repetitions allowed) and the other $C[j]$ is inserted $r_2 = 1$ code, and these codes come from a group with $L = 3$ codes. To find the probability that two cells have a common code, the following three situations should be considered: (1) 3 codes in $C[i]$ are the same, and its probability is $\frac{1}{9}$, so the probability that two cells have a common code is $\frac{1}{9} \times \frac{1}{3} = \frac{1}{27}$; (2) 2 of the 3 codes in $C[i]$ are the same and the other one is different, and its probability is $\frac{2}{3}$, so the probability that two cell have a common code is $\frac{2}{3} \times \frac{2}{3} = \frac{4}{9}$; (3) 3 codes in $C[i]$ are different, and its probability is $\frac{2}{9}$, so the probability that two cells have a common code is $\frac{2}{9} \times \frac{3}{3} = \frac{2}{9}$. The total probability that two cell have a common code is $\frac{1}{27} + \frac{4}{9} + \frac{2}{9} = \frac{19}{27}$.

For a general problem, assuming that k valid cells have inserted $r_1, r_2, \ldots, r_k$ codes, respectively. In order to calculate the probability that k cells have one or more common codes, we need to enumerate all possible combinations, and the number of combinations is $r_1 \times r_2 \times \cdots \times r_k$.

From Lemma 1, we will find that the probability of $C[i].count$ codes are different is the overwhelming in B_hBF when $L \gg h$. Based on which, we provide the following Lemma 2 to calculate the probability that two valid cells have only one or more common set ID codes.

Lemma 1. *Given a set of L codes belonging to a B_h sequence and a valid cell inserted with r codes where $L \gg h$ and $h \geq r$, the probability that r codes are different is*

$$\frac{L}{L} \times \frac{L-1}{L} \times \cdots \times \frac{L-r+1}{L} = \frac{\prod_{i=0}^{r-1} L-i}{L^r}. \tag{2}$$

Proof. A code is randomly selected from L codes and repeated r times to obtain r codes. If r codes are different, the process is: for the first time no code has been selected before, so selecting any code from the Lcodes satisfies the requirement. As a result, to select a code from L codes while guaranteeing that the selected codes are different, the probability that is 1. The second time $L-1$ codes can be selected, and to make the selected code different from the previous one, the probability of selection is $\frac{L-1}{L}$. This process continues, until r times $L-r+1$ codes can be selected and the probability is $\frac{L-r+1}{L}$. So the probability that r codes are different is $1 \times \frac{L-1}{L} \times \cdots \times \frac{L-r+1}{L} = \frac{\prod_{i=0}^{r-1} L-i}{L^r}$. □

We observe that in the network field, the number of codes in a set, L, is relatively large in many real scenario. That is, $L \gg h$ (*e.g.*, $L = 100h$), and the probability that $C[i].count$ exceeds 16 is

minuscule according to [10]. According to Lemma 1, we find that if $L \gg h$ (e.g., $L = 100h$), the probability is 99% and 93% when $h = r = 3$ and $h = r = 16$, respectively. That is, the probability that $C[i].count$ codes are different is overwhelmingly high. Reasonably, we focus on the case where different codes are stored in a cell.

Lemma 2. *Given two valid cells $C[i]$ and $C[j]$, and a set of L codes belonging to a B_h sequence where $L \gg h$, $C[i]$ is inserted with r_1 codes, $C[j]$ is inserted with r_2 codes, where $1 \le r_1 \le h$ and $1 \le r_2 \le h$, respectively. Then the probability of these two cells having c common codes, denoted as $F(r_1, r_2, c)$, is*

$$F(r_1, r_2, c) = \begin{cases} 0 & c > \min\{r_1, r_2\} \\ \prod_{l_1=0}^{r_1+r_2-1} \frac{L-l_1}{L} & c = 0 \\ \prod_{l_1=0}^{r_1-1} \frac{L-l_1}{L} \times \frac{r_1}{L} & r_2 = 1, c = 1 \\ \binom{r_2}{c} \times \rho_1\rho_2\rho_3 & 1 \le c < 2 \le r_2, \end{cases} \tag{3}$$

where $\rho_1 = \prod_{l_1=0}^{r_1-1} \frac{L-l_1}{L}$, $\rho_2 = \prod_{l_2=0}^{c-1} \frac{r_1-l_2}{L}$, and $\rho_3 = \prod_{l_3=0}^{r_2-c-1} \frac{L-r_1-l_3}{L}$.

Proof. There are four cases when calculating the probability that there exists c common codes between two valid cells.

Case 1 ($c > \min\{r_1, r_2\}$): The c common codes should exist in both r_1 codes in $C[i]$ and r_2 codes in $C[j]$, so the value c must not be greater than the minimum value of r_1 and r_1.

Case 2 ($c = 0$): The probability that $C[i]$ has r_1 different codes is $\rho_1 = \frac{L}{L} \times \frac{L-1}{L} \times \cdots \times \frac{L-r_1+1}{L}$. If $c = 0$, it means that there is no common code both in $C[i]$ and $C[j]$, the r_2 codes come from the remaining $L - r_1$ codes and the probability that $C[j]$ has r_2 different codes is $\beta_1 = \frac{L-r_1}{L} \times \frac{L-r_1-1}{L} \times \cdots \times \frac{L-r_1-r_2+1}{L}$. So we have $\rho_1 \times \beta_1 = \prod_{l_1=0}^{r_1+r_2-1} \frac{L-l_1}{L}$.

Case 3 ($r_2 = 1$, $c = 1$): The probability that $C[i]$ has r_1 different codes is ρ_1. If $c = 1$, it means that the only one code in $C[j]$ should come from r_1 codes in $C[i]$ and its probability is $\frac{r_1}{L}$. So we have $\rho_1 \times \frac{r_1}{L} = \prod_{l_1=0}^{r_1-1} \frac{L-l_1}{L} \times \frac{r_1}{L}$.

Case 4 ($1 \le c < 2 \le r_2 \le h$): To make these two cells have c common codes, it should satisfy that c codes in $C[j]$ also exist in $C[i]$, while $r_2 - c$ codes in $C[j]$ do not. The probability that $C[i]$ has r_1 different codes is ρ_1, the probability that c codes in $C[j]$ also exist in $C[i]$ is $\rho_2 = \frac{r_1}{L} \times \frac{r_1-1}{L} \times \cdots \times \frac{r_1-c+1}{L}$, and the probability that $r_2 - c$ in $C[j]$ do not exist in $C[i]$ is $\rho_3 = \frac{L-r_1}{L} \times \frac{L-r_1-1}{L} \times \cdots \times \frac{L-r_1-r_2+c+1}{L}$, and the combination number of r_2 positions selected from c common codes is $\binom{r_2}{c}$. So we have $\binom{r_2}{c} \times \rho_1\rho_2\rho_3$. □

For example, given $L = 300, h = 3, r_1 = 2, r_2 = 3$, we have $F(2, 3, 1) = \binom{3}{1} \times \frac{300\times299}{300^2} \times \frac{2}{300} \times \frac{298\times297}{300^2} = 1.9\%$ and $F(2, 3, 2) = \binom{3}{2} \times \frac{300\times299}{300^2} \times \frac{2\times1}{300^2} \times \frac{298}{300} = 0.007\%$.

