Building Fast and Compact Sketches for Approximately Multi-Set Multi-Membership Querying

Rundong Li[1*], Pinghui Wang[2,1*], Jiongli Zhu[1], Junzhou Zhao[1], Jia Di[3], Xiaofei Yang[1] and Kai Ye[1]
[1] MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an, China
[2]Shenzhen Research Institute of Xi'an Jiaotong University, Shenzhen, China
[3]Second Affiliated Hospital of Xi'an Jiaotong University, Xi'an, China
{xjtulirundong, wearyee}@stu.xjtu.edu.cn, phwang@mail.xjtu.edu.cn, 15102966819@126.com,
{junzhou.zhao, xfyang, kaiye}@xjtu.edu.cn.

ABSTRACT

Given a set S, *Membership Querying* (MQ) answers whether a query element $q \in S$. It is a fundamental task in areas like database systems and computer networks. In this paper, we consider a more general problem, *Multi-Set Multi-Membership Querying* (MS-MMQ). Given n sets $S_0, \ldots, S_{n-1}$, MS-MMQ answers which sets contain element q. A direct way to address MS-MMQ is to build an MQ structure (e.g., Bloom Filter) for each set. However, the query and space complexities grow linearly with n and become prohibitive for a large n. To address this challenge, we propose a novel *Circular Shift and Coalesce* (CSC) framework to efficiently achieve approximate MS-MMQ. Instead of building an MQ data structure for each set, the CSC index encodes all n sets into a compact sketch and retrieves only a few bytes in the sketch for a query, which achieves high memory-efficiency and boosts the query speed by several times. CSC is compatible with mainstream data structures for Approximate MQ. We conduct experiments on real-world datasets and results demonstrate that our framework is up to 91.2 times faster and up to 48.9 times more accurate than state-of-the-art methods.

CCS CONCEPTS

• Information systems → Query optimization; **• Theory of computation → Bloom filters and hashing**; **Sketching and sampling**.

KEYWORDS

sketch; membership query; probabilistic data structure

ACM Reference Format:
Rundong Li[1*], Pinghui Wang[2,1*], Jiongli Zhu[1], Junzhou Zhao[1], Jia Di[3], Xiaofei Yang[1] and Kai Ye[1]. 2021. Building Fast and Compact Sketches for Approximately Multi-Set Multi-Membership Querying. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3452829

*Equal contribution. Pinghui Wang is the corresponding author.

1 INTRODUCTION

Membership Querying (MQ) is a fundamental problem in areas like network systems and databases [33, 41, 42]. Given a set S and a query q, the goal of MQ is to answer whether $q \in S$. There are efficient data structures for approximate MQ for large sets, such as the Bloom Filter [3], Cuckoo Filters [16], Quotient Filters [2] etc. One of the most attractive properties of them is the high computational- and memory-efficiency at the cost of a few false positives. Therefore, they are popular in real-world applications. For example, the Bloom filter is used to identify malicious URLs by Google Chrome to reduce the disk lookups for non-existent rows or columns in databases including Google Bigtable [7] and Apache Cassandra [22].

We study a more complex MQ problem called *Multi-set Multi-membership Querying* (MS-MMQ). Given sets $S_0, \ldots, S_{n-1}$, MS-MMQ aims to answer which sets contain a query element q. Note that an element may exist in multiple sets. MS-MMQ is different from well-studied *Multi-set membership Querying* (MS-MQ) in computer network area [8, 14, 20, 31, 38, 39, 48, 49], which assumes each element belongs to just one set among $S_0, \ldots, S_{n-1}$. MS-MMQ arises in real-world scenarios. Take genomic data searching as an instance. Efficient sequence search benefits biological applications like rapid identification of already-sequenced organisms that are highly similar to outbreak strains. However, as the volume of genome sequence data increases exponentially [4], it is a computational burden to search among vast archives for DNA sequences of interest. Such a sequence search problem can be abstracted as MS-MMQ. Specifically, archives storing various DNA sequences are abstracted as sets $S_0, \ldots, S_{n-1}$. For DNA sequence q of interest, the goal is to answer in which sets (i.e., archives) q exists.

To address MS-MMQ, a naive way is to allocate a well-designed MQ data structure (e.g. a Bloom Filter) for each set. However, such a direct manner needs $O(n)$ time-consuming hash computations to query all n MQ data structures and is computationally expensive for a large n. To address this challenge, tree-based methods [36, 37] allocate each set a homogeneous Bloom Filter and organize them as a binary tree. Each node in the tree resides a homogeneous Bloom Filter, in which Bloom Filters in the leaf nodes are for each set and Bloom Filters in the internal nodes are computed by Bloom Filters of their two children nodes. The query process is a traversal from the root node to leaf nodes, checking fewer Bloom Filters. However, tree-based methods consume much space cost because Bloom Filters of all nodes need to be stored. To address this problem, Bradley et al. [4] developed *BIGSI* that only builds a homogeneous Bloom Filter for each set. BIGSI is more memory-efficient but is slower since it traverses all Bloom Filters during the query phase. To further accelerate the query speed, Gupta et al. [18] propose *Repeated And Merged Bloom Filter* (called RAMBO hereafter). They draw inspiration from Count-Min sketch [13], which is a two-dimensional counter array of size $b \times r$ designed for frequency estimation in streaming data. Specifically, RAMBO replaces each entry of a Count-Min sketch with a Bloom Filter. In this way, RAMBO merges n sets into b meta-sets (i.e., partitions) without overlapping. Given a query q, RAMBO instead answers in which meta-sets q exists. To reduce false positives, RAMBO repeats the above procedure for r times and takes intersections of query results to generate a final candidate set. Gupta et al. show that RAMBO is faster than BIGSI when $br < n$ since there are fewer Bloom Filters to query.

However, we observe that all the above methods suffer from memory-inefficiency caused by the skewed set distribution, which is common in real-world datasets. That is, some sets may contain many elements, while others contain a few. In this situation, it is difficult or even impossible to find a proper size for the Bloom Filter of each set. On the one hand, if we use a Bloom Filter with large size to guarantee performance on large sets, it will cause a waste of space on small sets. On the other hand, small size of the Bloom Filter will induce poor query performance on large sets. Although RAMBO merges sets into meta-sets randomly, it can only guarantee each meta-set contains approximately the same number of sets, but the total number of elements in each meta-set may still be unevenly distributed. Therefore, RAMBO does not address the memory-inefficiency issue introduced by the uneven set size distribution. Besides, to handle a query, BIGSI and RAMBO require $O(n)$ and $O(br)$ memory accesses respectively, which are computationally expensive for large n and br. Although various methods [8, 14, 20, 31, 38, 39, 48, 49] have been developed for MS-MQ, most of them cannot be used to effectively and efficiently solve MS-MMQ (more discussions in Section 7).

To address the above issues and further improve accuracy and efficiency, we propose a novel framework *Circular Shift and Coalesce* (CSC). Our contributions are summarized as follows:

(1) CSC effectively alleviates the skew set distribution and achieves high memory efficiency. Based on its design, CSC also has significant improvements in query speed and accuracy.

(2) CSC can be integrated flexibly with mainstream MQ structures. As examples, we further propose two methods, CSC-Bloom Filter (CSC-BF) and CSC-Cuckoo Filter (CSC-CF). Compared with CSC-BF, CSC-CF is more accurate but with a high computational cost. Besides, CSC-CF is capable to deal with element deletions, which cannot be handled by CSC-BF.

(3) CSC realizes cheap updates for streaming inputs and handles emerging sets effectively.

(4) We conduct extensive experiments on a variety of real-world datasets. Experimental results demonstrate that our methods are up to 91.2 times faster and up to 48.9 times more accurate than state-of-the-art methods under the same memory usage.

The rest of this paper is organized as follows. Section 2 exhibits the formulated problem. Section 3 introduces preliminary knowledge of MQ and MS-MMQ. Sections 4 presents our framework CSC as well as CSC-BF and CSC-CF in detail. The performance evaluation and experimental results are in Section 6. Section 7 summarizes related work. Concluding remarks then follow.

2 FORMULATED PROBLEM

To formally define our problem, we first introduce notations. Given n sets $S_0, \ldots, S_{n-1} \subset \Omega$, where Ω refers to the universal set containing all the elements. Denote M_q as the set of indices for all the sets among $S_0, \ldots, S_{n-1}$ that contain a query element q. That is,

$$M_q = \{i : q \in S_i, i = 0, \ldots, n-1\}.$$

In this paper, we propose to build a compact sketch $\mathcal{S}$ for sets $S_0, \ldots, S_{n-1}$. Based on $\mathcal{S}$, fast and accurate methods are given to approximate M_q (denoted as $\hat{M_q}$) for any element q.

3 PRELIMINARIES

In this section, we first introduce two MQ sketches, Bloom Filter [3] and Cuckoo filter [16], which are used as basic building blocks for the CSC framework in this paper. Then, we elaborate MS-MMQ sketches BIGSI [4] and RAMBO [18, 19], and reveal their drawbacks.

3.1 Existing Sketches for MQ

3.1.1 ***Bloom Filter.*** For a set S with $|S|$ distinct elements, its Bloom Filter is an m-bit array $\mathcal{S}$, in which each bit is initialized with 0, and k independent hash functions $h_0, \ldots, h_{k-1}$, each of which maps an element $x \in S$ into an integer in $\{0, \ldots, m-1\}$ uniformly at random. The Bloom Filter supports two kinds of operations:

• **Insertion.** To insert an element $x \in S$, the values of k locations $\mathcal{S}[h_0(x)], \ldots, \mathcal{S}[h_{k-1}(x)]$ in array $\mathcal{S}$ are set to 1.

• **Query.** For a query q, the Bloom Filter returns $q \in S$ if all the k bits $\mathcal{S}[h_0(q)], \ldots, \mathcal{S}[h_{k-1}(q)]$ are 1, and $q \notin S$ otherwise.

The Bloom Filter has the following properties:

1) **Ignorant for duplicates.** It does not matter if an element is inserted for multiple times. Therefore, the Bloom Filter naturally handles the case when set S contains duplicate elements.

2) **No false negatives.** For elements $x \in S$, it is easy to find that there exist no false negatives in the Bloom Filter since all bits $\mathcal{S}[h_0(x)], \ldots, \mathcal{S}[h_{k-1}(x)]$ are set to 1.

3) **Existing false positives.** For $x' \notin S$, the Bloom Filter may incorrectly report $x' \in S$, which yields false positives since its associated k bits $\mathcal{S}[h_0(x')], \ldots, \mathcal{S}[h_{k-1}(x')]$ may be set to 1 by other elements. When m is very large, the false positive rate [10, 27] of the Bloom filter can be approximately computed as

$$\epsilon_{\text{bf}}(m, k, |S|) \approx \left(1 - \left(1 - \frac{1}{m}\right)^{k|S|}\right)^k \approx \left(1 - e^{-k|S|/m}\right)^k. \quad (1)$$

3.1.2 ***Cuckoo Filter.*** The Bloom Filter fails to handle dynamic sets with element deletions. Therefore, the Cuckoo Filter [16] is proposed as an alternative. A Cuckoo Filter is a hash table $\mathcal{S}$ with m buckets and each bucket contains a slots. Each slot stores an f-bits *fingerprint* of an element, generated by a hash function $f(\cdot)$. Let h be another hash function uniformly mapping elements into the m buckets. For element x, the Cuckoo Filter stores its fingerprint $f(x)$ in one of two candidate buckets $H_1(x)$ and $H_2(x)$ computed as:

$$H_1(x) = h(x)\%m, \qquad H_2(x) = H_1(x) \oplus h(f(x)).$$

The Cuckoo Filter supports the following operations.

