# Magic Cube Bloom Filter: Answering Membership Queries for Multiple Sets

Zhouyi Sun*†, Siang Gao*, Bingqing Liu‡, Yufei Wang*, Tong Yang*†, Bin Cui*†
*Department of Computer Science, Peking University, China
†National Engineering Laboratory for Big Data Analysis Technology and Application (PKU), China
‡International School, Beijing University of Posts and Telecommunications, China

***Abstract*—Given $d$ sets without intersection, the *multi-set membership query problem* is to decide which set an incoming item $e$ belongs to. Multi-set membership query is a fundamental problem in a variety of fields of computer science, especially computer networking. Although several approaches have been proposed in the literature to address this problem, they cannot achieve a high processing speed, a high accuracy, and a small memory usage at the same time. In this paper, we propose a novel data structure, namely Magic Cube Bloom Filter (MC-BF), which outperforms the state-of-the-art in terms of accuracy and query speed with a limited memory usage. The *key idea* of the Magic Cube Bloom filter is that items from the same set are stored in different Bloom filters, improving the accuracy by the redistribution of items. The MC-BF also improves the query speed by utilizing spatial locality. Experimental results show that the MC-BF outperforms the state-of-the-art BUFFALO by up to 148.5 times in terms of accuracy, and up to 3.16 times in terms of query speed. We have made the source code of our MC-BF available on Github [1].**

## I. Introduction

### A. Background and Motivation

Given $d$ sets and any two of them have no intersection, a *multi-set membership query problem* is to answer whether an item $e$ belongs to this multi-set, and if so, which one of them. A *multi-set* is a collection of sets, and in each set an item can appear only once. The multi-set membership query is a crucial problem in computer applications and systems, such as network packets processing [2] [3], deep packet inspection [4], network traffic measurement [5], [6] and more [7], [8].

Multi-set membership query plays an important role in big data processing. Next we show two typical scenarios.

**Scenario 1:** Designing the MAC address table of a switch. There is a MAC address table in every switch. When forwarding a packet, a switch first uses a specific field (destination MAC address) in the arriving packet's header as the key of this packet, and then queries the key in the MAC table. The MAC table entries indicate the destination to which this packet should be forwarded. Regarding the keys of packets as the items and the forwarding table as the multi-set, the designing of the MAC table can be considered as solving a multi-set membership query problem. However, the MAC table of a switch located in an enterprise or a data center network may contain tens of thousands of entries, which disables straightforward hash solutions whose memory usage is saved by the linear relationship between that and the number of entries. We have to reduce the memory usage of the MAC table, because the fast memory size (SRAM or Block RAM in FPGA, or caches in CPU or GPU) of the switch is limited and too large a size will cause our data structure to be stored in the slow memory (DRAM).

**Scenario 2:** Distributed web caching. The most classic solution for distributed web caching is the Summary Cache [9]. There are multiple web servers providing the same services, and each server keeps a compact summary (a probabilistic data structure) of the content of each of other servers. When a server miss occurs, the server first checks all the summaries to see whether the requested content is contained in other servers, and then sends a query request only to those servers whose summaries exhibit positive results. This is a typical problem of multi-set query. Due to the importance of distributed caching, recent works [10], [11] are still managing to enhance the performance of web caching.

One straightforward solution is to use hash tables. However, for a large volume of data, using a hash table is not the best choice due to the memory inefficiency and hash collisions. One classic method of multi-set query is to rely on Bloom filters. Bloom filter (BF) [12], a space-efficient probabilistic data structure, could be used to solve multi-set membership query problems with relatively high speed and a certain error rate. However, when sizes of sets vary drastically, the conventional method may lead to a waste of memory or a huge loss in terms of accuracy. In a word, to handle multi-set membership query problems, a new data structure which has a small memory usage, a considerably high efficiency and a much lower error rate is strongly demanded.

The `design goal` of this paper is to propose a novel data structure to handle multi-set membership queries with a small usage of memory, a high query speed, and a much higher accuracy, especially when sizes of sets vary drastically.

### B. Limitations of Prior Art

BUFFALO [13] uses multiple Bloom filters to solve the problem of multi-set membership query. It allocates a Bloom filter for each set, and to insert an incoming item $e$ from set $i$, it inserts $e$ into the corresponding Bloom filter. When querying an item $e'$, it queries $e'$ in each Bloom filter and if only one Bloom filter reports that it has $e'$, it returns the corresponding set index as the result. Otherwise, it returns `error` as the result. The limitations of BUFFALO are twofold. On the one hand, since we cannot acquire the size of each set beforehand,

it is impossible to find a proper size for each Bloom filter. Therefore, a Bloom filter may be "too large" or "too small" for a given set, which means that after inserting all the items from that set into the Bloom filter, the number of 1s in the Bloom filter may be too small, indicating a waste of memory, or too large, indicating too many collisions and a drastic drop in accuracy. On the other hand, we cannot acquire the number of sets beforehand, instead, we have a rough estimation about the number (*e.g.*, no bigger than 64). Therefore, we have to either allocate a fixed number of Bloom filters and use several parts of them, which will result in a waste of memory, or dynamically allocate new Bloom filters during insertions, which will make the total memory usage unpredictable.

### C. Proposed Approach

To address the limitations of BUFFALO, we propose the Magic Cube Bloom filter. The `key idea` of the Magic Cube Bloom filter is that *items from the same set can be stored in different Bloom filters, improving the accuracy by redistributing items*. An item is "rotated" to a random place during its insertion, and rotated back when it is queried, which is like to shuffle and restore a magic cube. The Magic Cube Bloom filter consists of $d$ arrays, each of which has $w$ bits. Notice that in the first two versions of our Magic Cube Bloom filter, $d$ is equal to the number of sets to be inserted.