In order to calculate the false positive rate, we need to consider the probability that the valid cells have stored a common code among k cells. However, it is also an intractable combinatorics problem. For example, given $h = 3, k = 5$, and the count filed of these 5 cells are all 3. Among 5 valid cells, the first two cells may have one, two, or three common codes, or the first two cells may have three common codes and combine the third cell to get three, two or one common codes, etc. To solve the problem of intractability, we observed that the probability $F(r_1, r_2, c_a) \ll F(r_1, r_2, c_b)$ when $L \gg h, c_a > c_b \ge 1$. For example, we have $F(2, 3, 2) \ll F(2, 3, 1)$. Now we give the following Lemma.

Lemma 3. *Given two valid cells $C[i]$ and $C[j]$, and a set of L codes belonging to a B_h sequence where $L \gg h$, $C[i]$ is inserted with r_1 codes, $C[j]$ is inserted with r_2 codes, where $1 \le r_1 \le h$ and*

$1 \le r_2 \le h$, *respectively. The probability of these two cells having c common code is denoted as* $F(r_1, r_2, c)$. *If* $c_a > c_b \ge 1$, *we have* $F(r_1, r_2, c_a) \ll F(r_1, r_2, c_b)$.

Proof. According to the Equation (3), we have $F(r_1, r_2, c_a) = \binom{r_2}{c_a} \times \rho_{1a}\rho_{2a}\rho_{3a}$, where $\rho_{1a} = \prod_{l_1=0}^{r_1-1} \frac{L-l_1}{L}$, $\rho_{2a} = \prod_{l_2=0}^{c_a-1} \frac{r_1-l_2}{L}$, and $\rho_{3a} = \prod_{l_3=0}^{r_2-c_a-1} \frac{L-r_1-l_3}{L}$, and $F(r_1, r_2, c_b) = \binom{r_2}{c_b} \times \rho_{1b}\rho_{2b}\rho_{3b}$, where $\rho_{1b} = \prod_{l_1=0}^{r_1-1} \frac{L-l_1}{L}$, $\rho_{2b} = \prod_{l_2=0}^{c_b-1} \frac{r_1-l_2}{L}$, and $\rho_{3b} = \prod_{l_3=0}^{r_2-c_b-1} \frac{L-r_1-l_3}{L}$.

(1) Obviously, $\rho_{1a}=\rho_{1b}$.

(2) Since $c_a+1 > c_b+1$, we have $\rho_{2a} = \frac{r_1}{L} \times \frac{r_1-1}{L} \times \cdots \times \frac{r_1-c_b+1}{L} \times \cdots \times \frac{r_1-c_a+1}{L} = \rho_{2b} \times \prod_{l_2=c_b}^{c_a-1} \frac{r_1-l_2}{L}$.

(3) Since $r_2-1-c_b > r_2-1-c_a$, we have $\rho_{3b} = \frac{L-r_1}{L} \times \frac{L-r_1-1}{L} \times \cdots \times \frac{L-r_1-r_2+c_a+1}{L} \times \cdots \times \frac{L-r_1-r_2+c_b+1}{L} = \rho_{3a} \times \prod_{l_3=r_2-c_a}^{r_2-c_b-1} \frac{L-r_1-l_3}{L}$. So we have $\frac{F(i,j,c_a)}{\rho_{1a}\rho_{2b}\rho_{3a}} = \binom{r_2}{c_a} \prod_{l_2=c_b}^{c_a-1} \frac{r_1-l_2}{L}$, $\frac{F(i,j,c_b)}{\rho_{1a}\rho_{2b}\rho_{3a}} = \binom{r_2}{c_b} \prod_{l_3=r_2-c_a}^{r_2-c_b-1} \frac{L-r_1-l_3}{L}$, and $\frac{F(i,j,c_a)}{F(i,j,c_b)} < \frac{\binom{r_2}{c_a}}{\binom{r_2}{c_b}} \times (\frac{r_1-c_b}{L-r_1-r_2+c_b+1})^{c_a-c_b}$. Since $L \gg h$, $h \ge r_1$, $h \ge r_2$ and $h \ge c_a \ge \min\{r_1, r_2\}$, we have $r_1 - c_b < h$ and $F(i, j, c_a) \ll F(i, j, c_b)$, $(F(i, j, c_b) \approx (\frac{L}{h})^{c_a-c_b} F(i, j, c_a)$. □

In Lemma 3, we know that the probability that there exists one common code between two cells is overwhelmingly greater than that of two or more common codes, so we focus on the case that two cells have no or one common code. The following Lemmas 4 and 5 give the probability that k cells have one common code.

In our B_hBF, an invalid cell is the one that cannot be decoded.

Lemma 4. *After n element-set pairs have been inserted into a* B_h*BF, the probability that a cell increased i times is* $P(X = i) = \binom{nk}{i}(\frac{1}{m})^i(1 - \frac{1}{m})^{nk-i}$, *and the probability a cell is invalid is* $P(X > h) = \sum_{i=h+1}^{nk} \binom{nk}{i}(\frac{1}{m})^i(1 - \frac{1}{m})^{nk-i}$.

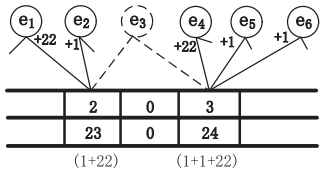
Proof. The probability that a cell is selected by a hash function is $\frac{1}{m}$. A count in a cell increased X times conforms to Binomial distribution, i.e., $X \sim Binom(X, \frac{1}{m})$. So the probability $P(X = i)$ that a cell increased i times is $P(X = i) = \binom{nk}{i}(\frac{1}{m})^i(1-\frac{1}{m})^{nk-i}$. Therefore, the probability that a counter is at least h is $\sum_{i=h+1}^{nk} \binom{nk}{i}(\frac{1}{m})^i(1 - \frac{1}{m})^{nk-i}$. □

Lemma 5. *After n element-set pairs have been inserted into a* B_h*BF, randomly select* k' *valid cells with* $2 \le k' \le k$, *the probability that these* k' *cells have only one common code is (c=1):*

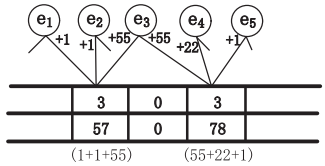
$$P_{c=1}(k') = \sum_{i=1}^{h} P(X = i) \times \sum_{j=1}^{h} (P(X = j) \times F(i, j, 1)) \times \sum_{l=1}^{h} P((X = l) \times F(1, l, 1))^{k'-2} \tag{4}$$

Proof. When $k' = 2$, the probability that two cells contain only one common code is $P(X = 1) \times [P(X = 1) \times F(1, 1, 1) + \cdots + P(X = h) \times F(1, h, 1)] + \cdots + P(X = h) \times [P(X = 1) \times F(h, 1, 1) + \cdots + P(X = h) \times F(h, h, 1)] = \sum_{i=1}^{h} P(X = i) \times \sum_{j=1}^{h} (P(X = j) \times F(i, j, 1))$.

