Multiple Set Matching with Bloom Matrix and Bloom Vector

FRANCESCO CONCAS, University of Helsinki, Finland
PENGFEI XU, University of Helsinki, Finland
MOHAMMAD A. HOQUE, University of Helsinki, Finland
JIAHENG LU, University of Helsinki, Finland
SASU TARKOMA, University of Helsinki, Finland

Bloom Filter is a space-efficient probabilistic data structure for checking the membership of elements in a set. Given multiple sets, a standard Bloom Filter is not sufficient when looking for the items to which an element or a set of input elements belong. An example case is searching for documents with keywords in a large text corpus, which is essentially a multiple set matching problem where the input is single or multiple keywords, and the result is a set of possible candidate documents. This article solves the multiple set matching problem by proposing two efficient Bloom Multifilters called Bloom Matrix and Bloom Vector, which generalize the standard Bloom Filter. Both structures are space-efficient and answer queries with a set of identifiers for multiple set matching problems. The space efficiency can be optimized according to the distribution of labels among multiple sets: Uniform and Zipf. Bloom Vector efficiently exploits the Zipf distribution of data for further space reduction. Indeed, both structures are much more space-efficient compared with the state-of-the-art, Bloofi. The results also highlight that a Lookup operation on Bloom Matrix is significantly faster than on Bloom Vector and Bloofi.

CCS Concepts: • **Theory of computation → Bloom filters and hashing**; • **Information systems → Query Optimization**.

Additional Key Words and Phrases: Bloom Filter, Multiple Sets, Bloomier Filter, Uniform Distribution, Zipf Distribution, Big Data.

ACM Reference Format:
Francesco Concas, Pengfei Xu, Mohammad A. Hoque, Jiaheng Lu, and Sasu Tarkoma. 2019. Multiple Set Matching with Bloom Matrix and Bloom Vector. *ACM Trans. Knowl. Discov. Data.* 1, 1, Article 1 (January 2019), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern popular Internet services, including Google search, Yahoo directory, and web-based storage services, rely on efficient data matching [5]. These services use custom techniques for providing scalable, fault-tolerant and low-cost services [21], [22], [25]. Fast matching of arbitrary identifiers to specific values is a fundamental requirement of these applications in which data objects are organized using unique local or global identifiers, usually called labels [16]. In a typical usage scenario, each label maps to a set of values [17].

Authors' addresses: Francesco Concas, University of Helsinki, Finland, francesco.concas@helsinki.fi; Pengfei Xu, University of Helsinki, Finland, pengfei.xu@helsinki.fi; Mohammad A. Hoque, University of Helsinki, Finland, mohammad.a.hoque@helsinki.fi; Jiaheng Lu, University of Helsinki, Finland, jiaheng.lu@helsinki.fi; Sasu Tarkoma, University of Helsinki, Finland, sasu.tarkoma@helsinki.fi.

1556-4681/2019/1-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

A probabilistic data structure, called Bloom Filter, can represent a set of labels and can answer whether a label belongs to a particular set with a probability of false positives (FPs). The usage of Bloom Filters in networking is widespread, which are suitable to summarize the contents of a P2P network for supporting collaborative operations [11], to enhance probabilistic algorithms for locating resources [23], for taking routing decisions [24], and for traffic monitoring [26].

1.1 Motivation

Although the standard Bloom Filter is a very space-efficient structure, it can only answer whether a label l belongs to a set e, with a probability of false positives. There are various extensions of the Bloom Filter, such as space-code [19], spectral [9], and Shift Bloom Filter [29]. These structures represent multisets, where a label can exist in multiple times in the same set and can answer multiplicity queries, like counting the occurrence of a label in a multiset.

Modern Internet services are hosted in the cloud for faster scaling. A large cluster of servers may provide multiple services, and different servers may host various services. A recent example is serverless computing [15]. A service provider may want to find the physical servers, which host a particular service or a few services, and also to avoid code replication for a faster load balancing. Similarly, thousands of caching servers cache millions of YouTube videos, and a server may host thousands of videos [13, 18]. Whenever a client needs to access a video, one of the local caches may handle the request. The service provider may need to find the proxy servers that cache contents to avoid content replication. In these cases, the query response for a particular video can be the caching servers.

Document search in a large corpus with keywords is another important problem, where each of the documents can have hundreds of words, and the corpus can have millions of documents. When there are thousands or millions of documents in a corpus or cached elements/services in hundreds of servers, the relations can be represented using multiple sets. The union of such sets is essentially a multiset. Although space-code, spectral, or Shift Bloom Filters can answer multiplicity queries, in these cases, simple yes/no answers may not be sufficient. Instead, a query may ask the documents in which such keywords appear.

1.2 Bloom Multifilters

Let $L = \{l_1, ..., l_{|L|}\}$ be a set of labels and $E = \{e_1, ..., e_N\}$ be a set of items, where $|L|$ and N denote the size of the corresponding set. We are interested in representing the function $f : L \longrightarrow \mathcal{P}(E)$, where $\mathcal{P}(E)$ is the power set of E. All of the Bloom Filter extensions mentioned above do not assist in locating *multiple* sets when finding a single or multiple labels.

This article presents two Bloom Multifilters (BMFs), namely Bloom Matrix and Bloom Vector, for representing the above relation, f, between a label and multiple sets, each set containing unique elements. While both structures use multiple Bloom Filters to represent multiple sets, a Bloom Matrix always contains Bloom Filters having equal size, whereas the Bloom Filters in a Bloom Vector can have different sizes. These new data structures associate multiple sets and support queries with single or multiple labels. This is achieved through an inverted indexing solution. The Bloom Matrix and the Bloom Vector, through an efficient $\textsc{Lookup}(l)$ operation, can return a set or list of items, formally $S \in \mathcal{P}(E)$, rather than a simple true or false answer.

If we represent the services or contents as labels L and the servers as a set of items E, we can apply BMFs to proxy caches of services for mapping services/contents to corresponding servers. Likewise, these structures can be applied as an alternative solution to represent large text corpora. For example, the Wikipedia corpus is the set of all documents or items, i.e., E, while the unique words in the whole corpus are the set of labels, L. The items could also be encoded as Solr indices rather than using actual words so that a search query returns a list of document identifiers.

Notations	Implications
l, L	a label and a set of labels (words)
e, E	an item and set of items (documents)
h, H	a hash function and a set of hash functions
m	the number of bits in one Bloom Filter, calculated with Eq. 2a
N	the total number of items (documents)
k	the number of hash functions in one Bloom Filter, calculated with Eq. 2b
p	the probability of false positive (FP), calculated with Eq. 1
n	the number of items inserted into the structure specified in the context

Table 1. Notations used for Bloom Filter and Multifilters.

1.3 Contributions

In this article, we present two new BMFs for fast and space-efficient matching of arbitrary identifiers to sets. Similarly to using standard Bloom Filters, this comes at the cost of introducing an FP rate. Our contributions are the following:

- We introduce two new Bloom Filter-based structures, namely Bloom Matrix and Bloom Vector, to solve the labels-to-sets matching problem. These data structures are inspired by the standard Bloom Filter and some of its extensions (see Section 6), in particular, the Bloomier filter [8] and the Bloom Multifilter by Xu et al. [28]. A Bloomier Filter can encode arbitrary functions, but only on a single set, whereas a Bloom Matrix and a Bloom Vector can encode multiple sets. Compared to the Bloom Multifilter by Xu et al. [28], a Lookup operation on the Bloom Matrix or Bloom Vector returns the set of items, or their identifiers, instead of a simple true or false answer.
- We theoretically analyze these structures and evaluate their performance with different configuration parameters and label distributions using synthetic and the 20 Newsgroups corpus test datasets [6]. Our results highlight that the basic Add and Lookup operations on a Bloom Matrix are faster than those on a Bloom Vector. Compared to the Bloofi [10], both Bloom Matrix and Bloom Vector are space-efficient. Bloom Vector exploits the distribution of labels for further space reduction. The Lookup operation is faster on a Bloom Matrix than on a Bloom Vector or a Bloofi.

The rest of this article is organized as follows. Section 2 explores the standard Bloom Filter and its properties and formalizes the research problem for multiple set matching. In Sections 3–4, we present the definitions and theoretical analysis of the Bloom Matrix and the Bloom Vector. Section 5 evaluates the performance of the proposed BMFs. Section 6 presents the related work. Finally, the paper is concluded in Section 7.

2 PRELIMINARIES

A Bloom Filter [2] is a probabilistic data structure to represent a set. It occupies much less memory space compared to a conventional representation method. This memory saving comes at the cost of introducing FPs. An FP occurs when a reported item does not exist in the set. A standard Bloom Filter does not produce any false negative (FN), which occurs when a label is reported absent, but it is actually in the set. Table 1 summarizes the notations used to describe the Bloom Filter and its variants.

