Additive and Subtractive Cuckoo Filters

Kun Huang[†*], Tong Yang[‡]

[†]*Southern University of Science and Technology*
Shenzhen, China
huangk@sustech.edu.cn

[*]*Peng Cheng Laboratory*
Shenzhen, China

[‡]*Peking University*
Beijing, China
yang.tong@pku.edu.cn

***Abstract*—Bloom filters (BFs) are fast and space-efficient data structures used for set membership queries in many applications. BFs are required to satisfy three key requirements: low space cost, high-speed lookups, and fast updates. Prior works do not satisfy these requirements at the same time. The standard BF does not support deletions of items and the variants that support deletions need additional space or performance overhead. The state-of-the-art cuckoo filters (CF) has high performance with seemingly low space cost. However, the CF suffers a critical issue of varying space cost per item. This is because the exclusive-OR (XOR) operation used by the CF requires the total number of buckets to be a power of two, leading to the space inflation. To address the issue, in this paper we propose a scalable variant of the cuckoo filter called additive and subtractive cuckoo filter (ASCF). We aim to improve the space efficiency while sustaining comparably high performance. The ASCF uses the addition and subtraction (ADD/SUB) operations instead of the XOR operation to compute an item's two candidate bucket indexes based on its fingerprint. Experimental results show that the ASCF achieves both low space cost and high performance. Compared to the CF, the ASCF reduces up to 1.9x space cost per item while maintaining the same lookup and update throughput. In addition, the ASCF outperforms other filters in both space cost and performance.**

Keywords—Bloom filters, cuckoo filters, cuckoo hashing

I. INTRODUCTION

Bloom filters (BFs) [1] are space-efficient probabilistic data structures for high-speed approximate set membership queries. This query is to answer whether a given item is in a set or not. BFs have high space efficiency as they represent a set of items with a constant number of bits. They have a small probability of false positives (i.e., an item is reported to be in the set although it is not), but have no false negatives. BFs have been widely used in network applications [2] and distributed systems [3], such as cooperative caching [4, 5], network processing [6, 7, 8, 9], key-value store [10, 11, 12, 13], and data deduplication [14, 15, 16].

In many applications, BFs are required to satisfy three key requirements: low space cost, high-speed lookups, and fast updates. First, BFs are often implemented using fast but small memory (e.g., SRAM) to avoid unnecessary expensive accesses to slow memory (e.g., DRAM or SSD). Therefore, it is critical to minimize the space cost of BFs as far as possible. Second, BFs must sustain high-speed lookups with the rapid growth of queries. Meanwhile, BFs must support fast incremental updates, which means allowing dynamic insertions and deletions of items without rebuilding the entire filter.

This work was supported by the National Key Research & Development Program of China (2019YFB1802800) and the PCL Project (LZC0019).

However, prior works do not satisfy these requirements at the same time. The standard BF [1] has low space cost and high-speed lookups, but does not support deletions. The counting BF (CBF) [4] is one well-known variant that supports deletions by using counters in place of bits of the standard BF. The CBF has high-speed lookups and fast updates, but requires 4x space cost over the standard BF. Several variants of CBFs [17, 18, 19, 20, 21, 22] have been proposed to improve the space efficiency but at the cost of substantial performance degradation. For instance, the dlCBF [17] reduces a factor of two or more space over the CBF, but suffers slow lookups and updates. The MLCCBF [18] and RCBF [19] use a hierarchical structure to compress the CBF, but require high update overhead for maintaining the hierarchy. The QF [21] and CQF [22] have low space cost and high lookup performance, but degrade the update performance significantly when the filter becomes full.

The cuckoo filter (CF) [23] is a state-of-the-art variant of the CBF. The CF uses cuckoo hashing and fingerprints to achieve higher space efficiency and lookup and update performance than previous variants of CBFs and even the non-deletable standard BF for low false positive rates (i.e., <3% [23]). However, the CF suffers a critical issue of varying space cost per item. This is because the CF uses the exclusive-OR (XOR) operation based on a fingerprint of an item to compute the two candidate bucket indexes. The XOR operation performs fast, but requires that the total number of buckets must be a power of two, which incurs up to 2x space inflation. Therefore, it is challenging for a filter to achieve the scalability in both space cost and performance at the same time.