When $k' > 2$, we can first combine the initial two together, then add the remaining cells one by one until all cells are added. The probability that each newly added cell has one common code with the previous cells all together is $\sum_{l=1}^{h} P((X = l) \times F(1, l, 1))$. Therefore, after adding $(k' - 2)$ left cells, all k' cells have one common code is Equation (4). □



(a) Classification failure:e_3 is absent but the query return UNKNOWN.



(b) Classification failure:e_3 is present but the query return UNKNOWN.

Fig. 4. The cases of classification failure.

Similarly, the probability that two valid cells have two common codes is

$$P_{c=2}(k') = \sum_{i=1}^{h} P(X=i) \times \sum_{j=1}^{h} (P(X=j) \times F(i,j,2)) \times \sum_{l=1}^{h} P((X=l) \times F(2,l,2))^{k'-2}. \tag{5}$$

Theorem 2. *After n element-set pairs have been inserted into a B_hBF, the false positive rate FPR is:*

$$FPR = \begin{cases} P(X=1) & k=1 \\ P(X=1) \times \binom{k}{1} \times (P(X>h))^{k-1} & \\ + \sum_{i=2}^{k} P_{c=1}(i) \times \binom{k}{i} \times (P(X>h))^{k-i} & k \geq 2. \end{cases} \tag{6}$$

Proof. When $k = 1$, only one cell corresponds to a queried absent element. When the cell is inserted one code by a present element, false positive occurs. Obviously, we have $FPR = P(X = 1)$.

When $k \geq 2$, false positive occurs in our B_hBF when all the valid cells in the k cells have only one common code. The probability should be the sum of following two probabilities. (a) The probability that there is one valid cell and $k-1$ invalid cells is $P(X=1) \times \binom{k}{1} \times (P(X>h))^{k-1}$. (b) The probability that there are more than one valid cell that has one common code and others are invalid cells is $\sum_{i=2}^{k} P_{c=1}(i) \times \binom{k}{i} \times (P(X>h))^{k-i}$. Therefore, when $k \geq 2$, the false positive probability is Equation (6). □

5.2 Classification Failure

Classification failure: When a query returns an uncertain result UNKNOWN, classification failure happens.

As shown in Figure 4, there are two cases when classification failures happen. In the case in Figure 4(a), suppose $\mathbf{e}_3$ is absent, but $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_4$, $\mathbf{e}_5$, and $\mathbf{e}_6$ are present and have already inserted their set ID code into the cells corresponding to $\mathbf{e}_3$. As a result, the two cells have two common codes, return UNKNOWN. In the case in Figure 4(b), $\mathbf{e}_3$ is present. Because other present elements insert their code into the cells of $\mathbf{e}_3$, the two cells corresponding to $\mathbf{e}_3$ have two common codes, return UNKNOWN.

Following, we present the probability of classification failure in two cases: when querying the absent element and when querying the present element.

Theorem 3. *After n element-set pairs have been inserted into a B_hBF, the probability that a query returns UNKNOWN for an absent element CFR_a is*

$$CFR_a = \begin{cases} P(X>1) & k=1 \\ \sum_{i=2}^{h} P(i) \times \binom{k}{1} \times (P(X>h))^{k-1} & \\ + \sum_{i=2}^{k} P_{c=2}(i) \times \binom{k}{i} \times (P(X>h))^{k-i} & k \geq 2. \end{cases} \tag{7}$$

Proof. When $k = 1$ and $L \gg h$, only one cell corresponds to a queried absent element. When the cell count is larger than 1, return UNKNOWN. Obviously, we have $CFR_a = P(X > 1)$.

When $k \geq 2$, UNKNOWN occurs in our B_hBF when all the valid cells in the k cells have more than one common code. According to the Lemma 3 we have $F(i, j, 2) \gg F(i, j, 3)$, so we focus on the case where two cells have two common codes. The probability of UNKNOWN occurrences should be the sum of the following two probabilities. (a) The probability that there is one valid cell inserted more than one time and $k - 1$ invalid cell is $\sum_{i=2}^{h} P(i) \times \binom{k}{1} \times (P(X > h))^{k-1}$. (b) The probability that there are more than one valid cell that have more than one common code and others are invalid cell is $\sum_{i=2}^{k} P_{c=2}(i) \times \binom{k}{i} \times (P(X > h))^{k-i}$ where $P_{c=2}(i)$ denotes the probability that these i valid cells have two common codes. □

Following, we give the formula of classification error rate when querying a present element.

Lemma 6. *When an element-set pair has been inserted into a cell, the probability that the count of the cell equals j is*

$$P'(X=i) = \begin{cases} 1 & i=1 \\ \binom{nk-k}{i-1} \left(\frac{1}{m}\right)^{i-1} \left(1-\frac{1}{m}\right)^{nk-k-i-1} & j \geq 2 \end{cases}. \tag{8}$$

Proof. when querying a present element, if $i = 1$ the element has been inserted into the cell, then the probability that cell count equals 1 is 100%. When $i \geq 2$, one of the codes in the cell is the element's code, and the other conforms to Binomial distribution. So the probability $P'(X = i)$ that a cell increased i times is $\binom{nk-k}{i-1}(\frac{1}{m})^{i-1}(1-\frac{1}{m})^{nk-k-i-1}$. □

Theorem 4. *After n element-set pairs have been inserted into a B_hBF, the probability that a query return UNKNOWN for a present element CFR_p is*

$$CFR_p = \begin{cases} P'(X>1) & k=1 \\ \sum_{i=2}^{h} P'(i) \times \binom{k}{1} \times (P'(X>h))^{k-1} & \\ + \sum_{i=2}^{k} P'_{c=2}(i) \times \binom{k}{i} \times (P'(X>h))^{k-i} & k>1 \end{cases}. \tag{9}$$

The proof is similar to the proof of Theorem 3, so it is abbreviated.

5.3 Comparison Between Theoretical and Empirical Values

Figure 5 shows the comparison between theoretical and empirical values of **false positive rate** (***FPR***) and failure rates of classification in two cases (CFR_a and CFR_p) when $n = 100{,}000$, k increases from 1 to 15, and m increases from 200,000, 250,000, and 300,000.

The theoretical curves and the empirical curves in Figure 5 nearly coincide with each other, which demonstrates our calculations on FPR in Equation (2), CFR_a in Equation (7), and CFR_p in Equation (9) are correct.

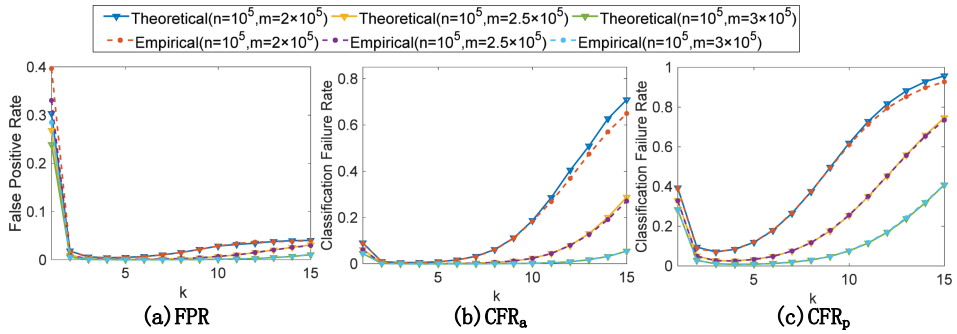


Fig. 5. Comparison of theoretical and empirical value for FPR and CFR.