2.1 Bloom Filter

Definition. Let U be the universe of all labels, and $L \subseteq U$ be a set of labels to be represented by a Bloom Filter. We are interested in encoding the function $f : U \longrightarrow \{0, 1\}$ which is defined as

$$f(l) = \begin{cases} 1, & \text{if } l \in L \\ 0, & \text{otherwise} \end{cases} .$$

To encode $f(l)$, a Bloom Filter can be used, which is a pair (B, H), where B is a bit set of size m and $H = \{h_1(l), ..., h_k(l)\}$ is a set of hash functions, each having image $[0, m-1]$. Initially, all the bits in a Bloom Filter, B, are set to 0. There are two basic operations on a Bloom Filter: Add(l) and Lookup(l). The set of distinct values returned by all the hash functions for an input label l is called the *hash neighborhood* of l; with abuse of notation, we defined it as $H(l)$.

Add. To add a label l to a Bloom Filter (B, H), the Add(l) function sets $B[i]$ for each $i \in H(l)$ to 1.

Lookup. To look up a label l, a Lookup(l) operation checks whether all of the bits $B[i]$ for each $i \in H(l)$ are set to 1. If this is the case, then the label is probably in the set; otherwise, the label is certainly not in the set.

False positive rate. FPs occur for a l that is not in the set, and the Lookup function returns true. Such function returns true whenever all the bits in the hash neighborhood of l are set to 1. Therefore, with more labels added, more bits are set to 1, and the FP rate increases. The number of hash functions also influences the FP rate. Although a higher number of hash functions decreases the chance of collisions between two different labels, too many hash functions set too many bits to 1. Consequently, the choice of an optimal number of hash functions is a compromise.

Bose et al. [4] have shown that the probability p of false positives in a Bloom Filter of size m, having n labels each encoded with k hash functions is

$$p = \Theta\left(\left[1 - \left(1 - \frac{1}{m}\right)^{kn}\right]^k\right) = \Theta\left(\left(1 - e^{-kn/m}\right)^k\right). \tag{1}$$

If the number of elements, n, to be added in a Bloom Filter is known beforehand, it is possible to choose the parameters so that the Bloom Filter will have a probability of false positives around a particular value p. If n and p are known, we derive from Equation 1:

$$m = -n\frac{\ln p}{\ln^2 2}, \tag{2a}$$

$$k = \ln 2 \cdot \frac{m}{n} = -\log_2 p. \tag{2b}$$

2.2 Problem Definition

The standard Bloom Filter can encode only one set and can answer only whether a label belongs to such set or not. This article extends the standard Bloom Filter so that not only it is possible to encode multiple sets and efficiently to check the membership of a label or multiple labels in all sets, but also to answer to which of the sets the label/labels belongs.

Let $L = \{l_1, ..., l_{|L|}\}$ be a set of labels and $E = \{e_1, ..., e_N\}$ be a set of items. We are interested in representing the function $f : L \longrightarrow \mathcal{P}(E)$, where $\mathcal{P}(E)$ is the power set of E. In order to do this, we use bit sets to store the binary representation of the element $S \in \mathcal{P}(E)$ to which the labels map. The binary representation supports two functions: Encode and Decode.

2.3 Encode and Decode

When using a binary representation, it is required to have a pair of functions to translate items to a binary encoding and vice versa. Let Π be the ordering on E. The $\textsc{Encode}(\Pi, S)$ function on a set of items $S \in \mathcal{P}(E)$ returns a binary representation $V = \{v_1, ..., v_N\}$ of S, such that

$$v_i = \begin{cases} 1, & \text{if } \Pi(i) \in S \\ 0, & \text{otherwise} \end{cases}.$$

In contrast, $\textsc{Decode}(\Pi, V)$ returns S, given an ordering Π of E and the binary representation V of a set of items $S \in \mathcal{P}(E)$. $\textsc{Encode}$ and $\textsc{Decode}$ essentially associate each element to a bit in a binary representation, as explained in Example 2.1:

Example 2.1. Let $E = \{e_1, e_2, e_3\}$ be a set of items and Π be an ordering on E, so that the items are ordered as listed in the definition of E. In this case,

$$\textsc{Encode}(\Pi, \{e_1, e_3\}) = \{1, 0, 1\} \text{ and } \textsc{Decode}(\Pi, 011) = \{e_2, e_3\}.$$

Analysis of Encode and Decode. The $\textsc{Encode}(\Pi, S)$ function sets some bits of the bit set V to 0 and some bits to 1, in order to return the encoded value of S. Therefore, the space complexity is $\Theta(N)$. For each element in S, we need to set $V[i] \leftarrow 1$, where i is the index of S in the ordering Π. The time complexity is $O(|S|)$.

The $\textsc{Decode}(\Pi, V)$ function creates a set S from the bit set V, which can be at most N. Therefore it takes $O(N)$ space. For each bit v_i set to 1 in V, the function fetches the element at position i in the array and adds it to the set. This procedure takes $O(|V|)$ time.

3 BLOOM MATRIX

This section introduces *Bloom Matrix* as the first effort to solve the membership checking problem with multiple sets. It consists of multiple columns of bit sets, in which each column represents an item, and the values of the corresponding bits are determined by the associated "labels", e.g., the set of unique words (as labels) in a document (as an item). Unlike a Bloom Filter, each bit in the Filter is replaced by another bit set of a fixed length, hence the name is Bloom Matrix.

3.1 Definition

A Bloom Matrix is a triplet $(\mathbf{G}, \Pi, H)$, where $\mathbf{G}$ is a binary matrix of size $m \times N$, Π represents an ordering on the set E as previously defined, and $H = \{h_1(l), ..., h_k(l)\}$ is a set of hash functions, each having image $[0, m-1]$.

3.2 Operations

Add. $\mathbf{G}$ is initialized with all its bits set to 0. To add a label l to a Bloom Matrix BM, the return value of $\textsc{Encode}(\Pi, f(l))$ is added to the rows in the bit matrix $\mathbf{G}$ having the indices equal to the hash neighborhood of l, using the bit-wise OR operator. The add operation can be formally defined as

$$BM.\textsc{Add}(l) := \mathbf{G}[h_i(l), \_] \leftarrow \mathbf{G}[h_i(l), \_] \vee \textsc{Encode}(\Pi, f(l)) \text{ for each } i \in [1, k].$$

Algorithm 1: Bloom Matrix: $\textsc{Add}(l)$ Operation

```
$V \leftarrow \textsc{Encode}(\Pi, f(l))$;
$H \leftarrow GetNeighbourhood(l, k, m)$;
for $i \leftarrow H$ do
    $\mathbf{G}[i, \_] \leftarrow \mathbf{G}[i, \_] \vee V$
end
```

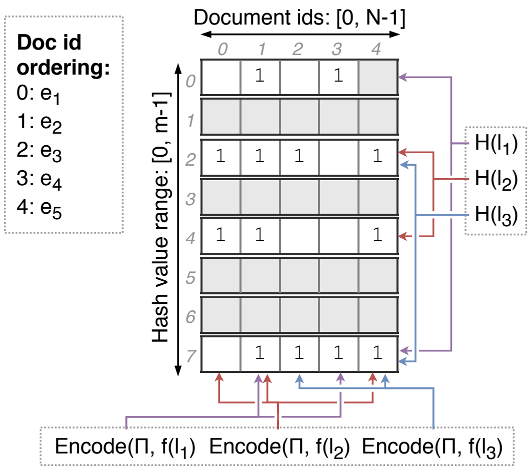

Fig. 1. Example of a Bloom matrix $\mathbf{G}[m, N]$ obtained from Example 3.1.

The steps are illustrated in Algorithm 1. The first step is to obtain a bit set V returned by Encode$(\Pi, f(l))$. The bits set to 1 at positions $V = \{v_1, ..., v_{|V|}\}$ represent the column indices of the Bloom Matrix. The row indices, on the other hand, are the output of the hash functions H. Finally, the Add(l) function calculates the Cartesian product of the row and column indices, i.e., $V \times H$, and sets all their corresponding bits to 1.

Algorithm 2: Bloom Matrix: Lookup(l) Operation

```
$H \leftarrow GetNeighbourhood(l, k, m)$;
Let $V$ be an empty bit set;
for $i \leftarrow H$ do
    $V \leftarrow V \wedge \mathbf{G}[i, \_]$;
end
return Decode $(\Pi, V)$;
```

Lookup. The lookup operation on a Bloom Matrix is illustrated in Algorithm 2. To find out which subset of $\mathcal{P}(E)$ is associated with l, the algorithm first performs bit-wise AND operations on the rows in $\mathbf{G}$ having the indices equal to the hash neighborhood of l. It then performs the Decode operation on the resulting bit set. The algorithm can be formalized as

$$BM.\text{Lookup}(l) := \text{Decode}\left(\Pi, \bigwedge_{1 \le i \le k} \mathbf{G}[h_i(l), \_]\right).$$

Example 3.1. This example illustrates the Add and Lookup operations on a Bloom Matrix. Given 3 labels $L = \{l_1, l_2, l_3\}$, 5 items $E = \{e_1, \cdots, e_5\}$, an ordering Π on E, and two hash functions $H = \{h_1(x), h_2(x)\}$ each with image $[0, 7]$, such that $H(l_1) = \{0, 7\}$, $H(l_2) = \{2, 4\}$ and $H(l_3) = \{2, 7\}$. The labels and items can be assumed as the words and documents, respectively.