To address the issue, in this paper we propose a scalable variant of the CF called additive and subtractive cuckoo filter (ASCF). We aim to improve the space efficiency over the CF while sustaining high performance comparable to the CF. The basic idea behind the ASCF is to use the addition and subtraction (ADD/SUB) operations, instead of the XOR operation used by the CF, to compute an item's two candidate bucket indexes. The ADD/SUB operations perform as fast as the XOR operation, and do not require the total number of buckets to be a power of two. Therefore, the ASCF requires lower space cost per item than the CF while maintaining the same high performance as the CF.

We conducted simulation experiments to evaluate the ASCF and compare with previous representative filters, including the standard BF [1], CBF [4], dlCBF [17], RCBF [19], CQF [22], and CF [23]. Experimental results show that the ASCF reduces up to 1.9x space cost per item over the CF while achieving the same lookup and update throughput as the CF. In addition, we

978-1-7281-6887-6/20/$31.00 ©2020 IEEE

show that the ASCF outperforms other filters in both space cost and performance for same false positive rates.

This paper makes two key contributions as follows:

- We propose ASCF, a novel scalable variant of the CF that improves the space efficiency while sustaining high performance. The key to the ASCF is to use the addition and subtraction (ADD/SUB) operations, instead of the XOR operation, to compute the indexes of two candidate buckets for an item's fingerprint. With the ADD/SUB operations, the ASCF has both high space efficiency and lookup and update performance.
- We conducted experiments to compare the ASCF with previous filters. The results show that the ASCF requires up to 1.9x lower space cost per item than the CF while sustaining the same lookup and update throughput as the CF. In addition, the ASCF achieves both higher space efficiency and higher performance than other filters.

The rest of this paper is organized as follows. We overview the background and related work on BFs and their variants in Section II. Section III describes the detailed design of the ASCF. We provide experimental results on ASCF evaluation in Section IV. Finally, Section V concludes this paper.

II. Background and Related Work

In this section, we first present the background on standard Bloom filters, and then review the related work on counting Bloom filters and their variants that support both insertions and deletions of items.

A. Standard Bloom Filters

Standard Bloom filters (BFs) [1] are fast and space-efficient data structures for approximate set membership queries. A BF represents a set of n items using an array of m bits, initially all set to 0. A BF uses k independent hash functions to map an item to a random index uniformly over the range [0, …, m-1]. When an item x is inserted, the BF maps x to k bits by k hash functions $h_0(x)$, …, $h_{k-1}(x)$, and sets all these bits in the filter to 1. When an item y is queried, the BF checks whether y's all k bits are set to 1 or not. If all k bits are set to 1, y is claimed to be in the set. If not, y is certainly not in the set. Standard BFs support insertions and lookups, but not deletions of items.

A BF allows a small false positive rate that an item is claimed to be in the set even though it is not. For a given false positive rate ε, the space-optimized standard BF has the minimum space cost of $1.44\times\log_2(1/\varepsilon)$ bits for an item by using $k=\log_2(1/\varepsilon)$ hash functions. The minimum space cost per item depends on ε, rather than the item size or the number of stored items. The theoretical lower bound of space cost is $\log_2(1/\varepsilon)$ bits per item achieved by using a perfect hash function for a static set [2]. Therefore, there is a gap of 44% in space cost between the space-optimized standard BF and the theoretical lower bound, which motivates one to minimize the space cost per item as far as possible.

B. Counting Bloom Filters and Variants

Counting Bloom filters (CBFs) [4] extend standard BFs to support deletions by using an array of m counters in place of an array of m bits. When an item x is inserted or deleted, the CBF increments or decrements x's all k counters. When an item y is queried, the CBF checks whether y's all k counters are non-zero or not. If all k counters are non-zero, y is claimed to be in the set; otherwise, y is not in the set. We note that counters must be sufficiently large to avoid the overflow. In practice, four bits per counter suffice for most of applications [2]. Therefore, the CBF requires 4x space over the standard BF. Several variants of CBFs have been proposed to improve the space efficiency as follows.

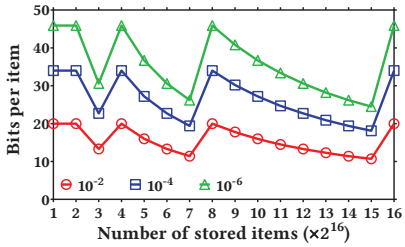


Fig. 1. Space cost per item of CFs varying with different numbers of stored items for target false positive rates 10^{-2}, 10^{-4}, and 10^{-6}.