Table 1. Statistics Of the Real Datasets

Dataset	No.packets	No.flows	No. forwording ports
MAWI	10 million	4,539,329	200
CAIDA	15 million	1,126,941	497

In Figure 5(a), all curves first drop to a minimum and then rise. Using more hash functions allows the query operation to be carried over more cells. It can better exclude the cells, which falsely report a false set code for a queried absent element. On the other hand, more hash functions also take more cells to store the set code of one element, which results in a higher chance of returning a set code from cells hashed by a queried absent element although it is not inserted there. All curves in Figure 5(b)–(c) follow the same rule with *FPR*: they first drop until reaching a minimum, and then rise.

6 EXPERIMENT EVALUATION

6.1 Experimental Setup

(1) **Datasets:** We use the real-world Internet datasets from MAWI and CAIDA to evaluate the proposed algorithm.

— **MAWI** [16]**:** This dataset contains 15 minutes' daily traffic captured from a transit link between Japan and US ISP starting from 14:00 September 1st 2018. We focus on the first 10 million packets of the dataset, where the packets belong to 4.54 million 5-tuple flows and are forwarded from 200 ports.

— **CAIDA** [5]**:** This dataset contains traffic collected from a high-speed commercial backbone link starting from December 21st 16:05 2018. We focus on the first 15 million packets of the dataset, where the packets belong to 1.13 million 5-tuple flows and are forwarded through 497 ports.

In our experiments, we consider flow as the element to query, the forwarding port as the set. That is, the pair <element, set> corresponds to <flow, forwarding port>. The flow is identified by using IP 5-tuple <src IP, dest IP, src port, dest port, protocol>. The statistics of these two datasets are shown in Table 1.

(2) **Platform:** We use a standard off-the-shelf desktop computer equipped with an Intel(*R*) Core i5-8300H CPU @2.30GHz and 32GB RAM running Windows 10 to do our experiments. We insert

Table 2. Parameter Setting for Mawi

Algorithm	Bits in a cell	No.Cells (i.e., m)	Total bits (memory allocation)
B_hBF	28 = 3(count) + 25(sum)	1.2×10^7	3.36×10^8
IBF	26 = 2(count) + 8(sum) +16(hashkeysum)	12,923,077	3.36×10^8
KBF	19 = 2(count) + 17(value)	17,684,211	3.36×10^8
NBF	–	–	3.36×10^8
COMB	–	–	3.36×10^8
FlatBF	64	$131{,}250 \times 4$	3.36×10^8

Table 3. Parameter Setting for Caida

Algorithm	Bits in a cell	No.Cells (i.e., m)	Total bits (memory allocation)
B_hBF	32 = 3(count) + 29(sum)	2.8×10^6	8.96×10^7
IBF	27 = 2(count) + 9(sum) +16(hashkeysum)	3,318,519	8.96×10^7
KBF	23 = 2(count) + 21(value)	3,895,653	8.96×10^7
NBF	–	–	8.96×10^7
COMB	–	–	8.96×10^7
FlatBF	64	$175{,}000 \times 8$	8.96×10^7

the timer into algorithm implementations to evaluate the computation time. For each experiment setting, we run the experiments ten times with the random seeds and get the average of the results.

(3) **Baseline algorithms and parameter setting:** Besides our B_hBF, we implement other five BF-based multi-set query algorithms IBF [11], KBF [25], NBF [8], COMB [12], and FlatBF [7]. We implement all the algorithms in java using Eclipse platform. We do not use parallelism in the implementation and all our data structures are held in RAM. These algorithms are all hash-based. In our implementation, we use MurmurHash3 provided by Google Guava [1].

In our B_hBF, to use B_h sequence to encode the forwarding port, we generate a B_h sequence offline. As the sum of any h integers is unique in a B_h sequence, we use a hash table to store the used sum value and its h integers. With the aid of this hash table, we can easily decode the B_hBF cell to return the forwarding port during the query.

To ensure the fairness of performance comparison, we allocate the same amount of memory to all the implemented algorithms instead of setting the same m (the number of bits or cells in BFs). We have done two types of experiments.

— For the first type, in the default memory setting, we allocate 3.36×10^8bit ($\approx$ 40.1 MB) memory for MAWI and 8.96×10^7bit ($\approx$ 10.7 MB) memory for CAIDA. As shown in Table 1, the number of flows is 4,539,329 and 1,126,941 for MAWI and CAIDA, respectively. Accordingly, we allocate $3.36 \times 10^8 \div 4539329 \approx 74.02$ "bits per element-set pair" and $8.96 \times 10^7 \div 1126941 \approx 79.51$ "bits per element-set pair" for MAWI and CAIDA, respectively. The detailed memory settting under the algorithms implemented is listed in Tables 2 and 3. We compare the correctness rate, false positive rate, query speed, and memory access overhead, and the results will be shown in Sections 6.4, 6.5, 6.7, and 6.8.

— For the second type, by varying the memory allocation, we further investigate the efficiency of different schemes under different memory costs, the result will be shown in Section 6.6.

As the total memory impact the length (i.e., m) of the BF, Tables 2 and 3 provide the detailed parameter setting using the default memory allocation.

NBF and COMB are formed as an array of bits, while IBF, KBF, and our B_hBF are in the form of an array of cells. Therefore, in Tables 2 and 3, for NBF and COMB, we only list the total bits to reflect the total memory consumption. A FlatBF array includes 64 standard BFs with each corresponding to a set. For MAWI and CAIDA datasets, we have 200 and 497 set IDs, respectively, thus $\lceil \frac{200}{64} \rceil = 4$ and $\lceil \frac{497}{64} \rceil = 8$ FlatBF arrays. Given the total memory allocation in MAWI (3.36×10^8bit) and CAIDA (8.96×10^7bit), we have 4 and 8 FlatBF arrays in MAWI and CAIDA, respectively.

The cell sizes are different in IBF, KBF, and our B_hBF, as they encode the set IDs into different lengths of codes. The number of codes determines the maximum length of code. The cell size is determined by the maximum length of code. The two datasets in evaluation require different number of codes, one is 200 codes and the other is 497 codes. The cell size is different in the two datasets, as shown in Tables 2 and 3. Following we take MAWI as example to illustrate the detailed parameter setting.

According to [4], we know that the maximum value of L codes in a B_h sequence is less than L^h. For our B_hBF, we use B_3 sequence to encode the set ID. Therefore, the maximum code is less than 200^3 (23 bits) for MAWI. Each cell in our B_hBF includes a count field and a sum field. For a cell whose count is not larger than 4, we will check for the decoding. Therefore, we allocate 3 bits to the count field, and 25 bits (4×200^3) to the sum field.