• **Insertion.** To insert an element x, the Cuckoo Filter updates the sketch $\mathcal{S}$ as: When there are empty slots in bucket $H_1(x)$ or $H_2(x)$, the Cuckoo Filter stores the fingerprint $f(x)$ in either bucket.

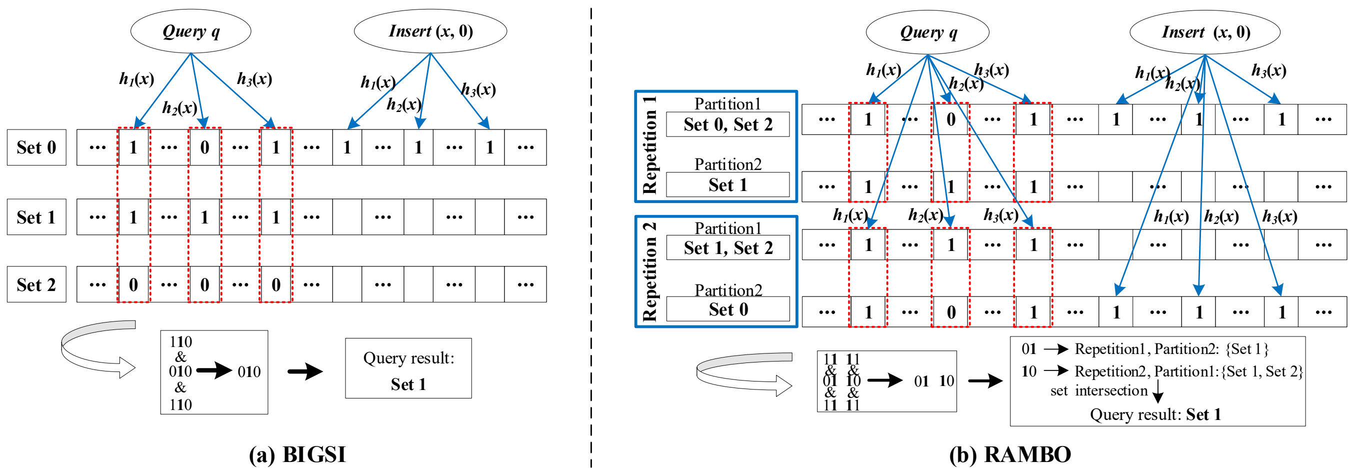


Figure 1: An example of BIGSI and RAMBO inserting an element x from set S_1 and querying an element q.

Otherwise, if slots in both buckets are occupied, the *eviction* process will be executed. That is, it randomly chooses a non-empty slot in $H_1(x)$ and $H_2(x)$, replaces the fingerprint $f(y)$ of element y in that slot with $f(x)$. One can easily find that $H_1(y) = H_2(y) \oplus h(f(y))$. In other words, with the knowledge of the fingerprint $f(y)$ and one of the bucket indices $H_1(y)$ or $H_2(y)$, one can locate the other bucket index easily. Based on this property, the Cuckoo Filter then places the removed fingerprint $f(y)$ into the alternate bucket of element y without retrieving the original element y. When the alternate bucket is also full, the *eviction* process will continue until an empty slot is found. Otherwise, if the number of recursive evictions reaches a pre-defined threshold, the Cuckoo Filter is considered as full, and it is necessary to expand its capacity.

• **Deletion.** To delete x, the Cuckoo Filter deletes its fingerprint $f(x)$ stored in bucket $H_1(x)$ or bucket $H_2(x)$.

• **Query.** For query q, the Cuckoo Filter returns $q \in S$ when either $H_1(q)$ or $H_2(q)$ stores $f(q)$, and $q \notin S$ otherwise.

There might be **False Positives** in a Cuckoo Filter: For a set S and two elements $x_1 \in S$, $x_2 \notin S$, if their fingerprints collide, i.e., $f(x_1) = f(x_2)$, then a false positive yields when querying the membership of x_2. Since we probe 2 buckets, each of which contains a slots, the false positive rate is approximately computed as

$$\epsilon_{\text{cf}}(a, m, f, |S|) \approx 1 - \left(1 - \frac{1}{2^f}\right)^{2a|S|/m}. \quad (2)$$

There are also variants of Bloom Filters and Cuckoo Filters with different characteristics, and we summarize them in Section 7. We argue that most variants can also be integrated with our framework in Section 4. For example, if the goal is to not only query M_q for element q, but also know its frequencies in membership sets, we can integrate our framework with Counting Bloom Filter [17] or a Cuckoo Filter with a compressed int count next to the fingerprint. For simplicity, we select the Bloom Filter and the Cuckoo Filter as two representative sketch structures for our framework.

3.2 Existing Sketches for MS-MMQ

3.2.1 ***BIGSI.*** Bradley et.al [4] proposed *BitSliced Genomic Signature Index* (BIGSI) for DNA sequence searching. The basic idea is to allocate an *identical* m-bit Bloom Filter with k hash functions for each of n sets. BIGSI inserts and queries elements as follows:

• **Insertion.** BIGSI inserts each $x \in S_i$ into the i-th Bloom Filter.

• **Query.** For a query q, BIGSI computes locations $h_0(q), \ldots, h_{k-1}(q)$ in the Filter and tests its membership in all n Bloom Filters.

Figure 1(a) is an example of BIGSI and there are only false positives. For a query $q \notin S_i$, when BIGSI reports $q \in S_i$ incorrectly, then S_i is called a false positive of q. Sets $S_i, i \in \{0, \ldots, n-1\} \setminus M_q$ can be q's false positives. The expectation of false positives of q is

$$\delta = \sum_{i \in \{0,\ldots,n-1\} \setminus M_q} \epsilon_{\text{bf}}(m, k, |S_i|),$$

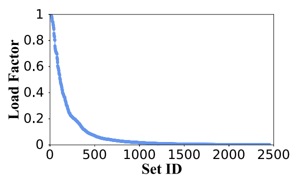
where $\epsilon_{\text{bf}}(m, k, |S_i|)$ is the false positive rate for a single Bloom Filter with $|S_i|$ elements inserted. The query complexity is $O(n)$ since it needs to check all the n Bloom Filters.

In [4], the authors further use bitwise "*and*" operation to accelerate the query process. That is, for position $h_i(q), 0 \leq i \leq k-1$, BIGSI accesses the corresponding bit in n Bloom Filters and gets a bit vector of length n. Overall, there are k n-bit vectors. BIGSI then performs bitwise "*and*" operation of these k vectors to obtain a result n-bit vector. The location of 1s in this n-bit vector corresponds to the sets containing the query. However, the formation of a n-bit vector needs kn memory accesses since each bit of it is not continuous in the memory space of BIGSI.

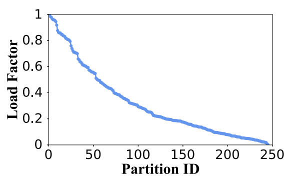
3.2.2 ***RAMBO.*** The query time of BIGSI grows linearly with the number of sets, which is a computational burden for large n. To address this problem, Gupta et al. [18, 19] proposed a structure called *Repeated And Merged Bloom Filter* (RAMBO), which also utilizes a group of Bloom Filters. As shown in Figure 1(b), the basic idea of RAMBO is to partition all n sets $S_0, \ldots, S_{n-1}$ uniformly into b ($b < n$) random partitions $P_0, \ldots, P_{b-1}$ by a hash function g, i.e., $P(g(S_i) = j) = \frac{1}{b}, i \in \{0, \ldots, n-1\}, j \in \{0, \ldots, b-1\}$. An identical m-bit Bloom Filter is then allocated for each **nonempty** partition. RAMBO inserts and queries elements as follows:

• **Insertion.** To insert an element x from set S_i, RAMBO inserts x into the Bloom Filter belonging to partition $g(S_i)$.

• **Query.** To answer the $\hat{M}_q$ of a query q, RAMBO computes the k locations $h_0(q), \ldots, h_{k-1}(q)$, and tests the membership in all b



(a) BIGSI



(b) RAMBO

Figure 2: (RCV1-X) Load factor distributions of Bloom Filters used in BIGSI and RAMBO.

Bloom Filters. For partitions reporting the existence of q, RAMBO takes the union of sets they contain as $\hat{M}_q$.

To reduce false positives, RAMBO uses r repetitions $R_j, j \in \{0, ..., r-1\}$ to repeat the random partition process independently. br Bloom Filters are used in total. At insertion phase, RAMBO inserts each element for r times. At query phase, RAMBO traverses br Bloom Filters and each repetition R_j generates a candidate set $\hat{M}_q^i$. To get $\hat{M}_q$, RAMBO takes the intersection of $\hat{M}_q^0, \ldots, \hat{M}_q^{r-1}$. As a special case, when $r = 1$ and $b \rightarrow +\infty$, RAMBO has one repetition and b non-empty partitions, each contains only one set among $S_0, \ldots, S_{n-1}$. In this case, RAMBO reduces to BIGSI [4] as it allocates each set an identical Bloom Filter. Experiments in [19] show that RAMBO accelerates BIGSI efficiently when $br < n$.

3.3 Limitations of Existing Sketches

Real-world datasets often have a skew set distribution. For example, there are 2,456 sets in dataset RCV1-X (detailed statistics in Table 1), in which some sets contain over 100,000 elements while others contain fewer than 1,000. Set distribution greatly influences the performance of an MS-MMQ data structure, as will be discussed in Section 5.4. Unfortunately, BIGSI and RAMBO fail to take it into consideration, which leads to serious inefficiency in query results and space usage. Particularly, we compute the load factors of Bloom Filters in both methods on dataset RCV1-X in Figure 2, where the load factor of a Bloom Filter is the ratio of its bits set to 1. We sort the load factors in descending order. Figure 2(a) shows that the load factors of Bloom Filters in BIGSI are also skewed. Such a heavy skewness leads to the following dilemmas:

• **High false positive rates in Bloom Filters with large load factors.** According to Figure 2(a), load factors of 10% Bloom Filters are larger than 0.2 and some of them are even larger than 0.9, which indicates that almost all bits in the corresponding Bloom Filters are set to 1. Therefore, it is probable that these sets to be false positives.

• **Serious space inefficiency in Bloom Filters with small load factors.** There are more than 2,000 Bloom Filters whose load factors are smaller than 0.1 in Figure 2(a). This indicates that the memory spaces allocated for these Bloom Filters are far larger than needed to achieve the desired accuracy, which is a serious waste of space.