The first version of the Magic Cube Bloom filter, namely MC-BF$_1$, is associated with $k$ hash functions, $h_1(.), h_2(.), ..., h_k(.)$, whose outputs are uniformly distributed in the range $[1, w]$. To insert an item $e$ from set $i$, it computes $k$ hash functions and set $A_i[h_1(e)], A_i[h_2(e)], ..., A_i[h_k(e)]$ to 1. To query an item $e$, for each of the $d$ sets, we denote set $i$ as the *candidate set* if $A_i[h_1(e)], A_i[h_2(e)], ..., A_i[h_k(e)]$ are all set to 1. In the end, 1) if there is no candidate set, it returns `none`; 2) if there is only one candidate set, it returns the set index; 3) if there are more than one candidate sets, it returns `error`.

However, all the $d$ arrays in the MC-BF$_1$ have fixed and equal widths, which makes the MC-BF$_1$ unable to adapt to variable sizes of sets. To address this limitation, MC-BF$_2$ adds an extra hash function $g(.)$ to compute an offset for each item. Specifically, to insert an item $e$ from set $i$, it computes $h_1(e), h_2(e), ..., h_k(e)$ and $g(e)$, and set $A_t[h_1(e)], A_t[h_2(e)], ..., A_t[h_k(e)]$ to 1, where $t = 1 + (i + g(e))\%d$. When querying an item $e$, for each of the $d$ sets, we denote set $i$ as the candidate set if $A_t[h_1(e)], A_t[h_2(e)], ..., A_t[h_k(e)]$ are all set to 1.

The limitation of the MC-BF$_2$ is that a fixed $d$ means the number of sets need to be acquired beforehand, which is too strong a requirement to be fulfilled in many scenarios. MC-BF$_3$ makes three improvements on MC-BF$_2$: 1) It sets $d$ to 64, ensuring that it can record any multi-sets whose sizes do not exceed 64. 2) It utilizes spacial locality to accelerate the query operation. 3) It uses $k$ hash functions $g_1(.), g_2(.), ..., g_k(.)$ instead of a single hash function $g(.)$ to compute the $k$ offsets, increasing the randomness and accuracy. To insert an item $e$ from set $i$, it computes $h_1(e), h_2(e), ..., h_k(e)$ and $g_1(e), g_2(e), ..., g_k(e)$, and set $A_{t_1}[h_1(e)], A_{t_2}[h_2(e)], ..., A_{t_k}[h_k(e)]$ to 1, where $t_j = 1 + (i + g_j(e))\%64$. When querying an item $e$, for each of the 64 sets, we denote set $i$ as the candidate set if $A_{t_1}[h_1(e)], A_{t_2}[h_2(e)], ..., A_{t_k}[h_k(e)]$ are all 1.

### D. Key Contributions

- We propose a novel data structure, namely Magic Cube Bloom filter, which has a much higher accuracy and query speed when dealing with multi-set membership queries with a limited memory size.
- We have made mathematical analysis and carried out extensive experiments on the Magic Cube Bloom filter. Both the theoretical and the experimental results show that our Magic Cube Bloom filter significantly outperforms the state-of-the-art in terms of accuracy and query speed.

## II. Related Work

Typical algorithms for multi-set membership query include BUFFALO [13], perfect hashing [14], Bloomtree [15], Bloomier [16], Summary Cache [9], Coded BF [5], Combinatorial BF [17], iSet [18], and more [19]–[21]. These algorithms are largely based on Bloom filter.

The standard Bloom filter [12] is used to tell whether an item belongs to a set or not. A Bloom filter is an array consisting of $w$ bits and is associated with $k$ independent hash functions $h_1(.), h_2(.), ..., h_k(.)$, whose outputs are uniformly distributed in the range $[1, w]$. We denote the $i$-th bit of the array with $A[i]$. To insert an item $e$, the Bloom filter computes $k$ hash functions and set $A[h_1(e)], A[h_2(e)], ..., A[h_k(e)]$ to 1. To query an item $e'$, it checks whether $A[h_1(e')], A[h_2(e')], ..., A[h_k(e')]$ are all 1. If the $k$ positions are all 1, then it reports that $e$ is in the set; otherwise, it reports that $e$ is not in the set. Obviously, BF never reports $e \notin S$ if $e$ actually belongs to $S$, *i.e.*, it has no false negatives. However, BF may report $e \in S$ when $e$ actually does not belong to $S$ sometimes, *i.e.*, it has false positives. The false positive rate of BF is often small enough to be accepted in practical scenarios. For muli-set query, BUFFALO [13] assigns $d$ Bloom filters, corresponding to d sets respectively. To query an item $e$, the BUFFALO queries it in all the $d$ Bloom filters. If the $i$-th Bloom filter reports true, then BUFFALO reports that $e$ belongs to the $i$-th set. As a result, when sizes of different sets vary drastically, we have to allocate enough memory to hold the biggest one, which leads to huge amount of memory waste upon relatively smaller ones. This is intolerable as we hope the data structure could be small enough to be stored in SRAM (static RAM), rather than DRAM (dynamic RAM), which is significantly slower than SRAM.

The coded BF [5] and Bloomier [16] are both based on a binary string generated from the item's set index (set ID). Supposing the length of the binary string is $w$, the coded BF and Bloomier allocate $w$ and $2w$ BFs respectively. Supposing the $i$-th bit of the string is $b_i$, the coded BF will insert $e$ into $i$-th BF if $b_i$ equals to 1, while Bloomier will insert $e$ into

$(2^{b_i}i)$-th BF. They are faster than BUFFALO but using more memory.

Combinatorial BF [17] uses one single BF along with multiple sets of hash functions. The hash functions are used to encode the set index. Though constant weight error correcting codes are used for encoding, the Combinatorial BF is not competitive either on speed or memory usage.