The d-left CBF (dlCBF) [17] uses d-left hashing and small counters to store and search multiple fingerprints of an item. The dlCBF reduces a factor of two or more space over the CBF. However, the dlCBF has slow lookups because it needs to search up to 24 fingerprints for a query.

The multilayer compressed CBF (MLCCBF) [18] uses a hierarchical structure and Huffman coding to compress the CBF. The MLCCBF requires up to 50% less space than the CBF while achieving the same lookup performance as the CBF. However, the MLCCBF has slow updates because it requires additional update overhead for maintaining the hierarchical structure.

The rank-indexed CBF (RCBF) [19] leverages rank-indexed hashing to construct an index hierarchy for compactly packing an array of fingerprints. The RCBF has high-speed lookups and requires less space than the CBF by a factor of three or more. However, the RCBF suffers slow updates because it needs high update overhead for maintaining the index hierarchy.

The variable-increment CBF (VICBF) [20] exploits variable increments instead of unit increments used by the CBF to update counters. For a query on an item, the VICBF checks whether a variable increment is a part of the sum in one of its counters hashed by the item. The VICBF requires 33% less space than the CBF, but suffers slow lookups and updates because it needs additional overhead to compute a variable increment for each counter.

The quotient filter (QF) [21] uses linear probing to store and search a fingerprint of an item in a compact hash table. The QF leverages additional metadata to accelerate lookups and updates, requiring up to 25% more space than the standard BF. However, the QF suffers significant update performance degradation when the filter becomes full [21].

The counting quotient filter (CQF) [22] improves the QF by using rank-and-select based metadata and counter embedding. The CQF uses the rank and select operations to restructure metadata for improving both the space efficiency and lookup

performance. However, the CQF suffers high update overhead when the table occupancy of the filter becomes large.

The cuckoo filter (CF) [23] uses partial-key cuckoo hashing to store and search a fingerprint of an item. The CF is a compact cuckoo hash table [24] with two hash functions per item and four slots per bucket. For an item, the CF uses a single hash function to compute its fingerprint and the first candidate bucket index. Then, it uses the exclusive-OR (XOR) operation based on the fingerprint to compute the second candidate bucket index. When an existing fingerprint f is relocated for inserting a new item, the CF computes the alternate candidate bucket index j for f by using $j=i \oplus \text{hash}(f)$, where i is the current bucket index of f and $\oplus$ is the XOR operation. Therefore, the CF not only achieves higher lookup and update performance, but also requires less space than previous variants of CBFs and even the non-deletable standard BF for low false positive rates (i.e., <3% [23]).

However, the CF suffers a critical issue of varying space cost per item described above. This is because the XOR operation used by the CF requires that the total number of buckets must be a power of two. Fig. 1 shows the space cost per item of CFs varying with different numbers of stored items for target false positive rates. We see that the ratio between the maximum and minimum bits per item required by the CF is about two, which means inflating up to 2x space cost per item. In this paper, we propose ASCF, a scalable variant of the CF to address the issue. We aim to achieve constant and lower space cost per item while sustaining comparably high performance.

In addition, other variants have been proposed to accelerate the performance by partitioning the filter into an array of small blocks. The blocked BF [25] and one memory access BF [26] construct a small BF in a CPU cache line or word to improve the performance but at the cost of increasing false positive rates. The shifting BF [27] uses the shifting operation to reduce both the number of memory accesses and number of hash computations for a query. The Morton filter (MF) [28] uses three techniques of compression, sparsity, and biasing to improve the lookup and update performance over the CF. These works are orthogonal to our work and can be applied to the ASCF for further improving the performance, which is left as our future work.

III. Additive and Subtractive Cuckoo Filters

We propose a novel scalable variant of the cuckoo filter (CF) called additive and subtractive cuckoo filter (ASCF). We aim to improve the space efficiency while sustaining high lookup and update performance. The basic idea behind the ASCF is to use the addition and subtraction (ADD/SUB) operations instead of the XOR operation to compute the two candidate bucket indexes based on an item's fingerprint. In this section, we first elaborate the insert, lookup, and delete algorithms of the ASCF, and then analyze the time and space complexities.

A. Insert Algorithm

Unlike the CF [23], the ASCF is a blocked compact cuckoo hash table that consists of two blocks each with $m/2$ buckets for representing a set of n items. Each bucket contains four slots, each storing an item's fingerprint instead of its key. The ASCF maps an item to two candidate buckets in two blocks, and inserts its fingerprint into one of the two candidate buckets. When both two candidate buckets are full, the ASCF inserts the fingerprint by relocating existing fingerprints to their alternate buckets.