Although RAMBO alleviates these problems to some extent, it still suffers from heavy skewness, as shown in Figure 2(b). Moreover, BIGSI and RAMBO fail to delete elements. To delete elements, a naive way is to replace Bloom Filters with Cuckoo Filters. However, such a direct modification still suffers from the skew set distribution. All these problems analyzed above inspire us to develop new frameworks for effective and efficient MS-MMQ.

4 OUR FRAMEWORK

We first introduce the basic idea behind our framework **CSC** (*Circular Shift and Coalesce*) in Section 4.1. Based on CSC, we further propose CSC-Bloom Filter (CSC-BF) and CSC-Cuckoo Filter (CSC-CF) in Section 4.2 and Section 4.3 respectively.

4.1 Basic Idea

For all sets S_i, $i = 0, \ldots, n-1$ of interest, define set

$$S = \{(x, i) : x \in S_i, i = 0, \ldots, n-1\}. \tag{3}$$

To answer whether $x \in S_i$, we instead answer whether pair $(x, i) \in S$. In this way, we merely construct one AMQ sketch for S. To achieve this goal, a naive way is to insert (x, i) into the sketch as a unique element. For a query q, we enumerate all possible pairs (q, i), $i = 0, \ldots, n-1$ and check the membership of each pair. However, it requires $O(n)$ time-consuming hash computation. To improve efficiency, we propose a *Circular Shift and Coalesce* (CSC) framework. A pair (x, i) contains two types of information: (1) ***existence information*** marking the appearance of element x. (2) ***set information*** recording that x is in set S_i. Given an AMQ sketch of size m with its hash functions $\mathcal{H}$, the mentioned way above is to insert each pair $(x, i) \in S$ via $\mathcal{H}((x, i))$. However, such a manner encodes the two types of information in the same position, making them indistinguishable. Therefore, during the query phase, checking the membership of all possible pairs $(q, 0), \ldots, (q, n-1)$ is unavoidable, which requires $O(n)$ hash computation and is computationally intensive for large n. CSC instead encodes the existence information as an anchor and set information as an offset. Specifically, for (x, i), CSC first computes $\mathcal{H}(x)\%m$ as its anchor location. After that, CSC encodes the set information as an offset i. The final location to insert pair (x, i) is $(\mathcal{H}(x)\%m + i)\%m$.

For a query q, CSC computes its anchor location $\mathcal{H}(q)\%m$ and then checks the next n consecutive locations from $(\mathcal{H}(q)\%m+0)\%m$ to $(\mathcal{H}(q)\%m+n-1)\%m$. If the existence of q is found when checking the i-th location, CSC updates $\hat{M}_q$ as $\hat{M}_q \leftarrow \hat{M}_q \cup \{i\}$, where $\hat{M}_q$ is an estimate of M_q and is initialized as an empty set.

To handle large n, we integrate the *Repeating and Merging* technique. We define $r \geq 2$ repetitions and each repetition R_j has an identical AMQ sketch with a unique partition function g_j mapping all n sets $S_0, \ldots, S_{n-1}$ uniformly into a few partitions $P_0, \ldots, P_{b-1}$. There are no intersections between any two partitions. The CSC framework updates and queries an element as follows:

• **Update.** To update (x, i) in repetition $R_j, 0 \leq j \leq r-1$, CSC updates it at location $(\mathcal{H}(x)\%m + g_j(S_i))\%m$, in which $\mathcal{H}(x)\%m$ is the anchor and $g_j(S_i)$ is the offset. An example is in Figure 3(a).

• **Query.** For a query q, CSC answers $\hat{M}_q^j$ in each repetition. In repetition $R_j, 0 \leq j \leq r-1$, CSC computes its anchor location $\mathcal{H}(q)\%m$, and checks the next b locations $(\mathcal{H}(q)\%m + 0)\%m, \ldots, (\mathcal{H}(q)\%m + b-1)\%m$ to determine which partitions $P_i, 0 \leq i \leq b-1$ contain q. For locations reporting the existence of q, CSC then takes the union of sets these partitions contain as $\hat{M}_q^j$ for R_j. After that, CSC takes the intersection of $\hat{M}_q^0, \ldots, \hat{M}_q^{r-1}$ as $\hat{M}_q$. An example of query process is shown in Figure 3(b).

CSC brings about the following advantages:

(1) Higher memory efficiency. CSC constructs just one AMQ sketch for compressed storage of n sets. Elements from different

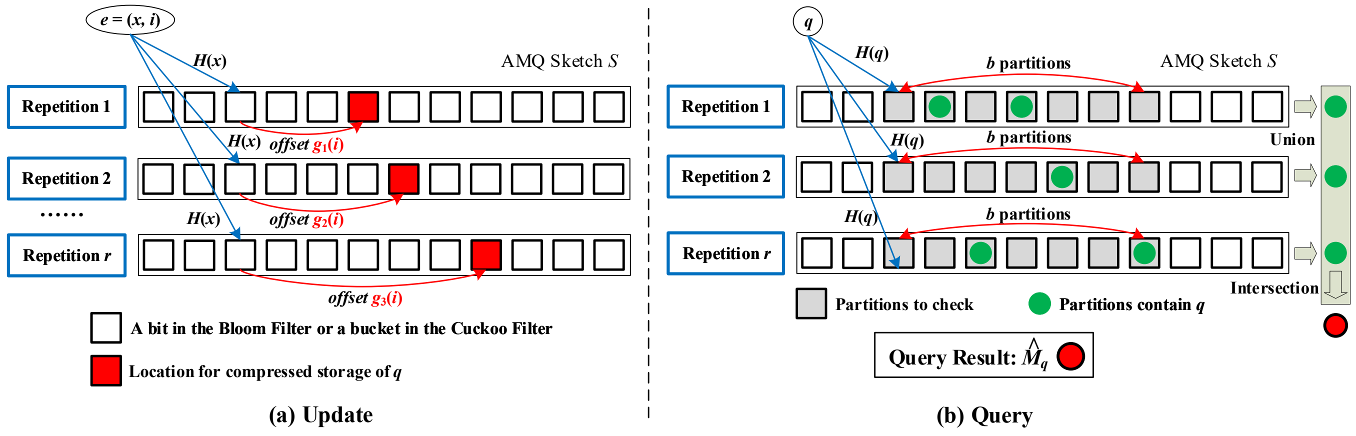


Figure 3: Our CSC framework for updating an element x from set S_i and performing a MS-MMQ for an element q.

sets are uniformly distributed, which effectively utilizes the memory space and alleviates skewness among different sets.

(2) Fewer false positive sets. Due to the high memory efficiency, CSC alleviates the congestion of sketches for large sets and utilizes the wasted space allocated to small sets more effectively. Therefore, there are fewer false positives for large sets while the false positives for small sets still meet the desired accuracy.

(3) Faster query speed. When inserting (x, i), CSC encodes the existence information as an anchor and the set information as an offset. This guarantees good locality in storage, which benefits the query efficiency. Given a query q, CSC first computes its anchor location $\mathcal{H}(q)\%m$. If q is inserted previously, its set information must be encoded in some of the next b locations $(\mathcal{H}(q)\%m + 0)\%m, \dots, (\mathcal{H}(q)\%m + b - 1)\%m$. These b locations are continuously stored in memory space, and we need fewer memory accesses compared with BIGSI and RAMBO, which significantly accelerates the query process. Detailed analysis is in Section 5.2.

(4) Flexible to emerging sets. When a new set S_n emerges and is inserted, CSC merely needs to compute its partitions $g_j(S_n)$ in each repetition R_j and encode elements in S_n like other sets without allocating extra memory space or reconstructing the sketch.

We integrate the CSC framework with two mainstream AMQ sketches, e.g., the Bloom Filter and the Cuckoo Filter in Section 3.

4.2 CSC-BF

The basic building block of CSC-BF is an m-bit Bloom Filter with k independent hash functions $h_0, \dots, h_{k-1}$ uniformly distributed among $\{0, \dots, m-1\}$. All bits are initialized to 0. We allocate such a Bloom Filter for each repetition $R_j, 0 \le j \le r-1$. Each repetition has its unique partition function g_j mapping n sets disjointly into b partitions. CSC-BF inserts and queries elements as:

• **Insertion.** For pair (x, i) in set S, CSC-BF computes its k locations $(h_1(x)\%m + g_j(S_i))\%m, \dots, (h_k(x)\%m + g_j(S_i))\%m$ in each repetition $R_j, 0 \le j \le r-1$ and sets the corresponding bit values to 1.

• **Query.** To query q in repetition $R_j, 0 \le j \le r-1$, CSC-BF computes k anchor locations $h_0(q)\%m, \dots, h_{k-1}(q)\%m$ and checks the k bits $(h_0(q)\%m + i)\%m, \dots, (h_{k-1}(q)\%m + i)\%m$ to determine whether q is in partition $P_i, 0 \le i \le b-1$. For partitions reporting the existence of q, we take the union of sets mapped into these partitions to generate a candidate membership set $\hat{M}_q^j$ of R_j. At last, we take the intersection of $\hat{M}_q^j, 0 \le j \le r-1$ to obtain $\hat{M}_q$.

Compared with RAMBO, CSC-BF achieves faster query speed by fewer memory accesses. Denote the size of a machine word as w, in each repetition, CSC-BF only needs $k \cdot \lceil \frac{b}{w} \rceil$ memory accesses since each B-bit vector is continuous in memory space, far fewer than RAMBO's kb random memory accesses. Moreover, CSC-BF uses the bitwise "*and*" operation for further acceleration. That is, when querying in each repetition, we first access k b-bit vectors by $k \cdot \lceil \frac{b}{w} \rceil$ memory accesses, and perform bitwise "*and*" operation among these k vectors to obtain a result b-bit vector. Locations of 1 in this vector correspond to partitions reporting the existence of q.

4.3 CSC-CF

We integrate CSC framework with the Cuckoo Filter as CSC-CF to support element deletions. The basic building block of CSC-CF is an m-bucket Cuckoo Filter with a slots in each bucket. Each slot stores an element's *fingerprint* generated by a hash function f. We allocate such a Cuckoo Filter for each repetition $R_j, 0 \le j \le r-1$. CSC-CF supports **Insertion**, **Deletion**, **Query** operations as follows:

• **Insertion.** To insert (x, i) in R_j, CSC-CF stores its fingerprint $f(x)$ in one of the two candidate buckets $H_1(x, g_j(S_i))$ and $H_2(x, g_j(S_i))$. $H_1(x)$ and $H_2(x)$ are not independent with each other because we can compute $H_2(x)$ once knowing $f(x)$ and $H_1(x)$. Therefore, CSC-CF explicitly encodes x and i only through H_1. For (x, i), its $H_1(x, g_j(S_i))$ and $H_2(x, g_j(S_i))$ are computed as:

$$H_1(x, g_j(S_i)) = (h(x)\%m + g_j(S_i))\%m,$$
$$H_2(x, g_j(S_i)) = H_1(x, g_j(S_i)) \oplus h(f(x)).$$

If both buckets are full, CSC-CF executes the *eviction* process.