**Other variants of BF:** There are various variants of Bloom filters, aiming to improve one aspect of the standard Bloom filter at the cost of performance degradation in others. Typical variants of BF include: zero-error BF [22], cuckoo filter [23], [24], dynamic cuckoo filter [25], and more [26]–[28].

## III. The Magic Cube Bloom Filter

In this section, we will present the details of our Magic Cube Bloom filter (MC-BF for short). To better illustrate the advantages of our MC-BF, before introducing the final design (namely MC-BF$_3$), we will first introduce two prior versions (namely MC-BF$_1$ and MC-BF$_2$). This section is organized as follows: three versions of MC-BF will be introduced, and for each version, we will first discuss its data structure, then its insertion and query operation. Limitations of MC-BF$_1$ and MC-BF$_2$ will be discussed, and corresponding improvements will be provided in MC-BF$_2$ and MC-BF$_3$.

### A. MC-BF$_1$: Multiple Bloom Filters With Equal Widths

**Rationale:** The most straightforward idea to solve a multi-set query problem is to treat it as multiple in-or-not problems. An *in-or-not problem* is to tell whether an item is in a set or not when given a single set, which can be solved with a single Bloom filter. Suppose there are $d$ sets without intersection. Similar to BUFFALO [13], we need to build a Bloom filter for each set, and query an item in the $d$ Bloom filters to determine which set this item is in.

**Data Structure:**

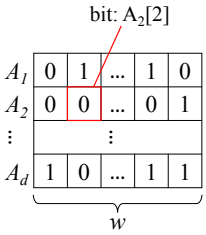



Fig. 1. Data structure of MC-BF$_1$.

As shown in Figure 1, MC-BF$_1$ consists of $d$ arrays, each of which contains $w$ bits. The MC-BF$_1$ is associated with $k$ independent hash functions, whose outputs are uniformly distributed in the range $[1, w]$. Notice that the value of $d$ is determined by the number of sets. We represent the $i$-th array with $A_i$, the $j$-th counter of the $i$-th array with $A_i[j]$, and the $l$-th hash function with $h_l(.)$. For simplicity's sake, we denote $A_1[i], A_2[i], ..., A_d[i]$ as $d$ *correlated bits of position* $i$, $h_1(e), h_2(e)..., h_k(e)$ as $k$ *mapped positions of item* $e$, and $A_i[h_1(e)], A_i[h_2(e)], ..., A_i[h_k(e)]$ as $k$ *mapped bits of item* $e$ *in* $A_i$.

**Insertion:** As shown in Figure 2, to insert an item $e$ from set $i$, we compute $k$ hash functions and set the $k$ mapped bits of $A_i$ ($A_i[h_1(e)], A_i[h_2(e)], ..., A_i[h_k(e)]$) to 1.

**Query:** As shown in Figure 2, to query an item $e$, we compute $k$ hash functions. For each array $A_i$ of the MC-BF$_1$, we check the $k$ mapped bits. If the $k$ mapped bits of $A_i$ are all 1, then we define set $i$ as a *candidate set* for the item $e$. After checking all the $d$ arrays, we will run into one of following three situations: 1) there is no candidate set for $e$, and the MC-BF$_1$ will return `none`; 2) there is only one candidate set $i$ for $e$, and the MC-BF$_1$ will return `i`; 3) there are more than one candidate sets for $e$, and the MC-BF$_1$ will return `error`. Notice that the query of an item can have one of the three results: none, set number $i$, or error. The query operation of MC-BF$_2$ and MC-BF$_3$ will also return one of those three results.

**Advantages and Limitations:** There is no salient advantage of MC-BF$_1$ since it is a straightforward algorithm to deal with multi-set query problems. The major limitation of MC-BF$_1$ is its *inflexibility to adjust the length of each array to adapt to various sizes of different sets.* When the sizes of the sets vary greatly, it is difficult to determine an appropriate length $w$ for the $k$ arrays, since too short a length causes heavy collisions in a large set and too long a length causes waste of memory in a small set. Therefore, MC-BF$_1$ only works well when sizes of the sets are all at the same level. In the next section, MC-BF$_2$ will be proposed to address this limitation.

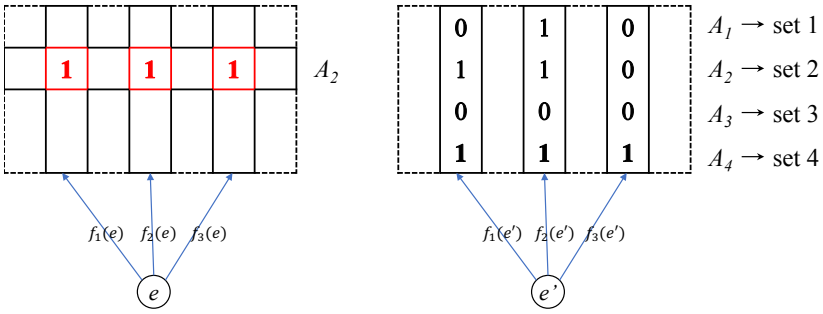
### B. MC-BF$_2$: Rotate The Magic Cube - Redistribution Of Mapped Bits

**Rationale:** To address the limitation of MC-BF$_1$, we need to break the boundaries of arrays and make a redistribution of mapped bits. If the mapped bits of one set are strictly located in one array, due to the fixed size of each array, too large or small a set will make the proportion of bits set to 1 too low or too high. Therefore, we remove the restriction and the mapped bits of one set can be located in any of the $k$ arrays, by adding an offset to each mapped bits of an item.

**Data Structure:** The data structure of MC-BF$_2$ is almost the same as that of MC-BF$_1$ except for an addition of one extra hash function $g(.)$.