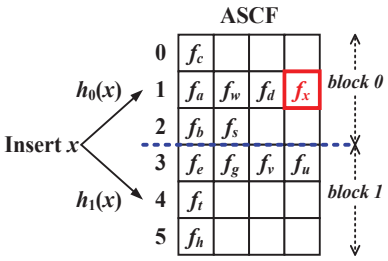


Fig. 2. Insert an item x into an ASCF by placing its fingerprint in a vacant bucket.

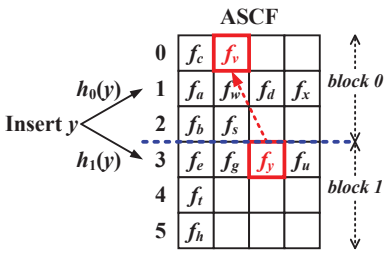


Fig. 3. Insert an item y into an ASCF by relocating existing fingerprints to their alternate buckets.

The ASCF differs from the CF [23] in the computation of the indexes of two candidate buckets. The ASCF uses the addition and subtraction (ADD/SUB) operations, instead of the XOR operation used by the CF, to compute the two candidate bucket indexes for an item's fingerprint. The ADD/SUB operations do not require the total number of buckets in the filter to be a power of two. In addition, these operations perform as fast as the XOR operation. With the ADD/SUB operations, the ASCF therefore improves the space efficiency over the CF while sustaining high performance comparable to the CF.

The insertion procedure of the ASCF is as follows. When an item x is inserted, the ASCF computes x's fingerprint f_x and the two candidate bucket indexes $h_0(x)$ and $h_1(x)$ in two blocks. We first use a single hash function $G(x)$ to compute the fingerprint f_x and the first candidate bucket index $h_0(x)$ in the first block (i.e., block 0) by

$$h_0(x) : f_x = G(x) \tag{1}$$

where f_x is the right digits of the hash value of $G(x)$, $h_0(x)$ is the left digits in the range $[0, \ldots, m/2\text{-}1]$, and ':' is the concatenation character. Then, we compute the second candidate bucket index $h_1(x)$ in the second block (i.e., block 1) for the fingerprint f_x by using the ADD operation based on $h_0(x)$ and f_x as follows:

$$h_1(x) = (h_0(x) + H(f_x)) \bmod m/2 + m/2 \tag{2}$$

where $H(f_x)$ is the hash value in the range $[0, \ldots, m/2\text{-}1]$ and $m/2$ is the number of buckets in each block. We note that $h_1(x)$ is the bucket index in the range $[m/2, \ldots, m\text{-}1]$ of block 1.

When at least one of x's two candidate buckets $h_0(x)$ and $h_1(x)$ has spare capacity, we place the fingerprint f_x in a vacant bucket $h_0(x)$ or $h_1(x)$. When both two candidate buckets $h_0(x)$ and $h_1(x)$

are full, we randomly place the fingerprint f_x in a bucket $h_0(x)$ or $h_1(x)$, and then kick out an existing fingerprint from the bucket to its alternate bucket. The process repeats until a vacant bucket is found or the maximum number of kick-outs (i.e., 500 in our experiments) is reached. To relocate an evicted fingerprint g to its alternate bucket, we compute the alternate bucket index j by using the ADD/SUB operations based on the evicted fingerprint g and the current bucket index i as follows:

$$\begin{cases} j = (i + \mathrm{H}(g)) \bmod m/2 + m/2, & if\ i \in block\ 0 \\ j = (i - \mathrm{H}(g)) \bmod m/2, & if\ i \in block\ 1 \end{cases} \quad (3)$$

where H(g) is the hash value in the range [0, …, m/2-1]. Eq. (3) shows that if the current bucket index i is in block 0, we use the first formula in (3) to compute the alternate bucket index j in block 1; if the current bucket index i is in block 1, we use the second formula in (3) to compute the alternate bucket index j in block 0.