• **Deletion.** To delete (x, i), CSC-CF removes its fingerprint $f(x)$ in either bucket $H_1(x, g_j(S_i))$ or $H_2(x, g_j(S_i))$ in each repetition.

• **Query.** To query q in R_j, CSC-CF computes its anchor location $H_1(q) = h(q)\%m$ and checks the next b locations. Note that since H_1 and H_2 are not independent with each other, CSC-CF only checks the b consecutive locations for $H_1(q)$ and uses a b-bit vector to record the membership partitions of q reported by $H_1(q, i), 0 \le i \le b-1$. If $f(q)$ is not found in $H_1(q, i)$, we check $H_2(q, i)$ and update $\hat{M}_q^j$. The set union and intersection operation is the same as CSC-BF.

The above query process is still true in the face of evictions since an element pair (x, i) exists in either $H_1(x, g_j(S_i))$ or $H_2(x, g_j(S_i))$.

The query speed of CSC-CF also benefits from fewer memory accesses. Suppose each bucket occupies c bits and denote the size of a machine word as w, CSC-CF uses $\lceil \frac{cb}{w} \rceil$ memory accesses to check b buckets next to anchor $H_1(q)$. Although $H_1(q, i), 0 \le i \le b-1$ are stored continuously, $H_2(q, i), 0 \le i \le b-1$ may not be continuous after the "*xor*" operation between $H_1(q, i)$ and $h(f(x))$. Fortunately, we find that $H_2(q, i), 0 \le i \le b-1$ keep good locality in storage.

Consider the binary address of location $H_1(q)$. For b partition IDs, we use a bit vector of length $\lceil \log_2 b \rceil$ to represent them. We first analyze the binary addresses of b locations from $H_1(q) + 0$ to $H_1(q) + b - 1$. Specifically, we discuss the following two cases:

• **Case 1: the** $(\lceil \log_2 b \rceil + 1)$**-th bit of** $H_1(q)$ **is 0.** In this case, the b binary addresses from head location $H_1(q, 0) = H_1(q) + 0$ and the binary address to tail location $H_1(q, b-1) = H_1(q) + b - 1$ only differ within the lowest $(\lceil \log_2 b \rceil + 1)$ bits. Since the b alternate locations are computed by bitwise "xor" between $H_1(q, i), 0 \le i \le b-1$ and a same $h(f(x))$, the b alternate locations also differ within the lowest $(\lceil \log_2 b \rceil + 1)$ bits and keep good locality.

• **Case 2: the** $(\lceil \log_2 b \rceil + 1)$**-th bit of** $H_1(q)$ **is 1.** In this case, we first consider an address $H_1(q, o) = H_1(q) + o$, in which o is an integer and its lowest $\lceil \log_2 b \rceil$ bits are all 1. If $o \ge b-1$, then the b addresses $H_1(q, 0), \ldots, H_1(q, b-1)$ only differ at the lowest $\lceil \log_2 b \rceil$ bits. The addresses of b alternate locations $H_2(q, 0), \ldots, H_2(q, b-1)$ also differ at the lowest $\lceil \log_2 b \rceil$ bits. If $o < b - 1$, then the address interval $[H_1(q, 0), H_1(q, b-1)]$ is further divided into two groups by the demarcation address $H_1(q, o)$. For subinterval $[H_1(q, 0), H_1(q, o)]$, the addresses within it differ within the lowest $\lceil \log_2 b \rceil$ bits. For subinterval $[H_1(q, o+1), H_1(q, b-1)]$, the addresses within it also differ within the lowest $\lceil \log_2 b \rceil$ bits. Therefore, The addresses of alternate locations of separate subintervals differ within the lowest $\lceil \log_2 b \rceil$ bits and keep good locality.

5 ANALYSIS

In this section, we analyze the accuracy of the CSC framework and the resource cost. After that, we give a comparison with existing work. Finally, we give a discussion to sum up the properties of CSC.

5.1 Accuracy Analysis

We first analyze how having sets mapped to consecutive bit positions affects the false positive rate.

Theorem 1. *For a query q existing in none of the n sets, the false positive rate $\epsilon_{\text{CSC-BF}}$ of CSC-BF ($r = 1, b \longrightarrow +\infty$) incorrectly reporting $q \in S_i, i \in \{0, \ldots, n-1\}$ is approximated as*

$$\epsilon_{\text{CSC-BF}} = \left(1 - \prod_{x \in \Omega} \left(1 - \frac{|M_x|}{m}\right)^k\right)^k \approx \epsilon_{\text{bf}}(m, k, \sum_{i=0}^{n-1} |S_i|). \tag{4}$$

Ω is the set of unique elements and M_x is the membership-set of x.

Proof. For an element pair (x, i), the probability of a bit set to 1 by $H(x, i) = (h(x)\%m + i)\%m$ is $1/m$. For two element pairs (x, i) and (y, j), we have two observations about $Pr(H(x, i) = H(y, j))$:

• **Observation 1. If** $x \ne y$**,** $Pr(H(x, i) = H(y, j)) = 1/m$**.** Since x and y are different elements, they are independent with each other.

• **Observation 2. If** $x = y$ **and** $i \ne j$**,** $Pr(H(x, i) = H(y, j)) = 0$**.** x and y are same element, and they have a same anchor location but different offsets. Therefore, they cannot be mapped to a same bit.

From **Observation 2**, a bit can only be set to 1 simultaneously by different elements. For an element $x \in \Omega$, any pair $(x, i)_{i \in M_x}$ can set a bit with a probability of $1/m$. Thus, the probability that a bit is set by element x is $|M_x|/m$. Further, a bit is still 0 after applying k hash functions with a probability $(1 - |M_x|/m)^k$.

From **Observation 1**, any two different elements from Ω set a bit independently. After all elements are inserted, a bit is still 0 with a probability $\prod_{x \in \Omega}(1 - |M_x|/m)^k$.

For a query q not existing in any of n sets, the false positive rate $\epsilon_{\text{CSC-BF}}$ of incorrectly reporting $q \in S_i$, i.e., all k bits $(h_0(q)\%m + i)\%m, \ldots, (h_{k-1}(q)\%m + i)\%m$ are 1, is computed as

$$\begin{aligned}\epsilon_{\text{CSC-BF}} &= (1 - \prod_{x \in \Omega}(1 - \frac{|M_x|}{m})^k)^k \approx (1 - \prod_{x \in \Omega} e^{-\frac{k|M_x|}{m}})^k \\ &= (1 - e^{-\frac{k \cdot \sum_{x \in \Omega} |M_x|}{m}})^k = (1 - e^{-\frac{k \sum_{i=0}^{n-1} |S_i|}{m}})^k \\ &\approx \epsilon_{\text{bf}}(m, k, \sum_{i=0}^{n-1} |S_i|).\end{aligned} \tag{5}$$

Since m is very large, the approximations in Eq. (5) is reasonable. □

According to Theorem 1, mapping sets to consecutive positions has little impact on the false positive rate. Therefore, in the follow-up accuracy analysis of CSC-BF and CSC-CF, we treat them as traditional BF and CF and use $\epsilon_{\text{bf}}, \epsilon_{\text{cf}}$ in analysis for convenience.

Theorem 2. *For a query q existing in v sets out of n sets $S_0, \ldots, S_{n-1}$, i.e., $v = |M_q|$. For a set $S_i, i \in \{0, \ldots, n-1\} \setminus M_q$, the false positive rate ϵ of CSC-BF incorrectly reporting $q \in S_i$ is bounded as*

$$\epsilon \lessapprox \left(1 - \left(1 - \epsilon_{\text{bf}}(m, k, \sum_{i=0}^{n-1} |S_i|)\right)\left(1 - \frac{1}{b}\right)^v\right)^r, \tag{6}$$

where $r \ge 2$ and $\epsilon_{\text{bf}}(m, k, \sum_{i=0}^{n-1} |S_i|)$ is the false positive rate of a Bloom Filter with m bits, k hash functions and $\sum_{i=0}^{n-1} |S_i|$ elements inserted. the formula of $\epsilon_{\text{bf}}(m, k, \sum_{i=0}^{n-1} |S_i|)$ is shown in Eq.(1).

Proof. In a repetition of CSC, there are two cases between any one of $n - v$ sets S_i satisfying $q \notin S_i$ and any one of v sets $S_j, j \in M_q$: **Case 1)** None of $S_j, j \in M_q$ is in the same partition as S_i. **Case 2)** At least one $S_j, j \in M_q$ is in the same partition as S_i;

The probability of any two sets mapped into the same partition is $Pr(g(S_x) = g(S_y)) = (1/b) \times (1/b) \times b = 1/b$. In Case 1, the probability that S_i is not in the same partition as $S_j, j \in M_q$ is $(1 - 1/b)^v$. Then Case 2 happens with a probability $1 - (1 - 1/b)^v$.

In one repetition, a false positive $q \in S_i$ must belong to one of the two above cases. In Case 2, such a false positive definitely occurs. In Case 1, the false positive occurs when the Bloom Filter wrongly reports it with a probability $\epsilon_{\text{bf}}(m, k, |S|)(1 - (1/b))^v$, in which $|S|$ is the number of unique elements. Therefore, the false positive rate in one repetition is $1 - (1 - (1/b))^v + \epsilon_{\text{bf}}(m, k, |S|)(1 - (1/b))^v$. For r repetitions, the total false positive rate is

$$\epsilon = \prod_{i=0}^{r-1}\left(1 - (1 - \epsilon_{\text{bf}}(m, k, |S_i|))\left(1 - \frac{1}{b}\right)^v\right). \tag{7}$$

Among $\sum_{i=0}^{n-1}|S_i|$ inserted pairs in each repetition, duplicates exist if same elements from different sets are mapped into same partitions, i.e., $|S| \le \sum_{i=0}^{n-1}|S_i|$. Then $\epsilon_{\text{bf}}(m,k,|S|) \le \epsilon_{\text{bf}}(m,k,\sum_{i=1}^{n}|S_i|)$. Combined with Equation (7), we obtain Inequality (6). □

Theorem 3. *For a query q existing in v sets out of n sets $S_0, \dots, S_{n-1}$, i.e., $v = |M_q|$. For a set $S_i, i \in \{0, \dots, n-1\} \setminus M_q$, the false positive rate ϵ of CSC-CF incorrectly reporting $q \in S_i$ is bounded as*

$$\epsilon \lessapprox \left(1 - \left(1 - \frac{1}{2f}\right)^{\frac{2a\sum_{i=0}^{n-1}|S_i|}{m}} \left(1 - \frac{1}{B}\right)^{v}\right)^{r}, \tag{8}$$

where $r \ge 2$, m is the number of buckets in each repetition, a is the number of slots in each bucket, and f is the length of fingerprint (in bits). When not using the Repeating and Merging technique, i.e., $r = 1$ and $b \to +\infty$, we have $\epsilon \approx \epsilon_{\text{cf}}(a, m, f, \sum_{i=0}^{n-1}|S_i|)$.