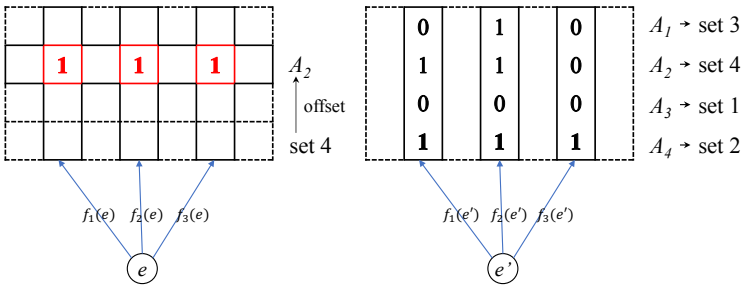
**Insertion:** As shown in Figure 3, to insert an item $e$ from set $i$, we compute $k$ hash functions and get the $k$ mapped locations: $h_1(e), ..., h_k(e)$. Instead of setting the $k$ mapped bits in $A_i$, we compute the target array $A_t$ by $t = 1 + (i + g(e))\%d$ (which makes sure that $t \in [1, d]$), and set $A_t[h_1(e)], A_t[h_2(e)], A_t[h_k(e)]$ to 1.

**Query:** As shown in Figure 3, to query an item $e$, first, we compute $h_1(e), ..., h_k(e)$ and $g(e)$. Second, for each set among $d$ sets, we denote set $i$ as the candidate set if $A_t[h_1(e)], A_t[h_2(e)], ..., A_t[h_k(e)]$ are all set to 1, where $t = 1 + (i + g(e))\%d$. After checking all the $d$ sets, MC-BF$_2$




Insertion of item $e$ from set 2. Query of item $e'$. In this case $e'$ belongs to set 4.

Fig. 2. Insertion and query of MC-BF$_1$. Suppose $k = 3$ and $d = 4$.




Insertion of item $e$ from set 4. Query of item $e'$. In this case $e'$ belongs to set 2.

Fig. 3. Insertion and query of MC-BF$_2$. Suppose $k = 3$ and $d = 4$.

returns one of the three results based on the number of candidate sets, as discussed in the query operation of MC-BF$_1$.

**Advantages and Limitations:** The most salient advantage of MC-BF$_2$ is that its accuracies of queries for different sets are basically the same, and won't be severely affected by the great variation of sizes among different sets. Unfortunately, the MC-BF$_2$ suffers two major limitations: *inflexibility* to the change of the number of sets, and *loss of speed* caused by across-machine-word memory access.

On the one hand, in many scenarios, the number of sets cannot be acquired beforehand, and only a rough estimation (*e.g.,* no bigger than 64) can be made. To make MC-BF$_2$ applicable in this kind of situations, we have to set $d$ to the upper bound of the number of sets, which may cause potential memory waste, or have to ask for extra memory during insertions, which makes the total memory usage unpredictable.

On the other hand, due to the cache mechanism, if a memory access to certain address *addr* is made, subsequent memory accesses to addresses located in the same machine word of *addr* will cost negligible amount of time. During queries of the MC-BF$_2$, the $d$ correlated bits of a certain position are frequently accessed at the same time. To utilize the spatial locality of cache mechanism, we store the $d$ correlated bits together. In other words, we first store $A_1[1], A_2[1], ..., A_d[1]$, then $A_1[2], A_2[2], ..., A_d[2]$, and so on. However, it cannot be guaranteed that those $d$ bits are located in the same machine word. It is possible that the former part of bits are located at the end of one machine word, and the latter part of bits are located at the beginning of the next machine word. In this case, the speed of the memory access will decrease.

In the next section, MC-BF$_3$ will be proposed to address these limitations.

### C. MC-BF$_3$: Fixed $d$ and Acceleration by Spatial Locality

**Rationale:** To address the limitations of MC-BF$_2$, we need to set a fixed size of $d$, which should be large enough to be applicable in most scenarios, and still keep a good spatial locality. Therefore, we choose 64 as the size of $d$ for two reasons. First, in most scenarios, the size $d$ is no larger than 64. Second, the size of a machine word is usually 64, so we can make full use of spatial locality. Also, we use an array of `long long` in C++ language to represent the MC-BF$_3$, to ensure that the $d$ correlated bits are located in the same machine word. Another improvement of MC-BF$_3$ is that every item has a distinctive offset for each of the mapped bits so as to improve randomness.

**Data Structure:** The data structure of MC-BF$_3$ is almost the same as that of MC-BF$_2$, except that $d$ is fixed at 64, and $g(.)$ is replaced by k hash functions $g_1(.), g_2(.), ..., g_k(.)$.

**Insertion:** As shown in Figure 4, to insert an item $e$ from set $i$, first, we compute $h_1(e), h_2(e), ..., h_k(e)$ and get $k$ mapped locations. We also compute $g_1(e), g_2(e), ..., g_k(e)$ and get $k$ offsets. Second, for each of the $k$ mapped locations, we compute the target array $A_{t_j}$ by $t_j = 1 + (i + g_j(e))\%d$ $(1 \leqslant j \leqslant k)$, and set $A_{t_1}[h_1(e)], A_{t_2}[h_2(e)], ..., A_{t_k}[h_k(e)]$ to 1.

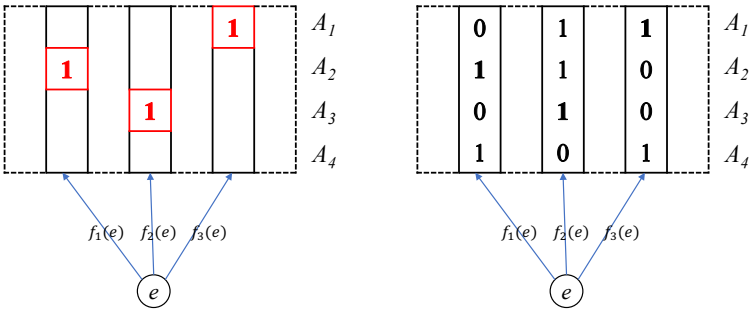



Fig. 4. Insertion and query of MC-BF$_3$. Suppose $k = 3$ and $d = 4$.