Each cell in IBF includes a count field, a sum field, and a hashkeysum filed. For a query, IBF only checks the cell with its counter less than 2. Therefore, we only allocate 2 bits for the count filed and 8 bits for the sum filed for MAWI, as IBF directly stores the sum ID without encoding. In the experiments, the key is the IP 5-tuple. We apply Murmurhash3 to the key to generate the 32-bit hash value and treat the low 16 bits as the fingerprint of the element in IBF. We allocate 16 bits to the hashkeysum field.

Each cell in KBF includes a count field and a sum field. According to the encoding algorithm in KBF, we generate 200 and 497 codes for MAWI and CAIDA, the maximum code lengths are 17 and 21, respectively. Inserting a code into KBF only requires the XOR operation with the code stored in the sum field. Therefore, the size of the sum field is equal to the code size and we set the sum size of KBF to 17 (MAWI) and 21 (CAIDA), respectively. If the counter is larger than 2, KBF can hardly perform the decoding. We only allocate 2 bits to the count field.

6.2 Validation of Strategy 1

Figure 6 investigates the time spent in microseconds on a query with the change of the number of hash function k, under our B_hBF with (B_hBF-1) and without Strategy 1 (B_hBF-No-1) in Section 4.2.1. B_hBF-1 can achieve higher query speed, which demonstrates that by reducing redundant decoding and intersection operations, Strategy 1 is very effective in speeding up the query process. For example, in Figure 6(a), the time spent on a query is 0.74μs under B_hBF-1 and 1.53μs under B_hBF-No-1 when $k = 4$ for CAIDA dataset, that is the speed under B_hBF-1 is up to nearly 2 times that under B_hBF-No-1.

6.3 Validation of Strategy 2

To validate the effectiveness of the proposed Strategy 2, we implement two versions of our B_hBF, with (B_hBF-2) and without (B_hBF-No-2) Strategy 2. We use metric unknown rate, i.e., the rate that the number of "UNKNOWN" results to the total number of queries, to evaluate effectiveness. Smaller unknown rate means better query accuracy.

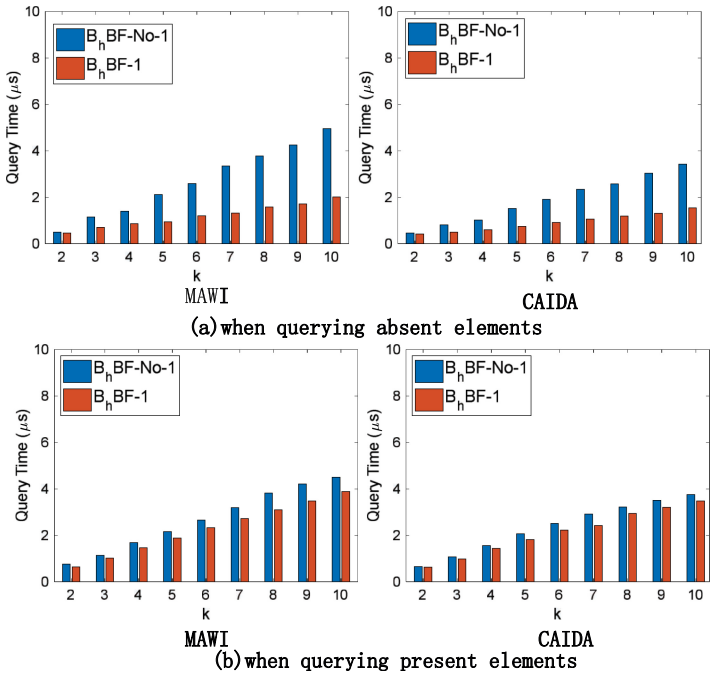


Fig. 6. Validation of strategy 1.

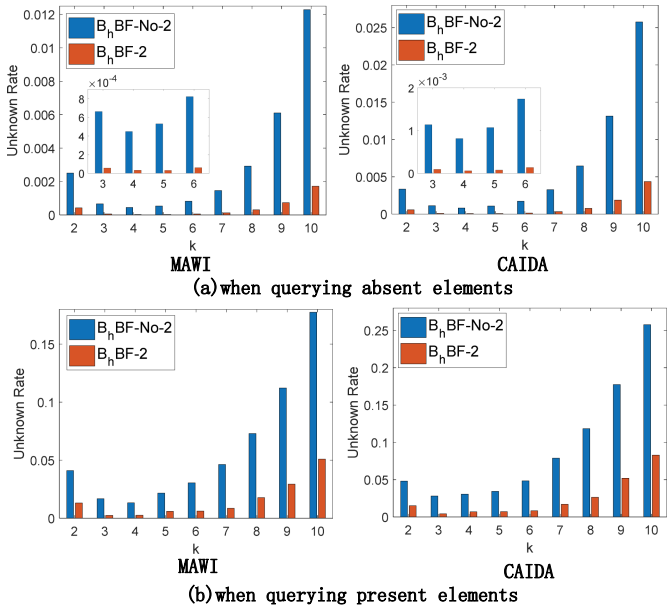


Fig. 7. Validation of strategy 2.

In Figure 7, obviously, the unknown rate under B_hBF-2 is much lower than that under B_hBF-No-2 in all the query scenarios using different datasets. Without Strategy 2, the unknown rate under B_hBF-No-2 is up to 40 times higher than that under our B_hBF-2. These results demonstrate that Strategy 2 is very effective in increasing the query accuracy of our B_hBF.

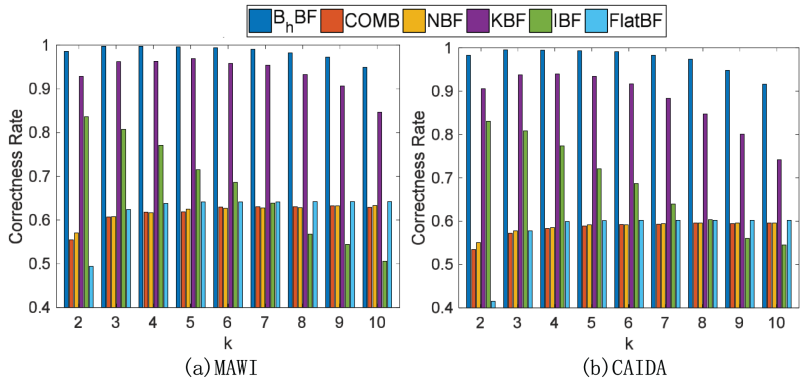


Fig. 8. Correctness rate of MAWI and CAIDA when 1% of elements change their set ID.

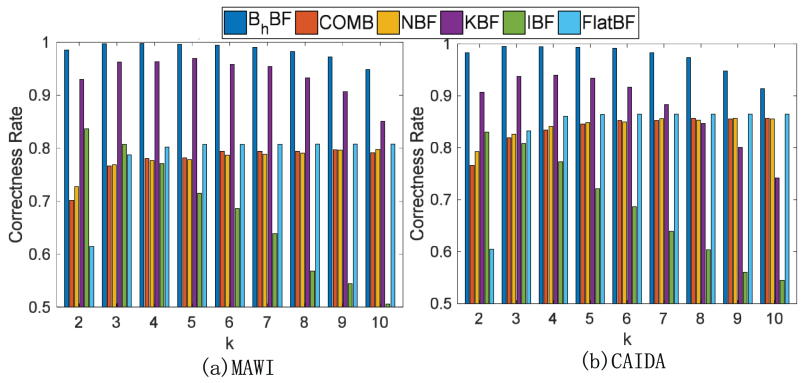


Fig. 9. Correctness rate of MAWI and CAIDA when 0.1% of elements change their set ID.