Proof. The proof is analogous with that of Theorem 2 and thus we omit it due to limited space. □

Particularly, the original paper of RAMBO [18] assumes that the false positive rate of a Bloom Filter is a constant value, which is ***incorrect*** because it is relevant with the number of inserted elements. Obviously, it is not a constant value. Therefore, it is difficult to theoretically compare CSC-BF and RAMBO in terms of accuracy. Instead, we perform empirical experiments to demonstrate the superiority of our method on real-world datasets in Section 6.

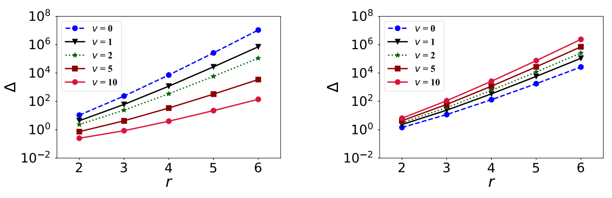
5.2 Complexities

Memory Cost. The space cost of the CSC framework depends on specific AMQ sketch. For example, a CSC-BF of r repetitions with an m-bit Bloom Filter for each cost mr bits.

Insertion Cost. For an AMQ sketch of size m, the CSC framework inserts (x, i) by $(\mathcal{H}(x)\%m + i)\%m$. Compared with $\mathcal{H}(x)\%m$ for only inserting x, it introduces an extra addition and modulo operation, which is not the computational overhead compared to the computation of $\mathcal{H}(x)$. Therefore, the CSC-AMQ has a similar insertion cost as the original AMQ sketch. For r repetitions, the insertion speed is linear with r in theory since each element is inserted for r times. This issue can be effectively alleviated through parallel insertion.

Query Cost. The query cost of CSC consists of two parts:
(1) Time to check which partitions contain q. There are b partitions to check in a repetition and r repetitions requires $O(rb)$ operations.
(2) Time for set union and intersection. In one repetition, for partitions which report the existence of q, we require to take the union of sets these partitions contain to get a candidate set. According to Theorem 2, the expected number of sets in the candidate set is $(n-v)\epsilon + v$. That is, the union in one repetition takes $(n-v)\epsilon + v$ operations. After that, set intersection among the r candidate sets takes $((n-v)\epsilon + v)r$ operations. Therefore, the total cost for set union and intersection is $O(((n-v)\epsilon + v)r)$.

The total query cost is $O(rb + ((n-v)\epsilon + v)R)$. Particularly, in our CSC framework, the b partitions we need to check in repetition are either continuous or keep good locality in storage. Therefore, our CSC framework needs fewer memory accesses and enjoys faster query throughput. For example, CSC-BF only needs $k \cdot \lceil \frac{b}{w} \rceil$ memory accesses in one repetition (w is the length of a machine word), which is much fewer than RAMBO's kb random memory accesses.



(a) $b = 120$ (b) $v = 1$

Figure 4: (RCV1-X) Performance gap Δ between CSC-BF and ShBF w.r.t repetition r under different settings.

5.3 Comparison with Shifting Bloom Filter

We notice that a recent work Shifting Bloom Filter (ShBF) [45, 46] shares a similar idea with CSC-BF. The basic structure of ShBF is a Bloom Filter, and it is proposed for multiplicity queries for one set, in which elements may appear more than once. ShBF achieves this by mapping an element to an anchor location and encoding its multiplicity information as an offset. Particularly, ShBF can also be adapted for MS-MMQ by encoding set information as an offset. From this point of view, ShBF is an instance of CSC-BF for $r = 1$ and $b \to +\infty$. Nonetheless, there are still differences. In this part, we aim to provide a detailed discussion to show the differences.

• **There exists no *Repeating and Merging* in ShBF.** The *Repeating and Merging* technique is an important component of CSC and achieves considerable gain in performance especially for small v. To demonstrate this, we perform a theoretical comparison under the same filter size. According to our analysis and [45, 46], the false positive rate of ShBF is $\epsilon_{\text{ShBF}} = \epsilon_{\text{bf}}(rm, k, \sum_{i=0}^{n-1}|S_i|)$, Define Δ as the ratio of ϵ_{ShBF} and the upper bound of ϵ in Equation (6),

$$\Delta = \frac{\epsilon_{\text{bf}}(rm, k, \sum_{i=0}^{n-1}|S_i|)}{\left(1 - \left(1 - \epsilon_{\text{bf}}(m, k, \sum_{i=0}^{n-1}|S_i|)\right)\left(1 - \frac{1}{b}\right)^{v}\right)^{r}}. \tag{9}$$

$\Delta > 1$ means CSC-BF achieves a lower false positive rate than ShBF after applying *Repeating and Merging*. Since there are many parameters in Δ, without loss of generality, we use statistics of dataset RCV1-X as an instance, fixing $k = 3, \sum_{i=1}^{n}|S_i| = 3,708,886, m = 40,000,000$ and varying r, b, v. Figure 4 summarizes the results.

With more repetitions, CSC-BF achieves better performance than ShBF. In Figure 4(a), we set $b = 120$ and vary r under different v. We can see: (1) CSC-BF with *Repeating and Merging* reports much less false positives for small v. For example, when $v = 2$ and $r = 2$, the false positive rate of CSC-BF is $\Delta = 2.3$ times lower than ShBF. When $r = 4$, Δ increases to 342.3, which verifies the effectiveness of *Repeating and Merging*. (2) when r is fixed and with v increases, Δ decreases and may be less than 1 at some settings. This is foreseen and a detailed analysis is in Section 6.4. Fortunately, in many real-world datasets, the vast majority of elements have relatively small multiplicity. For example, elements exist in on average 4.76 out of 2,456 sets for dataset RCV1-X and in on average 3.53 out of 14,588 sets for dataset AmazonCat. When querying these elements, CSC-BF is more superior. In Figure 4(b), we set $v = 1$ and vary r under different b. We can see that under the same repetitions, more partitions bring lower false positive rates. For example, $\frac{\Delta_{b=240}}{\Delta_{b=120}} = 2.2$ at $r = 4$, which is a significant performance gain. Moreover, Figure 4(b) also indicates that CSC-BF is more accurate in Multi-Set Membership Query (i.e., assuming $v = 1$) than ShBF. Based on the above analysis, CSC-BF is a generalization and improvement of ShBF, leading to considerably better performance.

• **We give another instance CSC-CF.** We also successfully apply our CSC framework on the Cuckoo Filter and propose CSC-CF, which supports element deletions and utilizes storage space more efficiently than CSC-BF. While Yang's work focuses more on other sketch structures such as Count-Min for frequency estimation.

5.4 Discussions

• **Impact of set size distribution.** The expected number of false positive sets in our CSC framework is related with the following factors: 1) Total number of elements, i.e., $\sum_{i=0}^{n-1}|S_i|$; 2) Parameters of the AMQ sketch, e.g. size of the Bloom Filter, length of a fingerprint in the Cuckoo Filter; 3) The number of sets in which a query q actually exists. We see that **the false positive rate of CSC-BF and CSC-CF is irrelevant with set size distribution**. According to analysis in Section 3.3, both BIGSI and RAMBO tend to yield false positives on Bloom Filters with large load factors. While the false positive rate of the CSC framework is only affected by the number of inserted elements. Therefore, it **handles different query distribution more effectively**. According to analysis in Section 5.1, the only factor related to query distribution in the CSC framework is the number of sets containing q, which is irrelevant with cardinalities of these sets. However, the performance of BIGSI and RAMBO is affected by the cardinalities of sets containing q. For example, when a query q exists in sets or partitions with small cardinalities, it is much more probably that sets or partitions with large cardinalities becomes false positives, as analyzed in Section 3.3.

• **Insertion-vs-Query tradeoff.** BIGSI allocates n m-bit vectors and needs $O(n)$ memory accesses in the query phase. Instead, BIGSI can also *group bits "vertically"*. That is, allocating m n-bit vectors. In this case, BIGSI needs $O(\lceil n/w \rceil)$ memory accesses (w is the size of a machine word) and is as fast as CSC-BF in query phase. On the one hand, it does not benefit the accuracy of BIGSI. On the other hand, it needs to know n in advance and loses the flexibility to deal with emerging sets to *group bits "vertically"* . Therefore, *grouping bits "vertically"* is suitable only for read-only workloads. By contrast, CSC-BF naturally groups bits "vertically" and deals with emerging sets effectively. Therefore, CSC-BF is suitable for both read-only and write-heavy workloads with higher accuracy.

• **CSC-BF vs CSC-CF.** We compare CSC-BF and CSC-CF from two aspects: (1) **Accuracy.** CSC-CF stores the fingerprints of elements in buckets, while CSC-BF hashes an element into k bits in a bit array. A Cuckoo Filter achieves a lower false positive rate than a Bloom Filter under the same memory usage [16]. Similarly, CSC-CF is more accurate than CSC-BF when allocated the same amount of memory, as will be shown in Section 6.7. (2) **Query Speed.** CSC-BF performs bitwise "*and*" operations to find partitions containing q, while CSC-CF probes buckets and compares fingerprints sequentially. Therefore, CSC-BF achieves a faster query speed than CSC-CF. Moreover, CSC-CF supports element deletions dynamically. In scenarios which emphasize query accuracy and fully dynamic update, we recommend CSC-CF. In scenarios which draw more attention to query efficiency such as online query, we recommend CSC-BF.

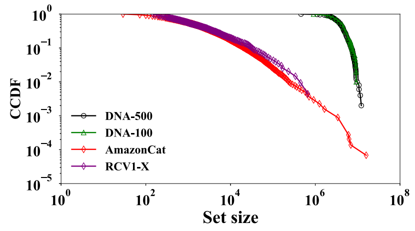


Figure 5: Set size distribution of datasets. CCDF stands for complementary cumulative distribution function.

6 EVALUATION

In this section, we evaluate the performance of the methods. Algorithms are implemented in C++ and run on a Linux server with 2 Quad-Core Intel(R) Xeon(R) Gold 6140 CPU @2.30GHz processors.

6.1 Datasets

We evaluate the performance on four datasets. **RCV1-X** [28] and **AmazonCat** [24] are real-world datasets for multi-label classification[1]. **RCV1-X** consists of categorized newswire stories. Each data point is a story and each story may belong to different categories. **AmazonCat** is a product to product recommendation dataset. Each data point is a product and each product may belong to different categories. We treat points with the same label as a set.

Another two datasets are from NCBI Assembly Resource database[2], including 161,023 individual genome assemblies from GenBank [11] and RefSeq [29]. Each assembly (set) contains genome sequences. Due to the huge volume (170TB), we uniformly sample 100 and 500 sets separately, marked as **DNA-100** and **DNA-500**. Detailed statistics of these datasets are summarized in Table 1. The set distribution of four datasets is in Figure 5. We observe that all datasets have unbalanced distribution on different levels.

Table 1: Datasets in experiments. "#sets" is the number of sets. Max.ele (resp. Min-ele) is the maximum (resp. minimum) number of elements a set contains. Avg.set is the average number of sets an element belongs to.