**Query:** As shown in Figure 4, to query an item $e$, first, we compute $h_1(e), h_2(e), ..., h_k(e)$ and $g_1(e), g_2(e), ..., g_k(e)$. Second, for each of the 64 sets, we denote set $i$ as the candidate set if $A_{t_1}[h_1(e)], A_{t_2}[h_2(e)], ..., A_{t_k}[h_k(e)]$ are all 1, where $t_j = 1 + (i + g_j(e))\%d$ $(1 \leqslant j \leqslant k)$. Notice that this process can be accelerated by bit manipulation. Finally, MC-BF$_3$ returns one of the three results based on the number of candidate sets, as discussed in the query operation of MC-BF$_1$.

**Advantages and Limitations:** There are two salient advantages of MC-BF$_3$. First, it supports the recording of an arbitrary number of sets as long as this number does not exceed 64. Second, during one query operation, it is guaranteed that only $k$ times of memory accesses are needed to produce the result, which ensures a high query speed. The major limitation of MC-BF$_3$ is that it cannot record multi-sets whose sizes are larger than 64. However, this limitation can be addressed by making a minor improvement on the MC-BF$_3$, which will be discussed in the next section.

### D. Scalability of MC-BF$_3$

One possible improvement that can be made to address the limitation of MC-BF$_3$ is to add more hash functions if more than 64 sets need to be recorded. If the maximum number of sizes does not exceed 128, then we use another $k$ hash functions $g_{k+1}(.), g_{k+2}(.), ..., g_{2k}(.)$ to compute the offsets for items from set $65, 66, ..., 128$. Specifically, during insertions, if the items are from set $1, 2, ..., 64$, we adopt the standard insertion operation; if the items are from set $65, 66, ..., 128$, we use $g_{k+1}(.), g_{k+2}(.), ..., g_{2k}(.)$ instead of $g_1(.), g_2(.), ..., g_k(.)$ to compute the $k$ corresponding offsets. During queries, we first calculate the candidate sets among set $1, 2, ..., 64$ using the standard query operation, and then use $g_{k+1}(.), g_{k+2}(.), ..., g_{2k}(.)$ instead of $g_1(.), g_2(.), ..., g_k(.)$ to calculate the candidate sets among set $65, 66, ..., 128$. Finally, we combine the two candidate sets together and return one of the three results based on the number of candidate sets. If the number of sets exceeds 128, then we could add more hash functions when computing the offsets. In this way, we make a trade-off between accuracy and scalability.

## IV. Mathematical Analysis

### A. Assumptions

**Assumptions about MC-BF$_3$:** In order to derive the formula, some assumptions about the Magic Cube Bloom filter and the multi-set query are necessary: the number of hash functions is $k$, the length of each array is $w$ bits, the number of arrays is $d$, the number of sets is $t$, the $i$-th set $(1 \leqslant i \leqslant t)$ has $n_i$ items and $\sum_{i=1}^{t} n_i = N$.

**Assumptions about MC-BF$_1$:** To compare the error rate between the MC-BF$_1$ and MC-BF$_3$, we follow the assumptions about multi-set query and memory usage above; hence there are $t$ arrays in the corresponding MC-BF$_1$, while each array holds $\frac{wd}{t}$ bits.

**Preliminaries and denotations:** According to prior work [12], the correct rate when querying a Bloom filter after $n$ items have been inserted into the array with $m$ bits and $k$ hash functions can be evaluated as $1-(1-e^{-kn/w})^k$. Furthermore, we denote $1-(1-e^{-kxt/wd})$ as $f(x)$. When $x$ equals to $n_i$ $(1 \leqslant i \leqslant t)$, $f(x)$ represents the correct rate when querying $i$-th set in $i$-th array of MC-BF$_1$. Moreover, assuming $l$ is a positive integer, $\mathbf{P}$ is a $l$-dimensional vector containing non-negative numbers $(p_1, p_2, \ldots, p_l)$, we denote $\prod_{i=1}^{l} f(p_i)$ as $F(\mathbf{P})$. In addition, set $n'_1 = n'_2 = \ldots = n'_t = \frac{N}{t}$, $\mathbf{P}_{\text{BF}} = (n_1, n_2, \ldots, n_t)$, $\mathbf{P}_{\text{MC-BF}_3} = (n'_1, n'_2, \ldots, n'_3)$. According to the algorithm of MC-BF$_3$, the process is equivalent to distribute all data evenly on a Bloom filter with $m \cdot d$ bits; hence the definition of $\mathbf{P}_{\text{MC-BF}_3}$ is reasonable.

### B. Error Classification

We can classify errors into two types: 1) item $e$ does not belong to any sets, but it is reported as a member of a specific set or more than one sets, *i.e.* `error`; 2) item $e$ only belongs to the $i$-th set, but it is also reported as a member of the others, leading to an `error` result. We will analyze the two types of error respectively.

**The First Type:** The occurrence of the first type error is the result of false positives when the queried item $e$ does not belong to any sets; thus the error rate of two structures are

$$
\begin{aligned}
E^1_{\text{MC-BF}_1} =& 1 - \prod_{i=1}^{t}[1 - (1 - e^{-kn_it/wd})^k] \\
=& 1 - F(\mathbf{P}_{\text{MC-BF}_1})
\end{aligned} \tag{1}
$$

and

$$
E^1_{\text{MC-BF}_3} = 1 - F(\mathbf{P}_{\text{MC-BF}_3})
$$

respectively.