6.4 Correctness Rate

To query the set ID of an element, if the element-set pair is present and has been inserted into the B_hBF, and the set ID returned is correct, we say the element's query is correct. The correctness rate measures the proportion of the queried elements whose query results are correct.

We have conducted two groups of experiments, one to investigate how the update of set ID impacts the query performance, and the other to investigate how the load (i.e., the number of element-set pairs n) impacts the query performance.

6.4.1 Impact of the Update of Set ID. Figures 8–10 respectively show the query results under following the three cases: (Case 1) 1% elements change their set IDs after being inserted. (Case 2) 0.1% elements change their set IDs after they are inserted; and (Case 3) no element changes its set ID after the element-set pairs have been inserted;

In Figures 8 (Case 1) and 9 (Case 2), the correctness rate of our B_hBF is the highest under all scenarios with different number of hash functions. B_hBF, KBF, and IBF can support the update operation, while FlatBF, COMB, and NBF can not. FlatBF, COMB, and NBF drop to nearly 63% for MAWI and 59% for CAIDA, a very low correctness rate, even when only 1% elements change their set IDs, as these algorithms use bit array and can hardly support update operations. Besides our

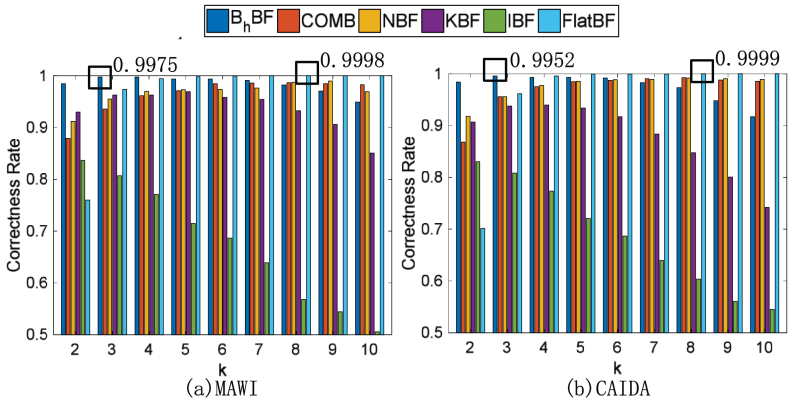


Fig. 10. Correctness rate of MAWI and CAIDA when pairs do not change.

Table 4. The Standard Error of Correctness Rate When Pairs Do Not Change for Mawi

k	2	3	4	5	6	7	8	9	10
B_hBF	0.00162	0.00096	0.00018	0.00123	0.00065	0.00133	0.00078	0.00144	0.00262
COMB	0.00334	0.00045	0.00017	0.00118	0.000451	0.00084	0.00042	0.00189	0.00247
NBF	0.00464	0.00319	0.00198	0.00099	0.00086	0.00120	0.00072	0.00185	0.00115
KBF	0.00259	0.00074	0.00015	0.00218	0.00169	0.00024	0.00249	0.00747	0.00048
IBF	0.00374	0.00518	0.00681	0.00950	0.00507	0.00929	0.0083	0.01337	0.00952
FlatBF	0.00438	0.00302	0.00070	0.00160	0.00001	0.00001	0.00001	0.000001	0.000001

B_hBF, although KBF and IBF have the ability of supporting update operation, the correctness rate is much lower than B_hBF.

In Figure 10 (Case 3), when $k = 3$, B_hBF achieves its highest correctness rate, 0.9975 for MAWI and 0.9952 for CAIDA, respectively. FlatBF and our B_hBF are the best two algorithms that achieve the highest correctness rate. Although FlatBF is slightly better than our B_hBF in the scenario with no element change in Figure 10, B_hBF achieves much higher correctness rate in the scenario with elements' change in Figures 8 and 9.

Among the three cases in Figures 8–10, and compared with Case 3, B_hBF performs much better in handling the dynamic environment where an element will change its set ID after the element-set pair is inserted.

Since the performance difference in Figure 10 is small, we report the **standard error** (**SE**) of correctness rate in Tables 4 and 5. Given n experimental values $\{X_1, \ldots, X_n\}$ and their mean $\bar{X}$, the calculation formula of SE is $\frac{SD}{\sqrt{n}}$, where SD is the standard deviation of experimental values calculated by $SD = \sqrt{\frac{\sum_{i=1}^{n}(X_i - \bar{X})^2}{n-1}}$. As shown in Tables 4 and 5, the standard error is small, and it is reasonable that we use the average of the correctness rate. Due to space constraints, we only report the SE of correctness rate under Case 3.

6.4.2 Impact of Load. In Figure 11, it shows how the load (i.e., the number of element-set pairs n) impacts query performance. All algorithms adopt the best k according to Figure 10. Since the NBF, COMB, and FlatBF can not support update operations, the correctness rate is very low even when the load rate is 10%. For the algorithms (B_hBF, KBF, and IBF) that can support update operations,

Table 5. The Standard Error of Correctness Rate When Pairs Do Not Change for Caida

k	2	3	4	5	6	7	8	9	10
B_hBF	0.00058	0.00041	0.00029	0.00046	0.00044	0.00043	0.00081	0.00093	0.00181
COMB	0.00419	0.00057	0.00185	0.00062	0.00197	0.00139	0.00082	0.00072	0.00158
NBF	0.00233	0.00101	0.00086	0.00136	0.00079	0.00031	0.00064	0.00044	0.00039
KBF	0.00171	0.00238	0.0007	0.00054	0.00211	0.00098	0.00213	0.00062	0.00047
IBF	0.00275	0.00432	0.00371	0.00369	0.00563	0.00273	0.00493	0.002558	0.00314
FlatBF	0.00251	0.00059	0.00048	0.00014	0.00002	0.00002	0.00001	0.000001	0.000001

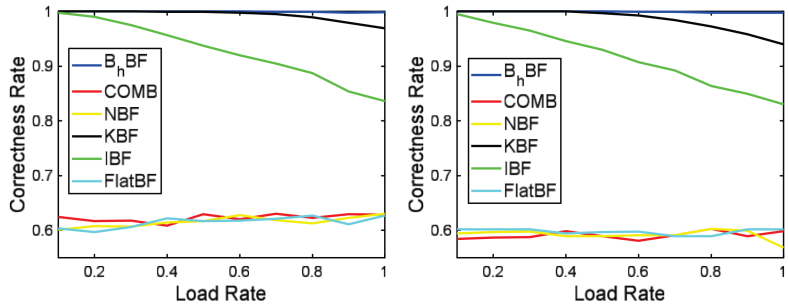


Fig. 11. Correctness rate vs. load rate.