Datasets	#sets	Max.ele	Min.ele	Ave.set	Size
RCV1-X	2,456	2,106,588	148	4.76	0.8GB
AmazonCat	14,588	15,890,312	29	3.53	4GB
DNA-100	100	9,286,278	913,912	1.00	13GB
DNA-500	500	12,207,582	461,136	1.04	63GB

6.2 Metrics

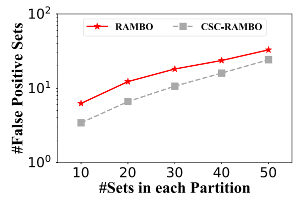
We use the following metrics. 1) ***the number of false positives* (#FPs)**: For a query q in v sets, #FPs is the number of remaining sets reporting existence of q; 2) ***insertion/query speed***: Time for inserting/querying an element. Results are averaged over all queries.

6.3 Comparing Methods and Settings

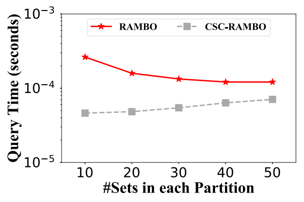
We list all methods used for comparison:

• **BF-based methods:** (1) *ShBF.* (2) *BIGSI.* (3) *RAMBO.* (4) *CSC-BIGSI*: CSC-BF with $r = 1$. (5) *CSC-RAMBO*: CSC-BF with $r \geq 2$ repetitions and b partitions. (6) *Sequence Bloom Tree* (SBT).

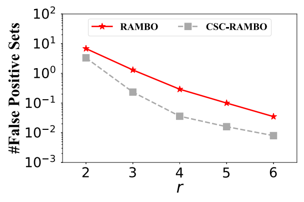
[1]Downloaded from http://manikvarma.org/downloads/XC/XMLRepository.html
[2]https://ftp.ncbi.nlm.nih.gov/genomes/all.



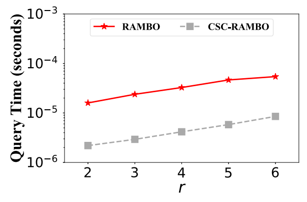
(a) effect of b when $r = 3$ on accuracy



(b) effect of b when $r = 3$ on efficiency

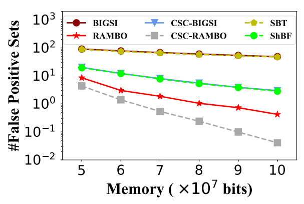


(c) effect of r when $b = \frac{n}{10}$ on accuracy

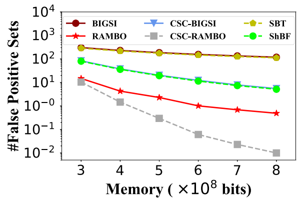


(d) effect of r when $b = \frac{n}{10}$ on efficiency

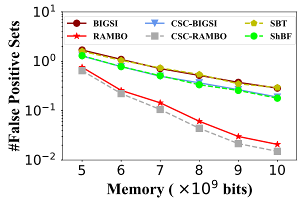
Figure 6: (RCV1-X) Accuracy and efficiency of RAMBO and CSC-RAMBO when under different parameters.



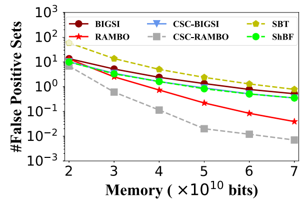
(a) RCV1-X



(b) AmazonCat

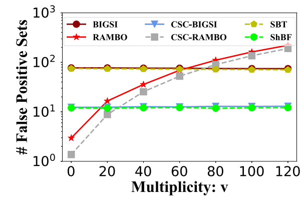


(c) DNA-100

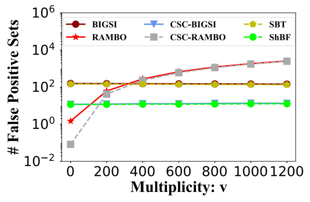


(d) DNA-500

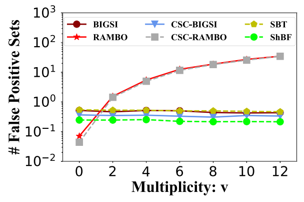
Figure 7: (BF-based methods) Average number of false positives of methods when given different memory spaces.



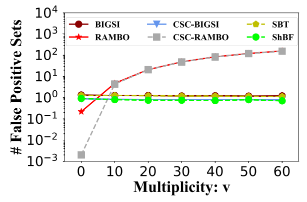
(a) RCV1-X



(b) AmazonCat



(c) DNA-100



(d) DNA-500

Figure 8: (BF-based methods) Average number of false positives of methods when queries have different multiplicities.

• **CF-based methods:** (1) *Naive Cuckoo Filter* (NCF): NCF allocates each set an identical Cuckoo Filter. (2) *CSC-NCF*: CSC-CF with $r = 1$. (3) *CSC-CF*: CSC-CF with $r \geq 2$ repetitions and b partitions.

We compare the above methods within separate categories under the same memory usage. For SBT, we only consider the amount of memory for BFs on leaf nodes. Next, we set up the parameters. For BF-based methods, we fix the number of hash functions in each Bloom Filter as $k = 3$. When total memory size is given, the sizes of Bloom Filters in BIGSI and CSC-BIGSI are also determined. For RAMBO and CSC-RAMBO, we need to determine the number of partitions b and repetitions r. Since [18] does not give a guideline, we perform experiments on dataset RCV1-X and the results are in Figure 6. We generate 1,000 elements, each of which is randomly inserted into 8 sets ($v = 8$), and use them as queries.

In Figure 6(a) and (b), we vary the average number of sets in each partition (i.e, n/b). As b decreases, #FPs increases because the size of M_q^j is larger, yielding more false positives. Meanwhile, the query speed of RAMBO is higher because of fewer Bloom Filters to check. While the query speed of CSC-RAMBO is slightly lower because with each partition containing more sets, it costs more time for the set union to generate $\hat{M}_q^j$, which is its computational overhead. Based on above analysis, we set $b = n/10$. In Figure 6(c) and (d), we vary r from 2 to 6. Both methods gain better performance as r increases and larger r leads to lower query speed because it costs more time for the intersection across $\hat{M}_q^j, 0 \leq j \leq r - 1$ to obtain $\hat{M}_q$. We set $r = 3$ since #FPs decreases fastest.

For CF-based methods, we set the number of slots in each bucket $a = 4$ and the maximum iterations of *eviction* process is 500 [16]. For NCF, we compute the number of buckets necessary to accommodate all elements for the set containing most elements. Then, we allocate each set with this identical CF. For CSC-NCF and CSC-CF, we compute the number of buckets necessary to accommodate elements in all sets. r and b for CSC-CF are set as CSC-BF. All methods vary the fingerprint length f to guarantee the same memory usage.

6.4 Accuracy of BF-based Methods

• **Impact of Memory Usage.** We generate 1,000 unseen elements as queries. Results are in Figure 7. With the increase of the memory budget, all methods gain better performance. The #FPs of RAMBO and CSC-RAMBO is smaller than BIGSI and CSC-BIGSI. For example, #FPs of CSC-RAMBO is 4.6 times smaller than CSC-BIGSI when allocated 5×10^7 bits on RCV1-X. Moreover, CSC-BF reports fewer false positives than BIGSI and RAMBO, which indicates that our CSC framework utilizes memory space more effectively. For example, when allocated 5×10^7 bits, CSC-BIGSI has 4.5 times fewer FPs than BIGSI on RCV1-X. When allocated 8×10^8 bits, CSC-RAMBO

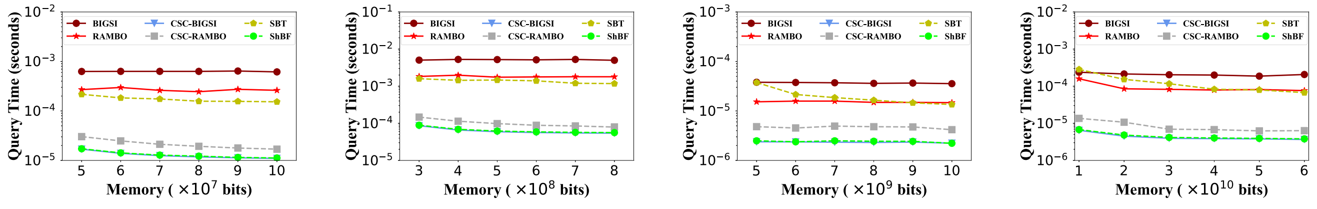

(a) RCV1-X (b) AmazonCat (c) DNA-100 (d) DNA-500

Figure 9: (BF-based methods) Average query time per element when given different memory spaces.

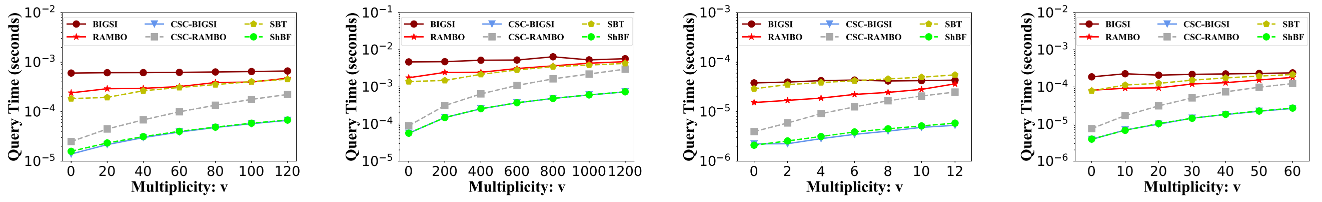

(a) RCV1-X (b) AmazonCat (c) DNA-100 (d) DNA-500

Figure 10: (BF-based methods) Average query time per element when queries have different multiplicities.

reports 48.9 times fewer #FPs than RAMBO on AmazonCat. While on DNA sets, the improvement is not that much because they are not as heavily skewed as the other two datasets.

SBT is as accurate as BIGSI because they both allocate a homogeneous Bloom Filter for each set. Our methods CSC-BIGSI and CSC-RAMBO are much more accurate than SBT. For example, on dataset RCV1-X, CSC-BIGSI (resp. CSC-RABMO) reports 4.4 (resp. 20.2) to 15.8 (resp. 1153.2) times fewer false positive sets than SBT.

For CSC-BF and ShBF, as shown in Figure 7, CSC-BIGSI exhibits similar accuracy as ShBF because they are both instances of CSC-BF for $r = 1$ and $b =\longrightarrow +\infty$. While CSC-RAMBO achieves better accuracy. For example, on dataset RCV1-X, CSC-RAMBO reports 4.5 to 69.3 times fewer false positive sets than ShBF.

• **Impact of Multiplicity.** We randomly insert 1,000 unseen elements into v sets and use them for query. The memory usage for compared methods is the same within datasets. Results are in Figure 8. All datasets show a similar trend. CSC-BIGSI and ShBF report fewer false positives than BIGSI and SBT on all datasets. On RCV1-X and AmazonCat, there are about 10 times more #FPs for BIGSI than CSC-BIGSI due to the high memory efficiency of the CSC framework. Moreover, the performance of SBT, BIGSI, ShBF, and CSC-BIGSI is not sensitive to element multiplicity, which is consistent with the analysis in Section 5.4.