Due to the convexity of $1 - F(\mathbf{P})$ and Jensen's inequality, we could simply draw the conclusion that

$$
E^1_{\text{MC-BF}_3} \leqslant E^1_{\text{MC-BF}_1}
$$

because of the uniform distribution of $P_{\text{MC-BF}_3}$. In fact, $P_{\text{MCBF}}$ is the minimum error rate under the constraints that the sum of all items of $t$-dimensional vector $P$ equals to $N$.

**The Second Type:** The occurrence of the second type error is also the result of false positives, but when the queried item $e$ only belongs to one specific set [29]–[31]; thus the error rate when the queried item belongs to $i$-th set is

$$
\begin{aligned}
E^2_{\text{MC-BF}_1}(i) &= 1 - \prod_{j \neq i}[1 - (1 - e^{-kn_jt/wd})^k] \\
&= 1 - \frac{F(\mathbf{P}_{\text{MC-BF}_1})}{f(n_i)}
\end{aligned} \tag{2}
$$

and

$$
E^2_{\text{MC-BF}_3}(i) = 1 - \frac{F(\mathbf{P}_{\text{MC-BF}_3})}{f(n'_i)}.
$$

According to law of total probability, we have

$$
\begin{aligned}
E^2_{\text{MC-BF}_1} &= \sum_{i=1}^{t} \frac{n_i}{N}\left[1 - \frac{F(\mathbf{P}_{\text{MC-BF}_1})}{f(n_i)}\right] \\
&= 1 - \frac{F(\mathbf{P}_{\text{MC-BF}_1})}{N}\sum_{i=1}^{t}\frac{n_i}{f(n_i)}.
\end{aligned} \tag{3}
$$

For a vector $\mathbf{P} = (p_1, p_2, \ldots, p_l)$, define

$$
G(\mathbf{P}) = 1 - \frac{F(\mathbf{P})}{\sum_{i=1}^{l} p_i}\sum_{i=1}^{l}\frac{p_i}{f(p_i)},
$$

then

$$
E^2_{\text{MC-BF}_1} = G(\mathbf{P}_{\text{MC-BF}_1}), E^2_{\text{MC-BF}_3} = G(\mathbf{P}_{\text{MC-BF}_3}).
$$

Fortunately, because of convexity of $G$, we can obtain the result

$$
E^2_{\text{MC-BF}_3} \leqslant E^2_{\text{MC-BF}_1}
$$

in the same way.

### C. Summary

In summary, when using the same size of memory, both types of error of MC-BF are lower than those of BF, as well as BUFFALO. Moreover, according to the convexity of two functions, the more the sizes of sets fluctuate, the more our algorithm can outperform the standard Bloom filter.

## V. Performance Evaluation

### A. Metrics

The performance of the Bloom filter is usually evaluated by accuracy and speed given a fixed memory. The accuracy is often measured by **Error Rate**, and the speed is often measured by **Throughput**.

**Error Rate (ER):** There are two types of errors: 1) An item is from set $i$ but the query result is not $i$ (the result is another set, or `none`, or `error`); 2) An item does not belong to any of the sets but the query result is not `none`. We denote the former type of error rate with $ER_{in}$, and the latter type of error rate with $ER_{out}$. Let $S$ be the query set, $S_e$ be the set of items whose query result generates a certain type of error. Then the error rate can be calculated as:

$$
ER = \frac{\mid S_e \mid}{\mid S \mid}
$$

**Throughput:** Let $n$ be the total number of certain operations (insertions or queries) executed on a MC-BF, and $t$ be the total execution time in nanosecond. The throughput is calculated as

$$
\text{Throughput} = \frac{n \cdot 1000}{t} \qquad \text{Mops}
$$

where Mops is the abbreviation of Million Operations per Second.

### B. Experimental Setup

**Datasets:** We use the MAC address tables acquired from the Stanford backbone as our experimental datasets, which can be acquired from [32]. Those MAC address tables are a collection of key-value pairs, where each key is a MAC address and each value is a destination port ID [33]–[35]. We preprocess those MAC address tables to make sure there is no intersection between any of the two sets. There are approximately 27000 distinct key-value pairs after preprocessing.

To evaluate the performance of our MC-BF, first we insert all the key-value pairs into the MC-BF. Suppose the total number of key-value pairs is $n$, and then we query those $n$ MAC addresses in the MC-BF. Since those $n$ MAC addresses all belong to this multi-set, we can calculate the $ER_{in}$. Then, we generate other $50 \cdot n$ MAC addresses that are not in this multi-set, query them in the MC-BF, and calculate the $ER_{out}$.

**Parameter Setting:** We allocate 64KB memory for the MC-BF and the BUFFALO. As for the BUFFALO, according to the optimal formula of the Bloom filter [12], # of hash functions needed can be calculated as:

$$
k = \frac{m}{n}\ln 2 \tag{4}
$$

where $m$ represents the memory size in bit, $n$ represents the total number of items. The result is $k = 13$. However, since we do not know the distribution of the multi-set beforehand, we vary $k$ from 2 to 13, and compare the performance between MC-BF and BUFFALO every time the $k$ is increased by 1.

**Implementation:** We use C++ as our programming language to implement the MC-BF and the BUFFALO. We use BOB Hash as our hash function which is acquired from an open source website [36]. The source code is available on GitHub [1].

**Computation Platform:** We perform all the experiments on a machine with 4-core CPUs (8 threads, Intel Core i7 @2.6 GHz) and 8 GB total DRAM memory. CPU has three levels of cache memory: two 32KB (where 1KB = 210 bytes) L1 caches for each core, one 256KB L2 cache for each core, and one 6MB (where 1MB = 220 bytes) L3 cache shared by all cores.

### C. Accuracy

In this section, we compare the performance between the MC-BF and the BUFFALO in terms of accuracy, and present the results in terms of error rate.