Fig. 1 depicts a Bloom Matrix after adding three labels: $f(l_1) = \{e_2, e_4\}$, $f(l_2) = \{e_1, e_2, e_5\}$, and $f(l_3) = \{e_3, e_5\}$. Let us take l_1 as an example. When adding l_1 to a Bloom Matrix, the target rows are $\mathbf{G}[0, \_]$ and $\mathbf{G}[7, \_]$, given by $H(l_1) = \{0, 7\}$. Then, the Encode function identifies two target columns $\mathbf{G}[\_, 2]$ and $\mathbf{G}[\_, 4]$, as Encode returns $\{0, 1, 0, 1, 0\}$. Finally, the Add operation identifies the locations of four bits and sets these bits to 1: $\mathbf{G}[0, 2]$, $\mathbf{G}[7, 2]$, $\mathbf{G}[0, 4]$, and $\mathbf{G}[7, 4]$. l_2 and l_3 can be similarly added to the Bloom Matrix.

The Lookup(l_1) function first finds the hash neighborhood of l_1 and gets two target rows: $\mathbf{G}[0, \_]$ and $\mathbf{G}[7, \_]$. After that, the AND operation is performed on these rows to get a bit set $\{0, 1, 0, 1, 0\}$. Therefore, Decode returns $\{e_2, e_4\}$ according to the positions of 1 in the bit set. Lookup(l_2) and Lookup(l_3) work in the same way. Noteworthy, Lookup(l_3) performs the AND operation on $\mathbf{G}[2, \_]$ and $\mathbf{G}[7, \_]$, and outputs $\{e_2, e_3, e_5\}$, where e_2 is an FP.

Multi-label Lookup. Algorithm. 2 can be extended to support multi-label Lookup operations: Lookup(L), where L is the set of input labels. Specifically, instead of performing hash neighborhood on a single label l (Line 2 in Algorithm 2), H is now the union of all the hash neighborhoods of all the labels in L.

Update. A Bloom Matrix is a fixed structure for multiple sets. It is not possible to update with the labels of a new item; doing so requires reconstructing the Bloom Matrix. However, it is still possible to add a label to an existing item at the cost of an increasing FP rate.

3.3 False Positive Rate

Lemma 3.2. *Let $L = \{l_1, \cdots, l_{|L|}\}$ be a set of labels, and let $f(l)$ be a function that maps each label l to some items e of E. The average false positive rate of a Bloom Matrix when performing a multiple label lookup on L, i.e., Lookup(L), is*

$$\frac{\sum_{l\in L}\sum_{e\in E\setminus f(l)}(1-(1-\frac{1}{m})^{nk})^k}{|L|},$$

where k is the number of hash functions, m is the range of hash neighborhoods, n is the number of items added to the Bloom Matrix, and E is the universe of all the items.

Proof. Given k hash functions, the Lookup operation performs the AND operation on k rows of a Bloom Matrix, having indices determined by the values returned by the hash functions. A false positive in this case happens when the bit at column c, i.e., bits of a document id in Fig. 1, of all the k rows are set to 1, where c is a positive integer.

The Add operation for a label l sets the bits identified by the Encode function on the m rows selected by the hash functions. The bits being set follow a uniform probability distribution. Therefore, the probability that any single bit in a Bloom Matrix is not set to 1 by k hash functions during one Add operation is

$$\left(1-\frac{1}{m}\right)^k. \tag{3}$$

Assuming that the Add operation performs on $n = |f(l)|$ items, then the probability that a certain bit is still 0 equals to $(1-\frac{1}{m})^{nk}$. In contrast, the probability that the same bit is set to 1 is

$$1-\left(1-\frac{1}{m}\right)^{nk}. \tag{4}$$

Let us assume that the Lookup(l) operation returns a false positive item e_f for a label l. This happens when the bits located at the column corresponding to e_f and rows determined by $H(l)$ are set to 1. It happens with probability

$$\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)^k. \tag{5}$$

In other words, each item, returned by the LOOKUP(l) operation, has a probability of being a false positive which is equal to Equation 5. Since any member of $E \setminus f(l)$ can be a false positive, the rate of having at least one false positive can be calculated as

$$\sum_{e \in E \setminus f(l)} \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^{k}. \tag{6}$$

Furthermore, for a multiple label lookup in L, the overall false positive rate for looking up all the labels $l \in L$ can be obtained by summing up the probabilities for the remaining items that are not in $f(l)$. Henceforth, the average probability is

$$\overline{\text{FPR}} = \frac{\sum_{l \in L} \sum_{e \in E \setminus f(l)} (1 - (1 - \frac{1}{m})^{nk})^{k}}{|L|}. \tag{7}$$

□

3.4 Complexity

LEMMA 3.3. *The Bloom Matrix has a space complexity $\Theta(mN)$, add time complexity $\Theta(k|f(l)|)$, and lookup time complexity $\Theta(N)$, where m is the size of hash neighborhood, k is the number of hash functions, N is the total number of items, and $|f(l)|$ is the number of items assigned to l.*

PROOF. Space complexity: a Bloom Matrix stores three components: (i) the bit set having size $m \times N$, (ii) the total ordering Π of size $2N$, required by the ENCODE and DECODE functions, and (iii) k hash functions. The space cost is therefore $\Theta(mN) + 2\Theta(N) + \Theta(k) = \Theta(mN)$ when $k \ll mN$.

Time complexity: an ADD(l) operation on a Bloom Matrix computes the neighborhood of l in $\Theta(k)$ time to get the row indices, and executes ENCODE($\Pi, f(l)$) in $\Theta(|f(l)|)$ time to get the column indices of the bits to be updated. In total, $k|f(l)|$ bits are altered. Therefore, an ADD operation takes $\Theta(|f(l)| + k + k|f(l)|) = \Theta(k|f(l)|)$ time.

A LOOKUP(l) operation identifies k rows by executing k hash functions, each row having N bits. The algorithm then performs a bit-wise AND operation on these k rows and sends the result to the DECODE function to get a list of items. The time cost is $\Theta(k)$ for the hashing, $\Theta(N)$ for a bit-wise AND, and $\Theta(N)$ for the DECODE function. Hence, the overall time complexity is $\Theta(k + N + N) = \Theta(N)$. □

3.5 Sparse Bloom Matrix

A Bloom Matrix can be *sparse* where some trailing bits of some rows are set to zero; this allows a sparse storage method to reduce the space cost. Space spared this way is less than $m \times N$. It is possible to reduce the space cost further by carefully defining the total ordering so that the sparse rows of such Bloom Matrix have more zeros at the end, as the trailing zeros can be spared. We name a Bloom Matrix with such ordering a *Sparse Bloom Matrix.* The following example illustrates the efficiency of a Sparse Bloom Matrix over a standard Bloom Matrix:

Example 3.4. Let us consider the Bloom Matrix in Example 3.1. The Bloom Matrix (Fig. 1) has 21 spare bits (gray cells) according to the specified ordering. In contrast, if the items are ordered as $\{e_2, e_5, e_4, e_3, e_1\}$, a Sparse Bloom Matrix can be constructed as in Fig. 2, which spares 2 more bits thanks to more zeros at the end of each row.

A Sparse Bloom Matrix can be constructed by changing the total ordering Π such that the items e in set E are ordered in descending order according to their size, i.e., the number of assigned labels. This maximizes the probability of having more zeros at the end of each row. More formally, let $C(e)$ be the number of labels assigned to an item e and Π be the total ordering s.t. $\Pi(e_i) > \Pi(e_j)$ iff

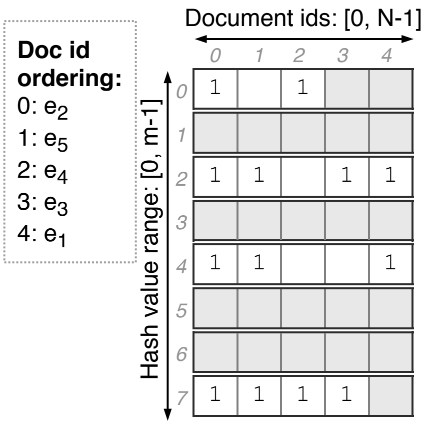

Fig. 2. A Sparse Bloom Matrix is constructed following Fig. 1 by changing the total ordering of the elements. The space cost reduces from 19 to 17 bits. The reduction is achieved by eliminating the storage of the gray cells at the end of each row.

$C(e_i) \leq C(e_j)$. Later in Section 5, we show that with such ordering, a Spare Bloom Matrix is more space-efficient than a standard Bloom Matrix.

Complexity. The time complexities of the Add and Lookup operations for Sparse Bloom Matrix are respectively $\Theta(k|f(l)|)$ and $\Theta(N)$, which are similar to the standard Bloom Matrix. Let $|\bar{e}|$ be the average number of occurrences of an item. The space complexity of a Sparse Bloom Matrix is bounded by $\Omega(m|\bar{e}|)$ and $O(mN)$, when the global ordering sorts all the bits set to 1 at the beginning of each row, and the last bit of each row is set to 1, respectively.