For RAMBO and CSC-RAMBO, their #FPs increases with v because of random partition. As v increases, an element appears in most partitions and candidate set $\hat{M}_q^j$ of repetition R_j may contain most of the n sets, yielding more false positives even after taking intersection across repetitions to generate $\hat{M}_q$.

6.5 Efficiency of BF-based Methods

• **Impact of Memory Usage.** The settings are the same as its counterpart in Section 6.4. Experimental results are shown in Figure 9.
(1) Query speed of SBT, BIGSI, and RAMBO is not sensitive to memory size. This is because the cost for checking Bloom Filters is the main computational overhead, and it is irrelevant with memory size. Moreover, for SBT, the query process is a top-down traversal from the root node to leaf nodes, thus it checks fewer Bloom Filters than BIGSI and is faster, even comparable with RAMBO.
(2) Query speed of CSC-BIGSI and CSC-RAMBO is slightly faster with the increase of memory budget. Locations of 1s in the bit vector by bitwise "and" operation record the sets or partitions that may contain q in CSC-BF. With the increase of memory budget and decrease of false positive sets, there are fewer bits of 1s in this bit vector and CSC-BF costs less time to detect these locations.
(3) CSC framework effectively improves the query efficiency. Both CSC-BIGSI and CSC-RAMBO are faster than BIGSI, RAMBO and SBT. For example, on dataset RCV1-X, CSC-BIGSI is about 49 times faster than BIGSI and CSC-RAMBO is about 8 times faster than RAMBO. CSC-BIGSI (resp. CSC-RAMBO) is about 12.6 (resp. 7.2) times faster than SBT. This is mainly due to fewer memory accesses and bitwise "and" operation of CSC framework.
(4) CSC-BIGSI (resp. ShBF) is faster than CSC-RAMBO. Compared with CSC-BIGSI (resp. ShBF), CSC-RAMBO costs less time on checking in which partitions a query q exists since $rb < n$. However, CSC-RAMBO needs to perform set union and intersection operations to obtain $\hat{M}_q$ afterward, which is the computational bottleneck. CSC-BIGSI (resp. ShBF) does not have these extra operations and thus is faster. For example, on dataset AmazonCat, CSC-BIGSI (resp. ShBF) is about 3.7 times faster than CSC-RAMBO.

• **Impact of Multiplicity.** The settings are the same as in Section 6.4. Experimental results are in Figure 10. We observe that
(1) All datasets show a similar trend. CSC-BIGSI are the fastest among all methods. CSC-RAMBO is faster than ShBF at small multiplicities while ShBF is more efficient at large multiplicities. SBT is approximately as fast as RAMBO, and BIGSI is basically the slowest.
(2) BIGSI is basically not sensitive to the element multiplicity. For BIGSI, the time for checking n Bloom Filters is the main computational cost, which is irrelevant with v.

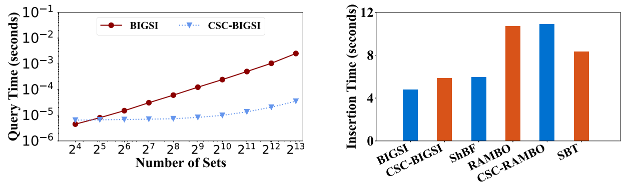


Figure 11: Left: (AmazonCat) Average query time when varying the number of querying sets. Right: (RCV1-X) Insertion speed of BF-based methods under the same memory usage.

(3) SBT, ShBF, and CSC-BIGSI are slightly slower with the increase of multiplicity. For SBT, this is because there are more internal nodes and leaf nodes SBT needs to be traversed, costing more time. For ShBF and CSC-BIGSI, their time for memory accesses is relevant to multiplicity. But there are more true positive sets, and they need more time to detect.

(4) Query speed of CSC-RAMBO and RAMBO becomes slower with the increase of multiplicity. As v increases, the number of partitions reporting existence of query q also increases, which costs more time for set union to generate $\hat{M}_q^j$ of each repetition R_j and set intersection across $\hat{M}_q^j, 0 \le j \le r-1$ to obtain $\hat{M}_q$. Meanwhile, the query time of CSC-RAMBO is approaching that of RAMBO. This is because the time for set operation becomes dominant in the query phase and two methods cost about the same time in this aspect.

• **Impact of the number of sets .** We use the dataset AmazonCat to study the influence of the number of sets. That is, after the insertion of AmazonCat, we use 1,000 non-existing elements to query their existence in different numbers of sets. Figure 11 shows the result. Since query time of CSC-RABMO and RAMBO is independent of the number of sets once r and b are fixed, we only discuss BIGSI and CSC-BIGSI. For BIGSI, its query time is linear with the number of sets. For CSC-BIGSI, due to fewer memory accesses, its query time increases much slower than BIGSI.

• **Insertion Speed.** We evaluate insertion cost on RCV1-X in Figure 11. Since our CSC framework encodes set information additionally, which leads to a little more insertion time but is still acceptable. Meanwhile, the insertion time of CSC-RAMBO and RAMBO is much more than CSC-BIGSI and BISGI. This is because both of the former methods insert an element for r times. ShBF shares the same insertion time with CSC-BIGSI. While SBT costs more time than BIGSI as it needs extra time to construct the binary tree.

6.6 Results on CSC-CF

In CF-based methods, due to the limited space, we evaluate the performance on dataset **DNA-500** and omit similar results on other datasets. According to settings in the last paragraph of Section 6.3, to guarantee a fair comparison under same memory usage, the total number of buckets allocated for NCF is about 3 times that of CSC-NCF. Since $r = 3$ for CSC-CF, then the fingerprint length of CSC-CF and NCF is approximately the same and the fingerprint of CSC-NCF is 3 times longer. Query elements are the same as Section 6.4. Experimental results are summarized in Figure 12.

• **Accuracy.** Figure 12(a) shows the impact of memory usage on #FPs. As the memory budget increases, all methods gain better performance because they use longer fingerprints to encode elements. CSC-CF reports more FPs than NCF when allocated memory is less

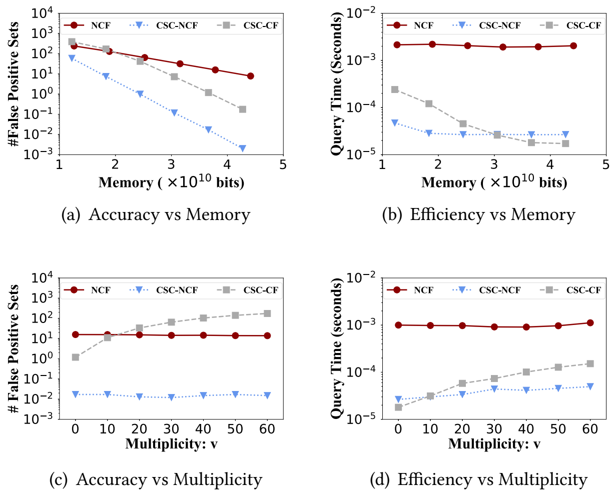


Figure 12: (DNA-500) Performance of CF-based methods.

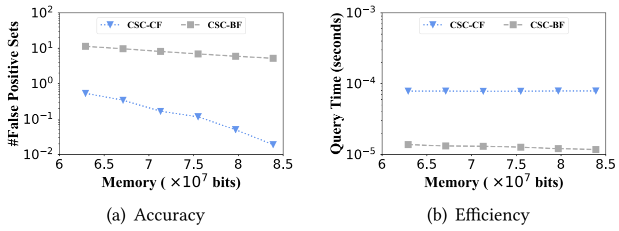


Figure 13: (RCV1-X) Accuracy and efficiency of CSC-CF compared with CSC-BF when given different memory space.

than 2×10^{10} bits because of many partitions incorrectly reporting the existence of q in a repetition. As the memory budget increases, there are fewer such partitions and CSC-CF outperforms NCF. CSC-NCF is more accurate than CSC-CF and NCF because of the longer fingerprint. Figure 12(c) shows the impact of element multiplicity on #FPs. Results are consistent with CSC-BF in Figure 8(d).

• **Efficiency.** Figure 12(b) shows the impact of memory usage on query speed. CSC-NCF and CSC-CF are significantly faster than NCF. For example, when allocated about 3×10^{10} bits, CSC-NCF is about 38 times faster than NCF. For CSC-CF, there are many FPs when allocated less than 2×10^{10} bits, and it costs much time on set union operation to generate $\hat{M}_q^j$ for each repetition R_j and set intersection across $\hat{M}_q^j, 0 \le j \le r-1$ to obtain $\hat{M}_q$. With the increase of memory budget, time for the above set operation decreases and CSC-CF is faster than CSC-NCF when the memory budget is more than 3×10^{10} bits. Figure 12(d) shows the impact of element multiplicity. Results are consistent with CSC-BF in Figure 10(d).

6.7 Comparison between CSC-BF and CSC-CF

We compare CSC-BF and CSC-CF on RCV1-X and omit similar results on other datasets. We set $r = 1$ for both methods and generate 1,000 non-existing elements for query. Figure 13 shows the experimental results. From Figure 13(a), we observe that CSC-CF reports fewer false positives than CSC-BF under all memory settings and the gap becomes larger with the increase of memory budget. For example, when allocated about 6.3×10^7 bits, CSC-BF reports 21.2 times more #FPs than CSC-CF. This number increases to 274.9 when allocated about 8.4×10^7 bits. Meanwhile, CSC-BF achieves higher query efficiency. In Figure 13(b), we find CSC-BF is about 5.7 times faster during query phase. Results on accuracy and efficiency are consistent with our analysis in Section 5.4.

7 RELATED WORK

• **Approximate Membership Querying (AMQ).** The *Bloom Filter* [3] is the most well-known for AMQ. It has many variants [1, 12, 15, 17, 30, 35, 47]. For example, the *Blocked Bloom Filter* [30] uses a series of small Bloom Filters, each of them fit into one CPU cache line. It is cache-efficient and achieves faster speed because of one cache miss for each query. Other variants include the *Counting Bloom Filter* [17], which supports element deletions by replacing each bit with a counter, and the *Deletable Bloom Filter* [34], which uses an *extra collision bitmap* to delete elements with a probability. For more variants, see [6] and [23] for a comprehensive view. Kraska et al. [21] propose *Learned Bloom Filter* (LBF) to reduce space cost of the Bloom Filter by machine learning. LBF consists of an ML model as a pre-filter and a back-up Bloom Filter. The ML model scores a query q as the probability q has appeared. If the score is above a threshold, LBF reports a positive result. Otherwise, the back-up Bloom Filter will judge whether q has appeared. LBF is memory-efficient and reduces the false positive rate. Mitzenmacher [25] proposed *Sandwiched LBF*, adding another BF before LBF to improve the performance. Rae et al. [32] proposed *Neural Bloom Filter* to deal with streaming sets via memory-augmented neural networks and meta-learning, which exploits data distribution effectively.