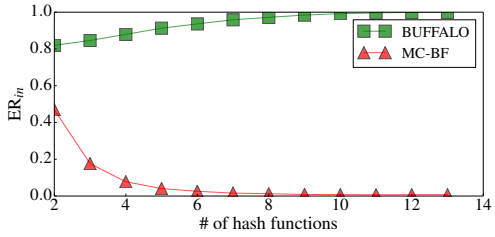



Fig. 5. $ER_{in}$ vs. # of hash functions

**$ER_{in}$ vs. # of hash functions:** Our experimental results, reported in Figure 5, show that as # of hash functions increases from 2 to 13, the $ER_{in}$ of the BUFFALO increases from 0.8 to around 1.0 and the $ER_{in}$ of the MC-BF decreases from 0.47 to almost 0. The $ER_{in}$ of the MC-BF is $[1.74, 148.5]$ times smaller than that of the BUFFALO as # of hash functions increases from 2 to 13.

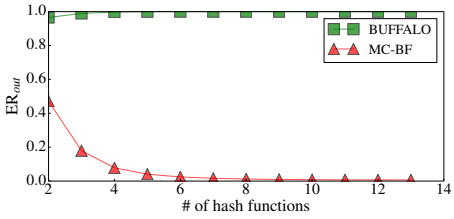



Fig. 6. $ER_{out}$ vs. # of hash functions

**$ER_{out}$ vs. # of hash functions:** Our experimental results, reported in Figure 6, show that as # of hash functions increases from 2 to 13, the $ER_{out}$ of the BUFFALO increases from 0.96 to 1.0 and the $ER_{out}$ of the MC-BF decreases from 0.47 to around 0. The $ER_{out}$ of the MC-BF is $[2.05, 149.7]$ times smaller than that of the BUFFALO as # of hash functions increases from 2 to 13.

### D. Speed

In this section, we compare the performance between the MC-BF and the BUFFALO in terms of speed, and present the results in terms of insertion throughput and query throughput. We use $k$ to represent # of hash functions in the experimental figures due to space limitation.

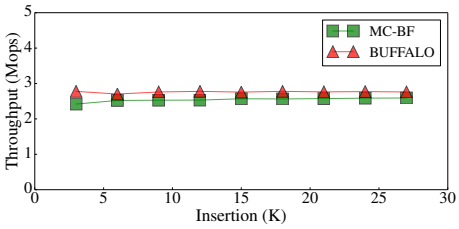



Fig. 7. Insertion throughput vs. # of insertions when $k = 2$

**Insertion throughput vs. # of insertions when k = 2:** Our experimental results, reported in Figure 7, show that when # of hash functions is fixed at 2, as # of insertions increases from 0.3K to 2.7K, the insertion throughput of the MC-BF is comparable to that of the BUFFALO.

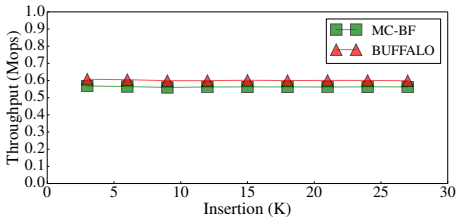



Fig. 8. Insertion throughput vs. # of insertions when $k = 13$

**Insertion throughput vs. # of insertions when k = 13:** Our experimental results, reported in Figure 8, show that when # of hash functions is fixed at 13, as # of insertions increases from 0.3K to 2.7K, the insertion throughput of the MC-BF is comparable to that of the BUFFALO.

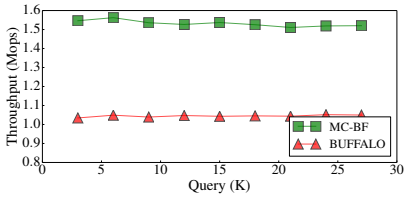



Fig. 9. Query throughput vs. # of queries when $k = 2$

**Query throughput vs. # of insertions when k = 2:** Our experimental results, reported in Figure 9, show that when # of hash functions is fixed at 2, as # of queries increases from 0.3K to 2.7K, the query throughput of the MC-BF is $[1.45, 1.53]$ times higher than that of the BUFFALO.

**Query throughput vs. # of insertions when k = 13:** Our experimental results, reported in Figure 10, show that when # of hash functions is fixed at 13, as # of queries increases from 0.3K to 2.7K, the query throughput of the MC-BF is $[3.14, 3.16]$ times higher than that of the BUFFALO.

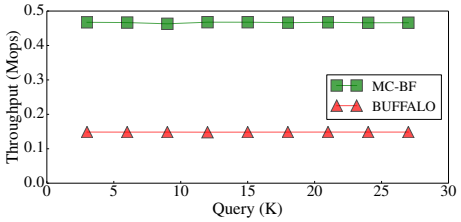



Fig. 10. Query throughput vs. # of queries when $k = 13$

## VI. Conclusion

Multi-set membership query is a crucial problem in many scenarios of big data application. In this paper, we propose a novel data structure, namely Magic Cube Bloom filter (MC-BF), for multi-set membership query. Compared to prior art, the Magic Cube Bloom filter is much more accurate when sizes of sets vary greatly, and its query speed is much higher. The `key technique` of the MC-BF is to redistribute items from different sets to offset the unevenness brought by various sizes. We have compared our MC-BF with BUFFALO, and experimental results show that our MC-BF outperforms BUFFALO by up to 148.5 times in terms of accuracy, and up to 3.16 times in terms of query speed. The source code of our algorithm is available on GitHub [1].

## Acknowledgement

This work is partially supported by Primary Research & Development Plan of China (2018YFB1004403, 2016YFB1000304), and NSFC (61672061). Tong Yang is the corresponding author.

## References

[1] Source code of the Magic Cube Bloom filter and its experiments. https://github.com/githubwyf/Magic-Cube-BF.