To construct a Sparse Bloom Matrix, we need to compute an ordering Π beforehand, and the time cost depends on the dataset representation. If the dataset is a collection of $|L|$ labels in the form of $l : \{e_1, e_2, \cdots\}$, then the cost is $\Theta(|L|N)$, as it is required to scan the whole structure and to keep a counter for each e. On the other hand, if the dataset is an array of N items in the form of $e : \{l_1, l_2, \cdots\}$, then the cost is $\Theta(N)$, as it is required to compute the size of each e.

4 BLOOM VECTOR

In a Bloom Matrix, all rows have the same size, m. In practice, the number of labels assigned to an item can vary, and thus, using a fixed size for all Filters is a waste of space. When the number of labels in the items is known beforehand, it is possible to utilize such knowledge to reduce the space cost further while maintaining the same FP rate. This section formalizes the idea as a new structure, called Bloom Vector.

4.1 Definition

A Bloom Vector is a tuple (G, Π), where G is an array of size $N = |E|$ in which each item corresponds to a Bloom Filter, and Π represents an ordering on the set E as previously defined. The Bloom Filters in G can be different from each other. Given an expected FP rate p for a Bloom Vector, the size of a Bloom Filter can be computed using Equation 2a, where n is the number of labels for that Bloom Filter. Note that all the Bloom Filters in the Bloom Vector have the same FP rate. The Bloom Filters can have the same or different hash functions (calculated from Equation 2b), as we discuss in detail in the upcoming subsection.

4.2 Operations

Let us now define the possible operations on a Bloom Vector, which are based on the operations on a standard Bloom Filter.

Algorithm 3: Bloom Vector: Add(l) Operation

```
$V \leftarrow$ Encode $(\Pi, f(l))$;
$I \leftarrow V.toList$;
for $i \leftarrow I$ do
    /* add operation on a standard Bloom Filter */
    $G[i].Add(l)$;
end
```

Add. The add operation on a Bloom Vector is illustrated in Algorithm 3. In order to add a label l associated with items $f(l)$, the first step is to compute Encode($\Pi, f(l)$) to get the row indices corresponding to the items in $f(l)$ according to the global ordering. Then, for each of these rows, a set of k hash functions determines the column indices of bits to be modified to 1. Formally, let I be the row indices returned by Encode($\Pi, f(l)$). The Add function is defined as:

$$BV.\,\textsc{Add}(l) := G[i].\,\textsc{Add}(l) \text{ for each } i \in I.$$

It is worth noticing that a Bloom Vector does not require having the same set of hash functions for all the rows because each row is a standalone Bloom Filter.

It is also possible to use one set of hash functions for all the Bloom Filters so that the hash operations are performed only once. This reduces the time cost of add and lookup operations on a Bloom Vector. In this case, the hash functions should return a real number within a specific range, so that this number can be regulated to suit any Bloom Filter. For example, let $\{h_1(l), \cdots, h_k(l)\}$ be k hash functions where each returns a real number within $[0, 1]$, and let m_i be the size of i-th Filter (row) in a Bloom Vector. Then the column indices in the ith Filter are located in $\{\lfloor h_1(l) \times m_i \rfloor, \cdots, \lfloor h_k(l) \times m_i \rfloor\}$. Another example: when using a 32-bit MurmurHash of range $[0, 2^{32}]$, the bits to be altered are located in columns $\left\{\lfloor h_1(l)/2^{32} \times m_i \rfloor, \cdots, \lfloor h_k(l)/2^{32} \times m_i \rfloor\right\}$. We implement a Bloom Vector class in Section 5, which uses the same hash functions for all the Bloom Filters.

Lookup. Algorithm 4 describes the Lookup operation on Bloom Vectors. To find a subset of E that is associated with l, the first step is to define V as a bit set, where each bit is

$$V[i] := \begin{cases} 1, & \text{if } G[i].\,\textsc{Lookup}(l) \\ 0, & \text{otherwise} \end{cases} \quad \text{where } 1 \le i \le N.$$

The Lookup operation then converts V back to a item set:

$$BV.\,\textsc{Lookup}(l) := \textsc{Decode}(\Pi, V).$$

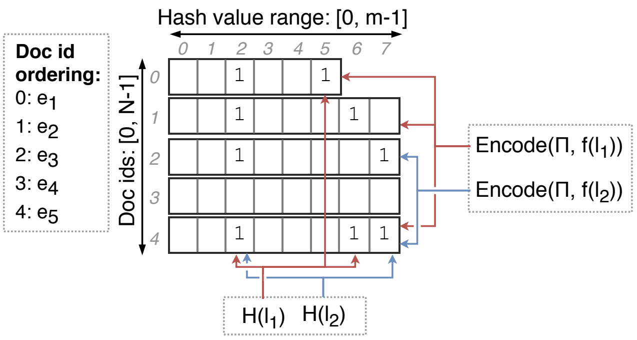

Fig. 3. A Bloom Vector, **G**$[N, \_]$, followed from Example 4.1. The size of each Bloom Filter in the Bloom Vector can be different.

Algorithm 4: Bloom Vector: LOOKUP(l) Operation

```
Let $V$ be an empty bit set;
for $i \leftarrow [0, N)$ do
    /* lookup operation on a standard Bloom Filter    */
    if $G[i].Lookup(l)$ then
        $V[i] \leftarrow 1$;
    end
end
return DECODE $(\Pi, V)$;
```

Example 4.1. Let $L = \{l_1, l_2\}$ be two labels, $E = \{e_1, \cdots, e_5\}$ be 5 items, Π be an ordering on the items, and $H = \{h_1, h_2\}$ be two hash functions returning two real numbers between 0 and 1. Assume that the regulated hash values are $\lfloor H(l_1) \times 6 \rfloor = \{2, 5\}$, $\lfloor H(l_1) \times 8 \rfloor = \{2, 6\}$, $\lfloor H(l_2) \times 6 \rfloor = \{2, 5\}$, and $\lfloor H(l_2) \times 8 \rfloor = \{2, 7\}$.

Now, a Bloom Vector can represent $f(l_1) = \{e_1, e_2, e_5\}$ and $f(l_2) = \{e_3, e_5\}$. As shown in Fig. 3, when adding l_1, the ENCODE function returns $\{1, 1, 0, 0, 1\}$ and hence the three target rows are $\mathbf{G}[1, \_]$, $\mathbf{G}[2, \_]$, and $\mathbf{G}[5, \_]$. Then, since $\lfloor H(l_1) \times 6 \rfloor = \{2, 5\}$ and $\lfloor H(l_1) \times 8 \rfloor = \{2, 6\}$, the first pair belongs to the 0th row, and the latter belongs to the 1st and 4th rows. Therefore, representing $f(l_1)$ needs to set six bits to 1: $\mathbf{G}[0, 2]$, $\mathbf{G}[0, 5]$, $\mathbf{G}[1, 2]$, $\mathbf{G}[1, 6]$, $\mathbf{G}[4, 2]$, and $\mathbf{G}[4, 6]$. l_2 can be added to the Bloom Vector in the same way.

To perform a LOOKUP(l_2), we first build an empty bit set V of length N and then scan through all the rows sequentially. Specifically, column 2 and 5 in the 0th row needs to be checked, since $\lfloor H(l_2) \times 6 \rfloor = \{2, 5\}$. Since both bits are already 1, we set the 1st bit of V to 1. For the remaining 4 rows, the columns are 2 and 7, as $\lfloor H(l_2) \times 8 \rfloor = \{2, 7\}$. Finally, V becomes $\{1, 0, 1, 0, 1\}$ and DECODE(V) outputs three items: $\{e_1, e_3, e_5\}$. Note that e_1 is an FP since it is not in $f(l_2)$ according to our empirical knowledge.

Multi-label Lookup. The same procedure in Algorithm 4 can lookup multiple labels at the same time, except that it needs to hash multiple labels before checking the corresponding bits simultaneously.

Update. Unlike Bloom Matrix, each Bloom Filter in a Bloom Vector has its own parameters. The ADD operation is performed on an individual filter. Therefore, it is possible to add new items incrementally to the Bloom Vector and so the corresponding labels. Updating an already existing Filter in a vector increases the FP rate.

4.3 False positive rate

Lemma 4.2. *Let $L = \{l_1, \cdots, l_{|L|}\}$ be a set of labels, where each l is associated with some items e. The average false positive rate of a Bloom Vector, when performing a Lookup for all the labels in L, is*

$$\frac{\sum_{l \in L} \sum_{i \in I(l)} \left[1 - \left(1 - \frac{1}{m_i}\right)^{k_i n_i}\right]^{k_i}}{|L|},$$

where $I(l)$ is a set of indices of rows with the corresponding bits set to 1, and k_i, m_i, and n_i are the number of hash functions, the number of bits, and the number of labels added to the ith Bloom Filter, respectively.

Proof. The Lookup operation on a Bloom Vector goes through each row to check whether the bits at the columns given by the k hash functions are all set to 1.