The *Cuckoo Filter* [16] and its variants are also widespread. Detailed introduction of the Cuckoo Filter is in Section 3.1.2. The Cuckoo Filter also has variants. *Dynamic Cuckoo Filter* [9] regards a Cuckoo Filter as a basic building block and uses a series of linked homogeneous Cuckoo Filters to extend capacity at the cost of a higher false positive rate. *Adaptive Cuckoo Filter* [26] reduces false positive rates by resetting a stored fingerprint with optional hash functions when a false positive is detected. *D-ary Cuckoo Filter* [44] reduces the memory usage of the Cuckoo Filter by increasing the number of candidate buckets for each element. *Morton Filter* [5] improves throughput by introducing compression, sparsity, and biasing techniques. Recently, Wang et al. proposed *Vacuum Filter* [40] to further improve memory efficiency.

• **Approximate MS-MMQ.** *Inverted Index* [43] exactly solves this problem by constructing a list of set IDs for each element. However, it costs lots of memory space. In bioinformatics, *Sequence Bloom Tree* (SBT) [36] is the first scalable method for searching raw sequences in archived datasets. However, both SBT and its successor *Split SBT* [37] fail when similarities between sets are low and the total number of sets is large. Then, Bradley et al. [4] propose *BIGSI* to address approximate MS-MMQ for DNA sequence. BIGSI builds a homogeneous Bloom Filter for each set and generates "*BItsliced Signature Index*" for fast retrieve of a query. BIGSI suffers from low memory efficiency and query speed when the number of sets goes larger. To address these problems, Gupta et al. [18] propose *Repeated and Merged Bloom Filter* (RAMBO) which combines the Bloom Filter and *Count-Min sketch* [13]. Specifically, they replace each counter of a Count-Min sketch with a Bloom Filter, hence acquiring a compact sketch and supporting cheap streaming updates. However, such a combination may neither be computationally- nor memory-efficient, as analyzed in Section 3.

•**Approximate MS-MQ.** MS-MQ assumes each element only exists in one set, i.e., there is no intersection between different sets. *BUFFALO* [49] assigns each set a Bloom Filter and needs to access multiple Bloom Filters for a query. Clearly, it suffers from memory inefficiency and low query speed. To reduce the number of Bloom Filters, *Bloom Tree* [48] and *Coded Bloom Filter* [8] let each Bloom Filter represent a part of encoded set IDs. There are variants which record elements of different sets in one Bloom Filter such as the *Combinatorial Bloom Filter* [20], which uses only one Bloom Filter with multiple groups of hash functions. Each set is mapped into the Bloom Filter by its corresponding group of hash functions. Other analogous work includes *iSet* [31] and *Noisy Bloom Filter* [14]. Sun et al. [38] propose *Magic Cube Bloom Filter*, which stores elements from the same set in different Bloom Filters and thus improves the accuracy by the redistribution of elements. Yang et al. [39] propose *Coloring Embedder*, which first maps elements to high dimensional space and then uses a dimensional reduction representation.

The above methods face difficulties if applied to MS-MMQ. For example, *Bloom Tree* [48] faces the same dilemma as SBT. *Coded Bloom Filter* [8] uses a fixed-length bit string to encode a set and allocates a Bloom Filter for each bit string. It yields lots of false positives if different sets collide at the same bit string. *Combinatorial Bloom Filter* [20] uses $\log n$ bits to encode the query result set, which is enough only for one membership set and fails if an element exists in multiple sets. *Coloring Embedder* [39] faces the same dilemma. Similar to BIGSI [4], *Magic Cube Bloom Filter* [38] enumerates each set for a query, requiring numbers of memory accesses if the number of sets goes larger. Moreover, it cannot handle emerging sets.

8 CONCLUSIONS AND FUTURE WORK

To deal with the skew set distribution and address MS-MMQ effectively and efficiently, in this paper, we propose a CSC framework, which can be integrated with mainstream sketches for approximate MS-MMQ. Based on CSC, we use the well-known Bloom Filter and Cuckoo Filter as examples and propose two methods, CSC-BF and CSC-CF. Both theoretical analysis and experimental results show that our CSC framework can effectively and efficiently handle skew set distribution and achieve both slower query time and fewer errors compared with state-of-the-art algorithms. In the future, we aim to promote our framework from two aspects: (1) Incorporating machine learning techniques to further reduce the space cost as well as improving the performance by further utilize set distribution. (2) Designing more effective partition schemes via *Group testing*.

ACKNOWLEDGMENTS

This research is supported in part by National Natural Science Foundation of China (61922067, U1736205, 61902305), Shenzhen Basic Research Grant (JCYJ20170816100819428), MoE-CMCC "Artifical Intelligence" Project (MCM20190701), Natural Science Basic Research Plan in Shaanxi Province of China (2019JM-159), Natural Science Basic Research Plan in Zhejiang Province of China (LGG18F020016).

REFERENCES

[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable bloom filters. *Inform. Process. Lett.* 101, 6 (2007), 255–261.

[2] Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. 2012. Don't thrash: how to cache your hash on flash. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1627–1637.

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

[4] Phelim Bradley, Henk C den Bakker, Eduardo PC Rocha, Gil McVean, and Zamin Iqbal. 2019. Ultrafast search of all deposited bacterial and viral genomic data. *Nature biotechnology* 37, 2 (2019), 152.

[5] Alex D Breslow and Nuwan S Jayasena. 2018. Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment* 11, 9 (2018), 1041–1055.

[6] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[8] Francis Chang, Wu-chang Feng, and Kang Li. 2004. Approximate caches for packet classification. In *IEEE INFOCOM*, Vol. 4. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 2196–2207.

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

[10] Ken Christensen, Allen Roginsky, and Miguel Jimeno. 2010. A new analysis of the false positive rate of a bloom filter. *Inform. Process. Lett.* 110, 21 (2010), 944–949.

[11] Karen Clark, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and Eric W Sayers. 2015. GenBank. *Nucleic acids research* 44, D1 (2015), D67–D72.

[12] Saar Cohen and Yossi Matias. 2003. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 241–252.

[13] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[14] Haipeng Dai, Yuankun Zhong, Alex X Liu, Wei Wang, and Meng Li. 2016. Noisy bloom filters for multi-set membership testing. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 44. ACM, 139–151.

[15] Fan Deng and Davood Rafiei. 2006. Approximately detecting duplicates for streaming data using stable bloom filters. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 25–36.

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

[17] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 1998. Summary cache: A scalable wide-area web cache sharing protocol. In *ACM SIGCOMM Computer Communication Review*, Vol. 28. ACM, 254–265.

[18] Gaurav Gupta, Benjamin Coleman, Tharun Medini, Vijai Mohan, and Anshumali Shrivastava. 2019. RAMBO: Repeated And Merged Bloom Filter for Multiple Set Membership Testing (MSMT) in Sub-linear time. *arXiv preprint arXiv:1910.02611* (2019).

[19] Gaurav Gupta, Minghao Yan, Benjamin Coleman, RA Elworth, Todd Treangen, and Anshumali Shrivastava. 2019. Sub-linear sequence search via a Repeated And Merged Bloom Filter (RAMBO): indexing 170 TB data in 14 hours. *arXiv* (2019), arXiv–1910.

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

[21] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 489–504.

[22] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.

[23] Lailong Luo, Deke Guo, Richard TB Ma, Ori Rottenstreich, and Xueshan Luo. 2018. Optimizing Bloom filter: Challenges, solutions, and comparisons. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1912–1949.

[24] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. 2015. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 43–52.

[25] Michael Mitzenmacher. 2018. A model for learned bloom filters and optimizing by sandwiching. In *Advances in Neural Information Processing Systems*. 464–473.

[26] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. 2018. Adaptive cuckoo filters. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 36–47.

[27] James K Mullin. 1983. A second look at Bloom filters. *Commun. ACM* 26, 8 (1983), 570–571.

[28] Yashoteja Prabhu and Manik Varma. 2014. Fastxml: A fast, accurate and stable tree-classifier for extreme multi-label learning. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 263–272.

[29] Kim D Pruitt, Tatiana Tatusova, and Donna R Maglott. 2006. NCBI reference sequences (RefSeq): a curated non-redundant sequence database of genomes, transcripts and proteins. *Nucleic acids research* 35, suppl_1 (2006), D61–D65.

[30] Felix Putze, Peter Sanders, and Johannes Singler. 2007. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*. Springer, 108–121.

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

[32] Jack Rae, Sergey Bartunov, and Timothy Lillicrap. 2019. Meta-Learning Neural Bloom Filters. In *International Conference on Machine Learning*. 5271–5280.

[33] Vadim A Raikhlin and Roman K Klassen. 2020. Clusterix-Like BigData DBMS. *Data Science and Engineering* 5, 1 (2020), 80–93.

[34] Christian Esteve Rothenberg, Carlos AB Macapuna, Fábio L Verdi, and Mauricio F Magalhaes. 2010. The deletable Bloom filter: a new member of the Bloom family. *IEEE Communications Letters* 14, 6 (2010), 557–559.

[35] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. 2014. The variable-increment counting Bloom filter. *IEEE/ACM Transactions on Networking (TON)* 22, 4 (2014), 1092–1105.

[36] Brad Solomon and Carl Kingsford. 2016. Fast search of thousands of short-read sequencing experiments. *Nature biotechnology* 34, 3 (2016), 300–302.

[37] Brad Solomon and Carl Kingsford. 2017. Improved search of large transcriptomic sequencing databases using split sequence bloom trees. In *International Conference on Research in Computational Molecular Biology*. Springer, 257–271.

[38] Zhouyi Sun, Siang Gao, Bingqing Liu, Yufei Wang, Tong Yang, and Bin Cui. 2019. Magic Cube Bloom Filter: Answering Membership Queries for Multiple Sets. In *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 1–8.

[39] Yang Tong, Dongsheng Yang, Jie Jiang, Siang Gao, Bin Cui, Lei Shi, and Xiaoming Li. 2019. Coloring Embedder: a Memory Efficient Data Structure for Answering Multi-set Query. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1142–1153.

[40] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. 2019. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment* 13, 2 (2019), 197–210.

[41] Pinghui Wang, Peng Jia, Jing Tao, and Xiaohong Guan. 2018. Detecting a Variety of Long-Term Stealthy User Behaviors on High Speed Links. *IEEE Transactions on Knowledge and Data Engineering* 31, 10 (2018), 1912–1925.

[42] Pinghui Wang, Peng Jia, Jing Tao, and Xiaohong Guan. 2018. Mining long-term stealthy user behaviors on high speed links. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2051–2059.

[43] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

[44] Zhuohan Xie, Wencheng Ding, Hongya Wang, Yingyuan Xiao, and Zhenyu Liu. 2017. D-Ary Cuckoo Filter: A space efficient data structure for set membership lookup. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 190–197.

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

[46] Tong Yang, Alex X Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. 2016. A Shifting Bloom Filter Framework for Set Queries. *Proceedings of the VLDB Endowment* 9, 5 (2016).

[47] MyungKeun Yoon. 2009. Aging bloom filter with two active buffers for dynamic sets. *IEEE Transactions on Knowledge & Data Engineering* 1 (2009), 134–138.

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

[49] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. 2009. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 313–324.