[2] Werner Bux, Wolfgang E Denzel, Ton Engbersen, Andreas Herkersdorf, and Ronald P Luijten. Technologies and building blocks for fast packet forwarding. *IEEE Communications Magazine*, 39(1):70–77, 2001.

[3] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X Liu, Qi Li, and Laurent Mathy. Guarantee ip lookup performance with fib explosion. *ACM SIGCOMM Computer Communication Review*, 44(4):39–50, 2015.

[4] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. In *High performance interconnects*, pages 44–51. IEEE, 2003.

[5] Francis Chang, Wu-chang Feng, and Kang 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. IEEE, 2004.

[6] Tong Yang, Alex X Liu, Muhammad Shahzad, Dongsheng Yang, Qiaobin Fu, Gaogang Xie, and Xiaoming Li. A shifting framework for set queries. *IEEE/ACM Transactions on Networking*, 25(5):3116–3131, 2017.

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

[8] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proc. ACM SIGCOMM*, pages 561–575, 2018.

[9] Li Fan, Pei Cao, and et al. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM ToN*, 8(3):281–293, 2000.

[10] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. Caching at the web scale. *VLDB 2017*.

[11] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *VLDB 2015*.

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

[13] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. Buffalo: Bloom filter forwarding architecture for large organizations. In *ACM CoNext 2009*.

[14] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auF der Heide, Hans Rohnert, and Robert E Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 1994.

[15] Myung Keun Yoon, JinWoo Son, and Seon-Ho Shin. Bloom tree: A search tree based on bloom filters for multiple-set membership testing. In *INFOCOM, 2014 Proceedings IEEE*, pages 1429–1437. IEEE, 2014.

[16] Denis Charles and Kumar Chellapilla. Bloomier filters: A second look. In *European Symposium on Algorithms*, pages 259–270. Springer, 2008.

[17] Fang Hao, Murali Kodialam, TV Lakshman, and Haoyu Song. Fast multiset membership testing using combinatorial bloom filters. In *INFOCOM 2009, IEEE*, pages 513–521. IEEE, 2009.

[18] Yan Qiao, Shigang Chen, Zhen Mo, and Myungkeun Yoon. When bloom filters are no longer compact: Multi-set membership lookup for network applications. *IEEE/ACM ToN*, 24(6):3326–3339, 2016.

[19] Haipeng Dai, Yuankun Zhong, Alex X Liu, Wei Wang, and Meng Li. Noisy bloom filters for multi-set membership testing. In *Proc. ACM SIGMETRICS*, pages 139–151, 2016.

[20] Haipeng Dai, L Meng, and Alex X Liu. Finding persistent items in distributed, datasets. In *Proc. IEEE INFOCOM*, 2018.

[21] Haipeng Dai, Muhammad Shahzad, Alex X Liu, and Yuankun Zhong. Finding persistent items in data streams. *Proceedings of the VLDB Endowment*, 10(4):289–300, 2016.

[22] Sándor Z Kiss, Eva Hosszu, János Tapolcai, Lajos Rónyai, and Ori Rottenstreich. Bloom filter with a false positive free zone. *Proc. IEEE INFOCOM, Honolulu, HI, USA*, 2018.

[23] David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Manuel R. Torres. 2-3 cuckoo filters for faster triangle listing and set intersection. In *ACM Sigmod-Sigact-Sigai Symposium on Principles of Database Systems*, pages 247–260, 2017.

[24] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.

[25] Hanhua Chen, Liangyi Liao, Hai Jin, and Jie Wu. The dynamic cuckoo filter. In *Network Protocols (ICNP), 2017 IEEE 25th International Conference on*, pages 1–10. IEEE, 2017.

[26] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[27] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Bloom filters: Design innovations and novel applications. (1):201–206, 2005.

[28] Yifeng Zhu, Hong Jiang, and Jun Wang. Hierarchical bloom filter arrays (hba): a novel, scalable metadata management system for large cluster-based storage. In *Cluster Computing, 2004 IEEE International Conference on*, pages 165–174. IEEE, 2004.

[29] Zhetao Li, Baoming Chang, Shiguo Wang, Anfeng Liu, Fanzi Zeng, and Guangming Luo. Dynamic compressive wide-band spectrum sensing based on channel energy reconstruction in cognitive internet of things. *IEEE Transactions on Industrial Informatics*, 2018.

[30] Zhetao Li, Fu Xiao, Shiguo Wang, Tingrui Pei, and Jie Li. Achievable rate maximization for cognitive hybrid satellite-terrestrial networks with af-relays. *IEEE Journal on Selected Areas in Communications*, 36(2):304–313, 2018.

[31] Zhetao Li, Yuxin Liu, Anfeng Liu, Shiguo Wang, and Haolin Liu. Minimizing convergecast time and energy consumption in green internet of things. *IEEE Transactions on Emerging Topics in Computing*, 2018.

[32] MAC address datasets. https://bitbucket.org/peymank/hassel-public/src/697b35c9f17e/hsa-python/examples/stanford/Stanford_backbone/?at=master.

[33] Fu Xiao, Zhongqin Wang, Ning Ye, Ruchuan Wang, and Xiang-Yang Li. One more tag enables fine-grained rfid localization and tracking. *IEEE/ACM ToN*, 26(1):161–174, 2018.

[34] Fu Xiao, Lei Chen, Chaoheng Sha, Lijuan Sun, Ruchuan Wang, Alex X Liu, and Faraz Ahmed. Noise tolerant localization for sensor networks. *IEEE/ACM Transactions on Networking*, 26(4):1701–1714, 2018.

[35] Hai Zhu, Fu Xiao, Lijuan Sun, Ruchuan Wang, and Panlong Yang. R-ttwd: Robust device-free through-the-wall detection of moving human with wifi. *IEEE Journal on Selected Areas in Communications*, 35(5):1090–1103, 2017.

[36] Hash website. http://burtleburtle.net/bob/hash/evahash.html.