Assume a Bloom Filter with m bits, k hash functions, and it has n encoded labels. The probability of having a false positive item e_f when looking up a label l is (same as Equation 5)

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^{k}. \tag{8}$$

Let m_i and k_i be the parameters of the ith Bloom Filter in a Bloom Vector, and let n_i be the number of labels that the ith Bloom Filter contains. The total expected FP rate when looking up a label l can be derived from Equation 8 by summing up the FP rates of all the Bloom Filters as

$$\text{FPR} = \sum_{i \in I(l)} \left[1 - \left(1 - \frac{1}{m_i}\right)^{k_i n_i}\right]^{k_i}, \tag{9}$$

where $I(l)$ returns the indices of Bloom Filters modified by Add(l).

Finally, when looking up multiple labels L, the average FP rate is

$$\overline{\text{FPR}} = \frac{\sum_{l \in L} \sum_{i \in I(l)} \left[1 - \left(1 - \frac{1}{m_i}\right)^{k_i n_i}\right]^{k_i}}{|L|}. \tag{10}$$

□

4.4 Complexity

Compared to the Bloom Matrix, a Bloom Vector requires less space. However, the Lookup operation can be slower, as it has to check all the Bloom Filters in the Bloom Vector.

Lemma 4.3. *Bloom Vector has a space complexity of $O(mN)$, an Add time complexity of $O(k|f(l)|)$, and a Lookup time complexity of $\Theta(kN)$, where m is the max size of the hash neighborhood among all rows, k is the max number of hash functions for all rows, N is the total number of items, and $|f(l)|$ is the number of items in a specific Add operation.*

Proof. Space complexity: a Bloom Vector needs to store three parameters: (i) $m \times N$ bits, when all N rows have an equal length m (worst case), (ii) the total ordering Π, which uses $2N$ space, and (iii) kN hash functions. Therefore, the space cost is $O(mN) + 2\Theta(N) + \Theta(kN) = O(mN)$, when $k \ll m$. In practice, a Bloom Vector will be smaller than mN, as the rows can have different lengths.

Name	Origin	Number of items	Number of labels	File size
Uniform (500)	Artificial	500	2,500,000	12.2 MB
Uniform (10k)	Artificial	10,000	50,000,000	245 MB
Zipf	Artificial	500	30,000	171 kB
20ng	Real	7,527	625,635	3.9 MB

Table 2. Properties of the used datasets.

Time complexity: the Add(l) operation needs to compute the positions of bits by executing Encode(Π, $f(l)$), and to compute $O(k)$ hash neighborhoods when each Filter is using a different set of hash functions. Therefore, the total time complexity is $O(k|f(l)|)$, as the time for updating one bit is negligible.

A Lookup(l) operation looks up all of the N rows. For each row, it needs to calculate k hash neighborhoods. This requires $\Theta(kN)$ time. The Decode function has a time complexity of $O(N)$, as the size of the final bit set is N. The overall lookup time is therefore $\Theta(kN) + O(N) = \Theta(kN)$. □

5 PERFORMANCE EVALUATION

This section presents a detailed performance evaluation of the proposed BMFs. Section 5.1 presents the dataset properties and implementation details of BMFs. The subsequent sections investigate their space efficiency and compare the performance of Add/Lookup operations on them.

5.1 Setup

We implemented the BMFs in Scala and conducted experiments on a computer with a quad-core processor of 3.4 GHz clock speed, 16 GB of RAM, and 512 GB of SSD.

We use three artificial datasets and a real dataset, as described in Table 2. The artificial datasets are useful to experiment with the behavior of BMFs in a controlled environment. Specifically, we generate each dataset following either a uniform or a Zipf distribution. Each line in the dataset begins with an item id followed by the corresponding labels: $e = \{l_1, l_2, l_3 \cdots, l_n\}$.

The dataset generation procedures are as follows:

- Uniform: given E, L, and a probability p, for each $e \in E$ and for each $l \in L$, we randomly decide with a probability of $p = 0.5$ whether to assign such label l to e. Two uniform datasets are generated: **uniform (500)** of 500 items and **uniform (10k)** of 10,000 items.
- **Zipf**: given E, L and a real number s, we generate the first $|E|$ Zipf rank numbers with exponent value s and $N = |E|$ using following the equation

$$f(k; s, n) = \frac{1/k^s}{\sum_{i=1}^{N}(1/n^s)}, \tag{11}$$

  where $s = 0.8$. Then, for each $e \in E$ and for each $l \in L$, we decide with a probability equal to the rank k whether to assign l to e.
- **20ng**: we obtain the single-label text categorization dataset [6] and form **20 n**ews**g**roups corpus with the stemmed words. No a priori knowledge is available about the probability distribution of the labels among the items.

5.2 Implementation Details

Each BMF is implemented according to a model interface with 4 methods in Scala: an `add` method, which adds a label; two different `lookup` methods, one for the lookup of a single label and one for multiple labels; and a method for computing the size of the corresponding BMF.

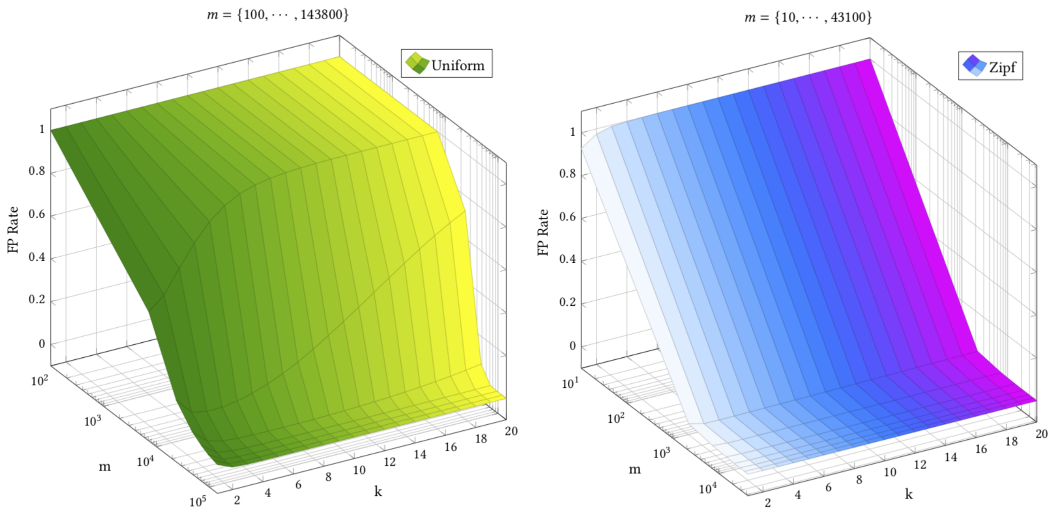

Fig. 4. The FP rates of the BMFs with respect to m, k and the distributions of the labels. All BM, SBM, and BV have similar characteristics.

The Bloom Matrix class employs 5 attributes, namely `E`, `m`, `k`, `F`, and `Em`. `E` is an array of all items; `m` is the height of the Bloom Matrix; `k` is the number of hash functions; `F` is an array of bit sets, each representing a row of the Bloom Matrix; and `Em` is a mapping from element ids to integers so that an ordering can be established on the items. This mapping is used by the ENCODE and DECODE functions.

The Bloom Matrix class has two constructors, accepting either (i) a quadruplet `(data,m,k,cmp)` or (ii) a triplet `(data,p,cmp)`. The first constructor implements a Bloom Matrix with the given m and k, whereas the second constructor derives m and k from Equations 2a and 2b, where n is the total number of labels associated to all the items. An Sparse Bloom Matrix is constructed by setting the Boolean variable, `cmp`, to be `true` in the constructor. Otherwise, a Bloom Matrix is constructed with the `false` value.

The Bloom Vector class has 6 attributes. `E`, `m`, and `k` represent the same attributes as in the Bloom Matrix; `F` contains an array of bit sets, each representing a single Bloom Filter; `M` contains an array of integers, each representing the size of one Bloom Filter in `F` in the same ordering; and `Em` is a map from element ids to integers, for assigning each item to an index, so that an ordering can be established on the items.

The Bloom Vector class also contains multiple constructors, accepting either (i) a triplet `(data,m,k)` or (ii) a pair `(data,p)`. The first constructor implements a Bloom Vector of $|E|$ Bloom Filters each with the given m and k, and the second constructor derives m and k given p for every Bloom Filter from Equations 2a and 2b, where n is the number of labels associated with the corresponding item.

5.3 Bloom Multifilters and standard Bloom Filter

In the first set of experiments, we construct Bloom Matrix (BM), Sparse Bloom Matrix (SBM), and Bloom Vector (BV) objects with various combinations of m and k using the first constructor type of the corresponding classes. We investigate whether their FP rates adhere to the principles of a standard Bloom Filter.