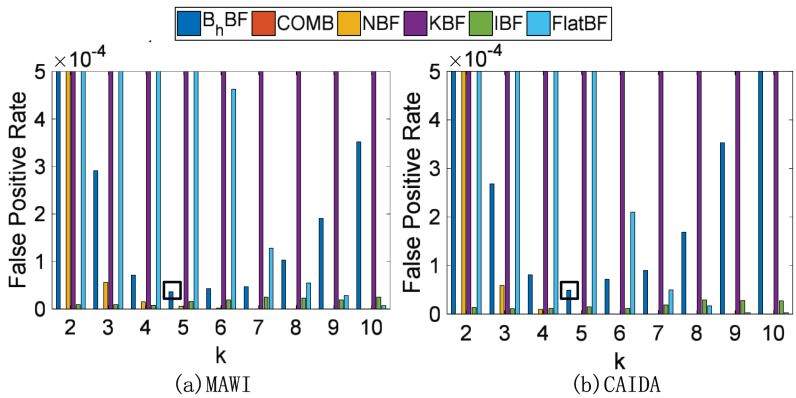


Fig. 12. False positive rate vs. k.

B_hBF declines very slowly with the load rate increases. Even when the load rate reaches 1 (i.e., all element-set pairs in the dataset are inserted), the correctness rate under our B_hBF is nearly 1.

6.5 False Positive Rate

Consistent with Figure 5(a), in Figure 12 with the increase of the number of hash functions (i.e., k), the false positive rate under our B_hBF first drops until reaching a minimum (0.000036 for MAWI and 0.000049 for CAIDA when $k = 5$), and then rises. Although the false positive rate under COMB, NBF, and FlatBF are slightly lower than that under our B_hBF, they do not support the update operation, as shown in Figures 8 and 9. COMB uses one group of k hash functions to verify each bit in the query result, and multiple groups of hash functions to check the whole result, which

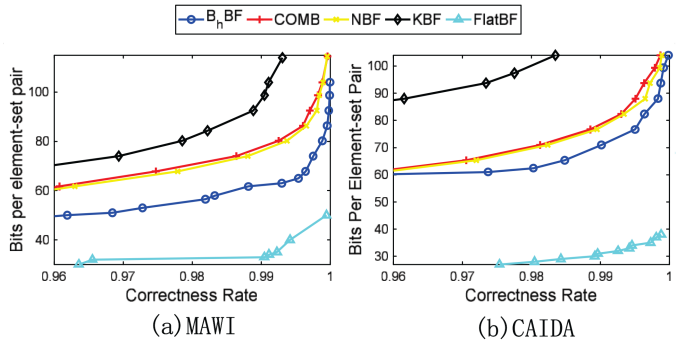


Fig. 13. Space efficiency vs. correctness rate.

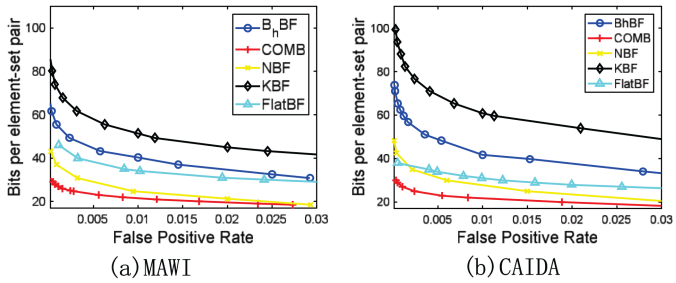


Fig. 14. Space efficiency vs. false positive rate.

results in much lower query processing speed and much higher memory access overhead, shown in Figures 15 and 16, respectively. Note that the false positive rate under IBF is low, since it not only stores the set ID but also the hash value of the element. When performing the query algorithm, the code is returned only if the element information of a queried element matches. However, as discussed in Section 2.3, in IBF, if hashing collisions occur more than once in a cell, the cell cannot be decoded, which results in a very low query correctness rate under IBF in Figure 10.

6.6 Space Efficiency

We investigate the efficiency of different schemes by comparing their space cost. We apply the metric "bits per element-set pair", which is calculated as the average number of bits to represent each element-set pair in a full filter. It is the ratio of the total bit cost of the filter memory and the total number of element-set pairs that a filter stores.

Figure 13 shows the space cost ("bits per element-set pair") under different correctness rates from 96% to 100%. The correctness rate is achieved under the best k. For IBF, when the correctness rate reaches 96%, it requires 171.83 "bits per element-set pair", a cost larger than the size of IP 5-tuple (104 bits). Therefore, we do not draw IBF in the figure. As expected, with the increase of the correctness rate, the space cost increases under all implemented algorithms. In Figure 8, only KBF our B_hBF support the update operation. Compared with KBF, the space cost under B_hBF is much smaller.

Figure 14 shows the space cost under different false positive rates from 0.0001 to 0.03. As space cost does not impact the false positive rate in IBF, we do not draw the curve of IBF. In Figures 8 and 9, for the two algorithms (KBF and B_hBF) that support update operation, we can find our B_hBF is more space efficient. When the false positive ratio is controlled to be 0.01, B_hBF requires 40.3

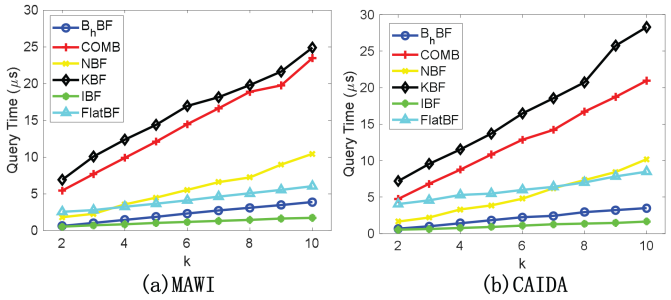


Fig. 15. Query precessing speed.

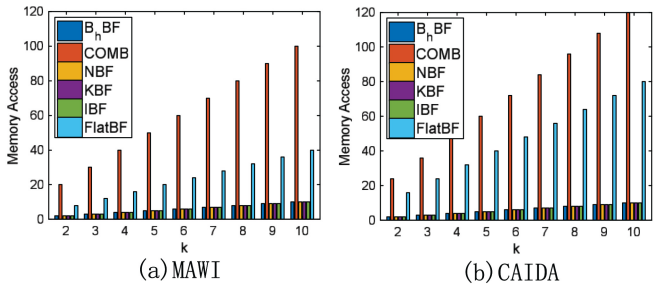


Fig. 16. Memory access.

and 40.7, while KBF requires 51.4 and 60.8 bits per element for MAWI and CAIDA dataset. B_hBF requires 11 and 20 fewer bits per element than KBF for MAWI and CAIDA datasets, respectively.

6.7 Query Speed

We investigate a query processing time of different schemes with the change of the number of hash functions k. Figure 15 compares the query speed under all algorithms implemented. As expected, with the increase of k, more hash functions are involved, so the query speed of all algorithms decreases. For the MAWI dataset, when using the optimal k according to Figure 10(a), the three fastest schemes are IBF, B_hBF, and FlatBF whose query processing durations are respectively 0.52μs, 1.04μs, and 4.12μs. For the CAIDA dataset, the three fastest schemes are IBF, B_hBF, and NBF whose query processing durations are 0.53μs, 0.98μs, and 6.21μs, respectively. IBF is a little faster than us, but its best correctness rate is no more than 84%, while B_hBF is more than 99.5% as shown in Figure 10.