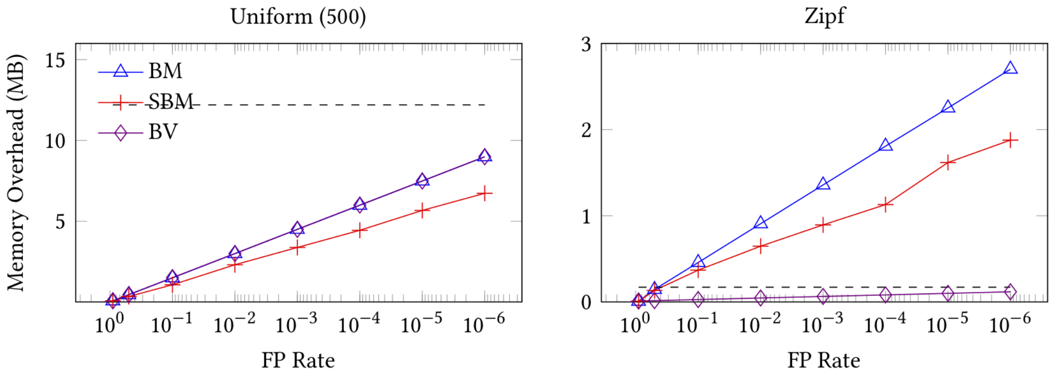

Fig. 5. Memory usage of the BMSs according to the FP rates and the distribution of labels. The horizontal dashed line represents memory usage on the disk.

Fig. 4 depicts that when m is between 10^4 and 10^5, the FP rate decreases and then increases again as k becomes larger. This trend is expected according to the theoretical analysis in the earlier sections, as two labels are more likely to have the same hash neighborhoods with too few hash functions. On the contrary, too many hash functions may set too many bits to 1, increasing the false positive rate. In other words, either of the two scenarios leads to more collisions. All the BMs, SBMs, and BVs perform similarly in this regard.

Fig. 4 also shows that the FP rate decreases as m grows larger. Again, such a trend confirms the theoretical analysis that too small m leads to a higher FP rate.

5.4 Performance of Bloom Multifilters

In this section, we investigate the space efficiency of the BMFs, and the time required to perform the Add and Lookup operations on these structures according to the expected FP rates on the same computer. The BMs, SBMs, and BVs are constructed with the second constructor type accepting the desired FP rate.

Memory Overhead. Fig. 5 demonstrates the memory overhead of the structures as the FP rate decreases. We notice that all the BMFs are well below the horizontal dashed line when representing the dataset following a uniform distribution. A Bloom Vector occupies a similar space to the Bloom Matrix because all the corresponding Filters have similar sizes when representing a uniformly distributed dataset. Meanwhile, the Sparse Bloom Matrix occupies a slightly smaller space, because it sets the ordering on the set E to maximize the number of zeros at the end of each row, which makes it the most space-efficient structure for a uniformly distributed data.

In contrast, with the Zipf distributed dataset, both the Bloom Matrix and the Sparse Bloom Matrix have a high space consumption. Although the Sparse Bloom Matrix performs better than the Bloom Matrix, both are above the dashed line when the desired FP rate is very low. In this case, the Bloom Vector is the most space-efficient, and the size remains below the dashed line even with an accepted FP rate of 10^{-6}, as it optimizes the size of each row to minimize the space consumption. With such a property, a Bloom Vector occupies at least 10 times less space than the other two structures.

Add Time. Fig. 6 illustrates that the Add operation takes nearly a linear time on a Bloom Matrix with the uniformly distributed dataset. This is expected, as an Add operation costs $O(k|l|)$. The Add operation time on a Bloom Vector also increases linearly with the Zipf distributed dataset. The Add operation takes a longer time on a Vector than on a Bloom Matrix; regardless of the distribution of labels in the datasets. The reason is that even though the Bloom Vector uses the

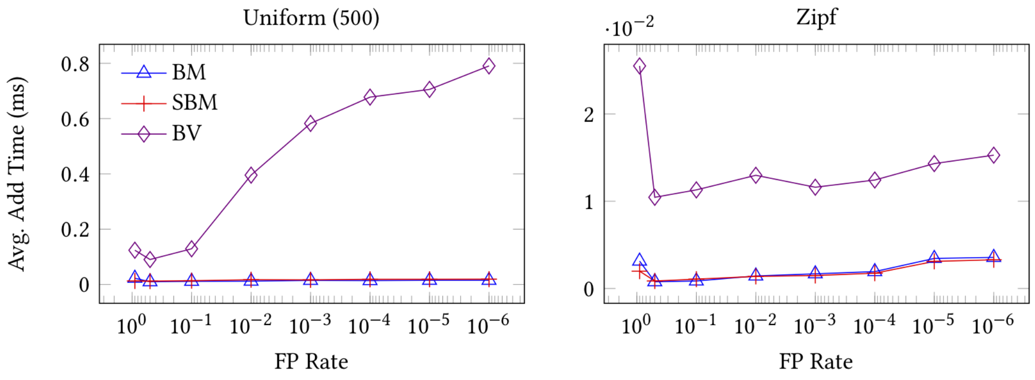

Fig. 6. Time required to add a label to the BMFs by the FP rates.

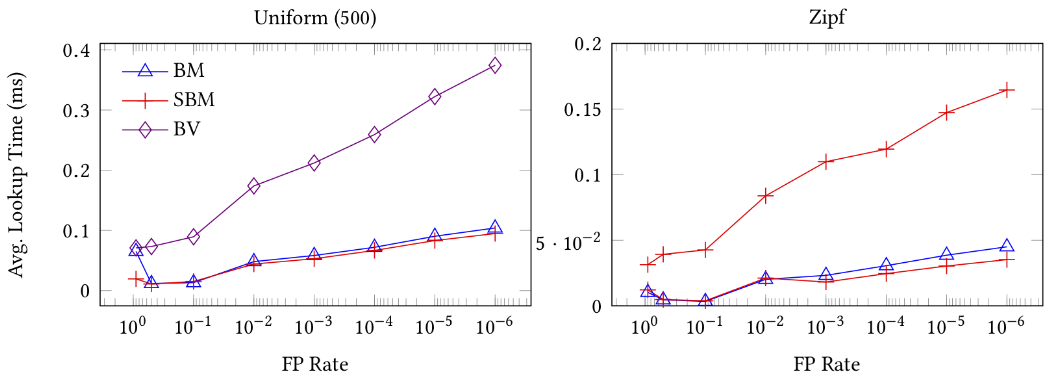

Fig. 7. Lookup time of a single label from the BMFs with different FP rates.

same hash functions for all the Bloom Filters, the Add operation still has to do a modulo operation on the results for each Filter, to adapt the results of the hash functions to its size.

Single Label Lookup. Fig. 7 depicts the single label Lookup performance of the Multifilters. The Lookup operation is performed on the structures having 10K unique inserted labels, and we compute the average lookup time. Both the Bloom Matrix and the Sparse Bloom Matrix have a similar lookup time, which is faster than the Bloom Vector. This is because the Lookup operation accesses only a few (specifically, k) rows in a Bloom Matrix, whereas, it goes through all N Filters in a Bloom Vector, ending up with an increased workload.

Fig. 8 demonstrates the performance of the BMFs with a real dataset, 20ng, where the labels follow an unknown distribution among 7,527 items. Similarly, as with the Zipf distributed dataset, the Bloom Vector efficiently exploits the size of the items, contrary to the other BMFs. As expected, the Lookup operations on the BMs/SBMs are faster than those on the BVs.

Multi-Label Lookup. The proposed BMFs support lookup queries having multiple labels. Such queries are useful when searching for similar documents. Fig. 9 compares the lookup time of multiple labels on the BMFs representing uniform and Zipf distributed datasets. We take the average of 10K lookup operations.

Multi-label lookup operations are also faster on a Bloom Matrix than on a Bloom Vector, and the size of the query (number of labels) has a negligible effect when the query size is at most 10 labels.

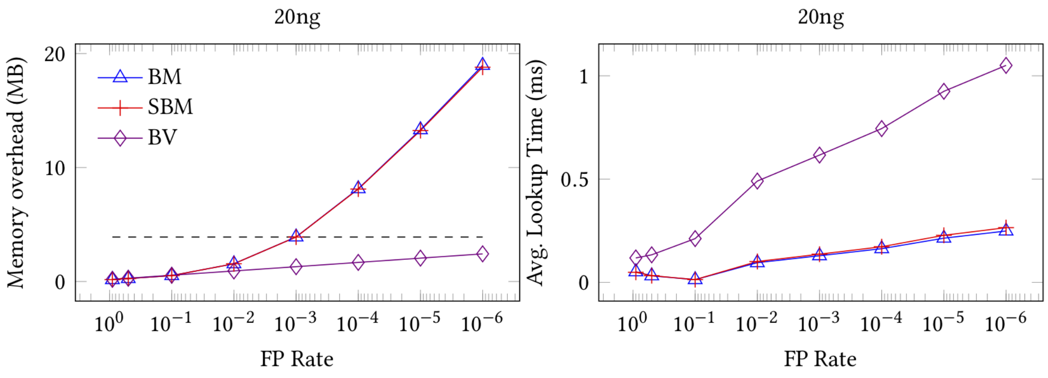

Fig. 8. Memory overhead of the BMFs (left) and the average lookup times of a single label (right) for the 20ng dataset according to the FP rates.

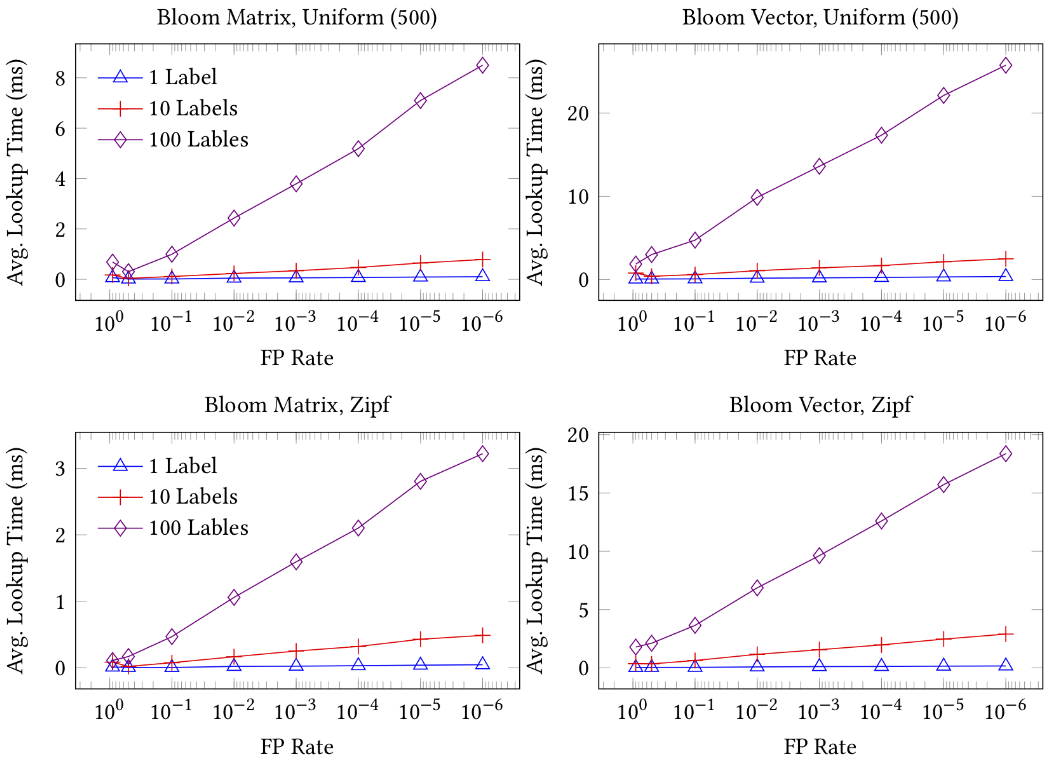

Fig. 9. The average time for multi-label lookups according to the FP rates. The average time for each set of lookup labels is computed from 10,000 lookup operations.

This is because the lookup function needs to compute the hash neighborhood of all the labels, which is the only additional task. The range of hash neighborhood increases further as the FP rate decreases. Specifically, looking up 100 labels requires a much longer time compared to 10 labels. After a careful investigation, we conclude that it is the increased labels that result in increased hash neighborhoods so that the algorithm has to access more bits. A Sparse Bloom Matrix would

Feature	Bloom Matrix	Bloom Vector	Bloofi
Index	Bit set	Bit set	Tree
Size of Each Filter	m	$\leq m$	m
Index Size	N	N	$\leq (N-1) \times m$
Insert Complexity	$k\|f(l)\|$	$k\|f(l)\|$	$k\|f(l)\| \log N$
Lookup Complexity	N	kN	kN

Table 3. Comparison of BMFs and Bloofi. This table assumes that the Bloofi index has an order of 2. $|l|$ is the number of items to be added into the structures.

perform similar to the Bloom Matrix. In the case of a Bloom Vector, the lookup is much slower, as the operation checks all the Bloom Filters.

Note that the Lookup operation works on a complete set of query labels. In other words, all the labels are searched together by an AND operation. Performing an OR operation on any BMF for subset queries provides invalid indexes of the items during the Decode (Π, V) operation. Finding the match for such a subset of the queries, it would first require to look for the items of each label and then perform an OR after the Decode (Π, V) operation.

5.5 Comparison with Bloofi

In this section, we compare the performance of the proposed BMFs with a state-of-the-art structure named Bloofi[1] [10], which uses a tree structure to index multiple Bloom Filters.

Table 3 highlights the differences between the BMFs and Bloofi. Both Bloom Matrix and Bloofi consist of Bloom Filters of m bits, and use the same set of hash functions. In contrast, the Bloom Filters in a Bloom Vector can have different sizes and hash functions. The BMFs use a global ordering of N, whereas the Bloofi employs a tree. An Add(l) operation on a Bloofi is will alter $|f(l)|$ Bloom Filters in leaf nodes, and all Filters in corresponding internal nodes, i.e., $|f(l)| \log N$ nodes, in the worst case. This yields a complexity $O(k(|f(l)| + |f(l)| \log N)) = O(k|f(l)| \log N)$. For Lookup$(l)$, the best case is when there is only one path to follow towards one leaf. In such a case, the algorithm needs to scan all Filters in the leaf, which requires $k(\log N + 4)$ time since there are $\log N$ internal Filters and 4 leaf Filters when Bloofi has an order of 2. In the worst case, it scans all the Bloom Filters, and thus, complexity is $O(kN)$ [10].

The differences among these structures are explored by using a more substantial dataset as follows.

Dataset. This set of experiments uses another uniform (10K) dataset, which consists of 10K items of 50M labels in which 30K labels are unique. Representing such large datasets with BMFs and Bloofi further emphasizes their performance differences in Big data perspective.

Memory Overhead. Fig. 10 shows the memory usage of all the structures. These structures require more memory space since the decreasing FP rates lead to longer bit sets for the Bloom Filters. Specifically, Bloofi requires the most space due to the additional index nodes: when given 10,000 items or documents, Bloofi creates about 5,300 index nodes that result in 15,300 Bloom Filters. In contrast, the BMFs require less memory because each has only 10,000 Bloom Filters and no other Filters for indexing. When the desired FP rate is close to 1, the BMFs and Bloofi occupy a similar amount of space. As the FP rate decreases, Bloofi becomes more and more space inefficient, as the size of the Bloom Filter at an index node also increases.

[1]Source code is obtained from https://github.com/lemire/bloofi.

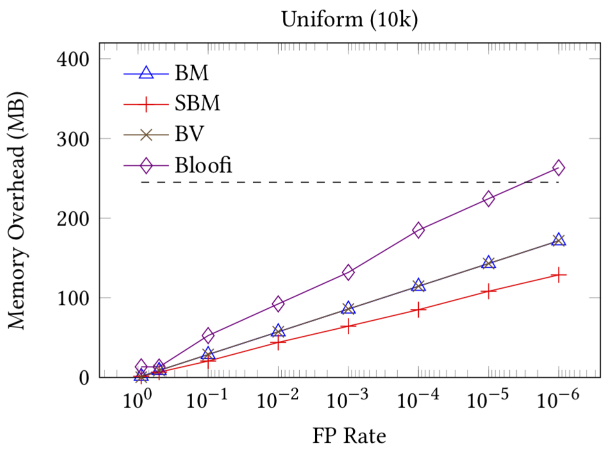

Fig. 10. Memory overhead of the BMFs and Bloofi according to the desired FP rates. The horizontal dashed line represents the actual data size on the disk (245 MB).

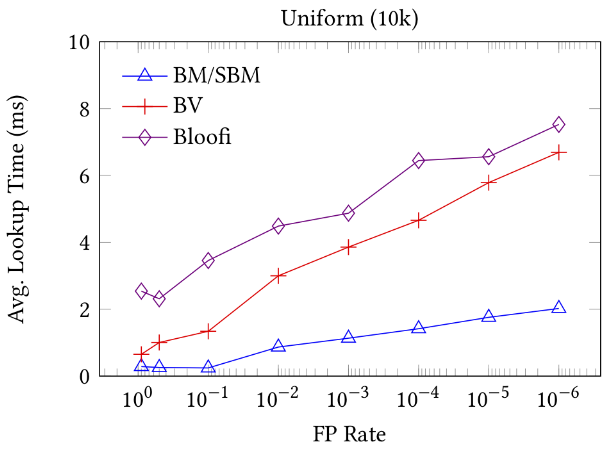

Fig. 11. Lookup time of a single label on BMFs and Bloofi.

Lookup Time. Fig. 11 summarizes the lookup time of a single label for all the structures. We look up 10K already inserted labels sequentially and compute the average lookup time. As we can see, the Bloom Matrix offers the fastest lookup time, followed by the Bloom Vector and Bloofi. This result is expected since each lookup operation on Bloom Matrix accesses only k rows, as shown in Fig. 1, whereas the other two access more Filters: the Bloom Vector performs an AND operations on all rows; Bloofi checks multiple Bloom Filters in the index structure as well as some Filters in the leaf nodes.