6.8 Memory Access Overhead

Figure 16 shows the number of memory accesses for a query operation under all algorithms implemented. Obviously, the memory access overheads under COMB and FlagBF are the highest two. COMB uses multiple groups of hash functions and the number of groups is determined by the length of the set code, while all other algorithms including our B_hBF use one group of k hash functions.

6.9 Performance summarization

Table 6 summarizes the performance of different metrics. For each metric, we use "++" and "+" to respectively denote the best algorithm and the next best algorithm. Obviously, our B_hBF

Table 6. Algorithm Performance Comparison

Metric	B_hBF	COMB	NBF	KBF	IBF	FlatBF
Correctness rate	+					++
Impact of load	++			+		
False positive rate		++	+			
Space efficiency vs Correctneee rate	+					++
Space efficiency vs False positive rate		++	+			
Speed	+				++	
Memory access	++		++	++	++	
Support update	++			+		

"++": optimal performance; "+": suboptimal performance.

outperforms other algorithms with the capability of supporting the update operation, and it can achieve a high correctness rate with low space cost and low memory access overhead.

7 CONCLUSION

In this article, we design B_hBF, a novel compact probabilistic data structure to support multi-set membership query with four basic operations: insertion, query, deletion, and update. We propose two strategies, one is by reducing redundant operations to speed up the query process, and the other is by well exploiting invalid cells to increase the query accuracy. We analyze theoretically the false positive and classification failure rate of our B_hBF. We have done extensive experiments using two real world datasets to evaluate the effectiveness and efficiency of our B_hBF.

REFERENCES

[1] Hash website. Retrieved on 11 Jan. 2022 from https://guava.dev/releases/19.0/api/docs/com/google/common/hash/Hashing.html.

[2] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 422–426.

[3] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrah, Sushil Singh, and George Varghese. 2006. Beyond bloom filters: From approximate membership checks to approximate state machines. *ACM SIGCOMM Computer Communication Review* 36, 4 (2006), 315–326.

[4] Raj Chandra Bose and Sarvadaman Chowla. 1960. *Theorems in the Additive Theory of Numbers*. Technical Report. North Carolina State University. Dept. of Statistics.

[5] The CAIDA Anonymized Internet Traces. Retrieved on 11 Jan. 2022 from http://www.caida.org/data/.

[6] Francis Chang, Wu-chang Feng, and Kang Li. 2004. Approximate caches for packet classification. In *Proceedings of the IEEE INFOCOM 2004*, Vol. 4. 2196–2207.

[7] Adina Crainiceanu and Daniel Lemire. 2015. Bloofi: Multidimensional bloom filters. *Information Systems* 54 (2015), 311–324.

[8] Haipeng Dai, Yuankun Zhong, Alex X Liu, Wei Wang, and Meng Li. 2016. Noisy bloom filters for multi-set membership testing. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*. 139–151.

[9] HARM DERKSEN. 2004. Error-correcting codes and b_h-sequences. *IEEE Transactions on Information Theory* 50, 3 (2004), 476–485.

[10] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking* 8, 3 (2000), 281–293.

[11] Michael T Goodrich and Michael Mitzenmacher. 2011. Invertible bloom lookup tables. In *Proceedings of the 2011 49th Annual Allerton Conference on Communication, Control, and Computing*. IEEE, 792–799.

[12] Fang Hao, Murali Kodialam, TV Lakshman, and Haoyu Song. 2012. Fast dynamic multiple-set membership testing using combinatorial bloom filters. *IEEE/ACM Transactions on Networking* 20, 1 (2012), 295–304.

[13] Jianyuan Lu, Tong Yang, Yi Wang, Huichen Dai, Xi Chen, Linxiao Jin, Haoyu Song, and Bin Liu. 2018. Low computational cost bloom filters. *IEEE/ACM Transactions on Networking* 26, 5 (2018), 2254–2267.

[14] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. 2018. Optimizing Bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1912–1949.
[15] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard TB Ma, Xueshan Luo, and Bangbang Ren. 2019. The consistent Cuckoo filter. In *Proceedings of the IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. 712–720.
[16] The MAWI Working Group Traffic Archive. [n.d.]. Retrieved on 11 Jan. 2022 from http://mawi.nezu.wide.ad.jp/mawi/.
[17] Michael Mitzenmacher, Pedro Reviriego, and Salvatore Pontarelli. 2016. OMASS: One memory access set separation. *IEEE Transactions on Knowledge and Data Engineering* 28, 7 (2016), 1940–1943.
[18] Jiangbo Qian, Zhipeng Huang, Qiang Zhu, and Huahui Chen. 2018. Hamming metric multi-granularity locality-sensitive bloom filter. *IEEE/ACM Transactions on Networking* 26, 4 (2018), 1660–1673.
[19] Yan Qiao, Shigang Chen, Zhen Mo, and Myungkeun Yoon. 2016. When bloom filters are no longer compact: Multi-set membership lookup for network applications. *IEEE/ACM Transactions on Networking* 24, 6 (2016), 3326–3339.
[20] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. 2013. The variable-increment counting Bloom filter. *IEEE/ACM Transactions on Networking* 22, 4 (2013), 1092–1105.
[21] Simon Sidon. 1932. Ein satz *ü*ber trigonometrische Polynome und seine Anwendung in der Theorie der Fourier-Reihen. *Mathematische Annalen* 106, 1 (1932), 536–539.
[22] Lu Tang, Qun Huang, and Patrick P. C. Lee. 2019. MV-Sketch: A fast and compact invertible sketch for heavy flow detection in network data streams. In *Proceedings of the IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. 2026–2034.
[23] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2011. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys and Tutorials* 14, 1 (2011), 131–155.
[24] Yang Tong, Dongsheng Yang, Jie Jiang, Siang Gao, Bin Cui, Lei Shi, and Xiaoming Li. 2019. Coloring embedder: A memory efficient data structure for answering multi-set query. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering*. 1142–1153.
[25] Sisi Xiong, Yanjun Yao, Shuangjiang Li, Qing Cao, Tian He, Hairong Qi, Leon Tolbert, and Yilu Liu. 2017. KBF: Towards approximate and bloom filter based key-value storage for cloud computing systems. *IEEE Transactions on Cloud Computing* 5, 1 (2017), 85–98.
[26] Tong Yang, Alex X Liu, Muhammad Shahzad, Dongsheng Yang, Qiaobin Fu, Gaogang Xie, and Xiaoming Li. 2017. A shifting framework for set queries. *IEEE/ACM Transactions on Networking* 25, 5 (2017), 3116–3131.
[27] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X Liu, Qi Li, and Laurent Mathy. 2014. Guarantee IP lookup performance with FIB explosion. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 39–50.
[28] Myung Keun Yoon, JinWoo Son, and Seon-Ho Shin. 2014. Bloom tree: A search tree based on bloom filters for multiple-set membership testing. In *Proceedings of the IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 1429–1437.
[29] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. 2009. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*. 313–324.

Received April 2021; revised September 2021; accepted November 2021