5.6 Discussion

Both Bloom Matrix and Vector obey the principles of standard Bloom Filter. Their space efficiency depends on the distribution of the labels in the input dataset. Bloom Vector is the most space-efficient with Zipf distributed data. The performance of the Add and Lookup operations depends on the implementation of the BMFs. The operations are faster on Bloom Matrix. Compared to the state-of-the-art, Bloofi, both Bloom Matrix and Bloom Vector are more space-efficient. The LOOKUP operation is faster on Bloom Matrix than on Bloom Vector and Bloofi.

6 RELATED WORK

There are many extensions of the Bloom Filter. Standard Bloom Filter and the variants were surveyed in [20][27]. In this section, we discuss the most relevant Bloom Filter variants.

Counting Bloom Filter. The Counting Bloom Filter [12] is a variant of the standard Bloom Filter. It was introduced to summarize and maintain proxy caches. A proxy maintains an integer array for the bit positions of a Bloom Filter. The add operation increments the integers to which the hash functions map a label and sets the corresponding bits of the Bloom Filter to 1. The delete operation decrements those integers and sets the corresponding bits of the Bloom Filter to 0. The lookup function checks whether all the integers mapped to a label are higher than 0.

Stateful Bloom Filter. The Stateful Bloom Filter [3] also extends the standard Bloom Filter. The stored elements are neither bits nor counters. Instead, each element of m represents a value corresponding to a state, i.e., state and state counter, for identifying P2P traffic and congestion control for video traffic.

Split Bloom Filter. The Split Bloom Filter [7] splits a bit set into multiple bins. Each bin has an associated hash function, and the hash functions are all different from each other. The corresponding add operation inserts an element into all bins. More formally, a Split Bloom Filter is composed by k bins $G = \{B_1, ..., B_k\}$ each of size m, where k is also the number of hash functions. Each hash function $h_i(x)$ relates to the bit set B_i having the same index. To insert an element x, the add operation sets $B_i[h_i(x)]$, for $1 \leq i \leq k$ bits to 1. Similarly, the lookup operation checks whether all those bits are set to 1.

Scalable/Dynamic Bloom Filter. A Scalable Bloom Filter [1] starts with a Split Bloom Filter. With k_0 bin and P_0 expected FP rate, a bin supports at most some elements that keep the FP rate below P_0. When the bin gets full, another one is added with k_1 bins and $P_1 = P_0 r$ expected FP rate, where r is a tightening ratio decided during the implementation. Scalable Bloom Filter is useful when the number of labels in a set is unknown. Alternatively, dynamic Bloom Filters [14] can be used if the size of the data is expected to grow with time.

Bloomier Filter. The standard Bloom Filter can encode only Boolean function. The Bloomier Filter [8] was proposed to represent arbitrary function on finite sets.

Let $E = \{e_1, ..., e_N\}$ and $R = \{1, ..., |R| - 1\}$. Let $A = \{(e_1, v_1), ..., (e_N, v_N)\}$ be an assignment, where $v_i \in R$ for $1 \leq i \leq N$. The encoding of such an assignment can also be seen as a function $f : E \rightarrow R$ defined as:

$$f(x) = \begin{cases} v_i, & \text{if } x \in E \\ \varnothing, & \text{otherwise} \end{cases} .$$

The Bloomier Filter uses a bit matrix to encode the function previously defined. To build such a matrix, it also uses ENCODE and DECODE.

Bloom Multifilter. The Bloom Multifilter proposed by Xu et al. [28] is close to this work, which extends the standard Bloom Filter to check whether multiple elements exist in multiple sets or not at once with yes/no answer.

Let $\mathbb{S} = \{S_1, ..., S_n\}$, where each S_i is a set of multiple elements. To check whether an S exists, i.e., $\exists S \in \mathbb{S}$, which contains all the elements in a query q, it is required to implement a Boolean function f defined as:

$$f(q) = \begin{cases} 1, & \exists S \in \mathbb{S} : q \subseteq S \\ 0, & \text{otherwise} \end{cases} .$$

Bloofi. Bloofi [10] can represent multiple sets and the parent-child relations among the elements. It uses a tree structure to index multiple Bloom Filters, and it is the closest to Bloom Matrix. Specifically, all the Bloom Filters are at the bottom of the tree as leaves, and each of the other nodes is an index node that contains a bit set having values obtained by performing a bit-wise OR operation on all its descendants. When looking up for a given label, the algorithm first applies the hash functions to the label, then it looks up each index node for the corresponding bits and thereby chooses the right path in the tree. At the last index node, it sequentially scans all the Bloom Filters in the leaf nodes to find the correct Filters. The BMFs presented in this work also represent multiple sets and more efficient compared to Bloofi.

7 CONCLUSIONS

This article presents two statistical data structures which not only answers whether labels belong to multiple sets, but also returns the associated sets. These proposed structures are space-efficient and thus can be cached in RAM to provide faster access compared to disk-stored structures or even Bloofi. With uniformly distributed labels among the sets, the variant of Bloom Matrix, i.e., Sparse Bloom Matrix, is more space-efficient at the expense of a reordering cost. On the other hand, Bloom Vector is the most space-efficient structure when representing a Zipf distributed dataset.

ACKNOWLEDGEMENTS

This work was in part supported by the Academy of Finland grant number 314008 and the Business Finland 5G-FORCE research project. We would also like to thank Dr. Antonio Cano, who worked on early Bloom Multifilter designs while visiting the University of Helsinki.

REFERENCES

[1] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

[2] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, July 1970.

[3] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese. Beyond bloom filters: From approximate membership checks to approximate state machines. *SIGCOMM Comput. Commun. Rev.*, 36(4):315–326, Aug. 2006.

[4] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang. On the false-positive rate of bloom filters. *Information Processing Letters*, 108(4):210–213, 2008.

[5] S. Brin and L. Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 56(18):3825–3833, 2012. The {WEB} we live in.

[6] A. Cardoso-Cachopo. Improving Methods for Single-label Text Categorization. PhD Thesis, Instituto Superior Tecnico, Universidade Tecnica de Lisboa, 2007.

[7] F. Chang, W. chang Feng, and K. Li. Approximate caches for packet classification. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2196–2207, March 2004.

[8] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 30–39, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[9] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 241–252, New York, NY, USA, 2003. ACM.

[10] A. Crainiceanu and D. Lemire. Bloofi: Multidimensional bloom filters. *Inf. Syst.*, 54:311–324, 2015.

[11] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. Planetp: using gossiping to build content addressable peer-to-peer information sharing communities. In *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on*, pages 236–246, June 2003.

[12] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, Jun 2000.

[13] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. Youtube traffic characterization: A view from the edge. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 15–28, New York, NY, USA, 2007. ACM.

[14] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo. The dynamic bloom filters. *IEEE Transactions on Knowledge and Data Engineering*, 22(1):120–133, Jan 2010.

[15] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless Computing: One Step Forward, Two Steps Back. In *9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[16] K. G. Kakoulis and I. G. Tollis. Algorithms for the multiple label placement problem. *Computational Geometry*, 35(3):143–161, 2006.

[17] K. G. Kakoulis and I. G. Tollis. Labeling algorithms. pages 489–515, 2013.

[18] D. K. Krishnappa, M. Zink, C. Griwodz, and P. Halvorsen. Cache-centric video recommendation: An approach to improve the efficiency of youtube caches. *ACM Trans. Multimedia Comput. Commun. Appl.*, 11(4):48:1–48:20, June 2015.

[19] A. Kumar, J. Xu, and J. Wang. Space-code bloom filter for efficient per-flow traffic measurement. *IEEE Journal on Selected Areas in Communications*, 24(12):2327–2339, Dec 2006.

[20] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo. Optimizing bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials*, 21:1912–1949, 2018.

[21] Y. W. Park, K. H. Baek, and K. D. Chung. Reducing network traffic using two-layered cache servers for continuous media data on the internet. In *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*, pages 389–394, 2000.

[22] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, Apr. 1989.

[23] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 21–40, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[24] S. C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1248–1257 vol.3, 2002.

[25] B. A. Shirazi, K. M. Kavi, and A. R. Hurson, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[26] J. Tapolcai, J. Bíró, P. Babarczi, A. Gulyás, Z. Heszberger, and D. Trossen. Optimal false-positive-free bloom filter design for scalable multicast forwarding. *IEEE/ACM Trans. Netw.*, 23(6):1832–1845, Dec. 2015.

[27] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys Tutorials*, 14(1):131–155, First 2012.

[28] C. Xu, Q. Liu, and W. Rao. Bmf: An indexing structure to support multi-element check. In B. Cui, N. Zhang, J. Xu, X. Lian, and D. Liu, editors, *Web-Age Information Management*, pages 441–453, Cham, 2016. Springer International Publishing.

[29] T. Yang, A. X. Liu, M. Shahzad, Y. Zhong, Q. Fu, Z. Li, G. Xie, and X. Li. A shifting bloom filter framework for set queries. *Proc. VLDB Endow.*, 9(5):408–419, Jan. 2016.