Algorithm 1: ASCF_Insert (x)

```
1: h0 : fx = G(x);
2: h1 = (h0 + H(fx)) mod m/2 + m/2;
3: if bucket h0 or h1 has an empty slot then {
4:     store fx in this bucket;
5:     return true; }
6: h = randomly select a bucket from h0 and h1;
7: f = the fingerprint fx for h;
8: for i = 0; i< MaxKicks; i++ do {
9:     s = randomly select a slot from bucket h;
10:    g = the fingerprint stored in slot s;
11:    store f in slot s and displace g;
12:    if h >= m/2 then {
13:        h = (h - H(g)) mod m/2; }
14:    else {
15:        h = (h + H(g)) mod m/2 + m/2; }
16:    if bucket h has an empty slot then {
17:        relocate g to bucket h;
18:        return true; }
19:    f = the fingerprint g for h; }
20: return false;
```

Algorithm 1 shows the insert algorithm of the ASCF. When inserting an item x into the ASCF, Lines 1 and 2 compute x's fingerprint f_x and the two candidate bucket indexes h_0 and h_1 by (1) and (2). Lines 3 to 5 place the fingerprint f_x in a vacant bucket h_0 or h_1 if at least one of the two candidate buckets h_0 and h_1 has empty slots. Lines 6 to 20 randomly place the fingerprint f_x in a bucket h_0 or h_1 if both two candidate buckets h_0 and h_1 are full, and relocate other existing fingerprints (i.e., g) to their alternate buckets, whose indexes are computed by (3).

Fig. 2 shows an example of inserting an item x into an ASCF by placing its fingerprint in a vacant bucket. We assume that the ASCF consists of two blocks, each containing m/2=3 buckets, where m=6. For x, we first compute the fingerprint f_x and the first candidate bucket index $h_0(x)$=1 in block 0 by (1). Given H(f_x)=0, then we compute the second candidate bucket index $h_1(x)$=4 in block 1 by (2). Finally, we check to find that both two buckets 1 and 4 are vacant, and therefore randomly place the fingerprint f_x in bucket 1.

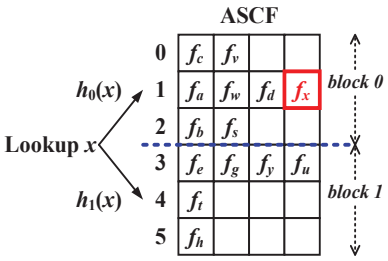


Fig. 4. Lookup an item x against an ASCF by checking whether at least one matching fingerprint is found in its two candidate buckets or not.

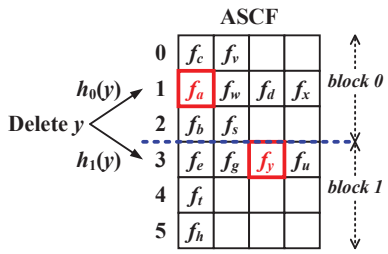


Fig. 5. Delete an item y from an ASCF by removing one copy of matching fingerprints.

Fig. 3 shows another example of inserting an item y into an ASCF by relocating other existing fingerprints to their alternate buckets. For y, we first compute the fingerprint f_y and the two candidate bucket indexes $h_0(y)$=1 and $h_1(y)$=3 by (1) and (2). Because both two buckets 1 and 3 are full, we randomly place f_y in bucket 3 by displacing an existing fingerprint f_v in the bucket. Given H(f_v)=3, then we compute the alternate candidate bucket index $h_0(v)$=0 based on the evicted fingerprint f_v and the current bucket index $h_1(v)$=3 by (3). Finally, we that bucket 0 is vacant, and therefore relocate the fingerprint f_v to bucket 0.

Algorithm 2: ASCF_Lookup (x)

```
1: h0 : fx = G(x);
2: h1 = (h0 + H(fx)) mod m/2 + m/2;
3: search fx in bucket h0 or h1;
4: if find fx in bucket h0 or h1 then {
5:     return true; }
6: return false;
```

B. Lookup Algorithm

Algorithm 2 shows the lookup algorithm of the ASCF. The lookup procedure of the ASCF is simple. When query an item x against the ASCF, Lines 1 to 2 compute x's fingerprint f_x and the two candidate bucket indexes h_0 and h_1 by (1) and (2). Line 3 searches the fingerprint f_x in the two candidate buckets h_0 and h_1 to check whether there is at least one matching fingerprint or not. If f_x is found in at least one of the two buckets h_0 or h_1, then x is in the set; otherwise, x is not in the set (see Lines 4 to 6).

Fig. 4 shows an example of looking up an item x against an ASCF by checking whether at least one matching fingerprint is found in its two candidate buckets or not. For x, we first compute the fingerprint f_x and the two candidate bucket indexes $h_0(x)$=1 and $h_1(x)$=4 by (1) and (2). Then, we search the two candidate

buckets 1 and 4 to check whether f_x is found in these two buckets or not. We find that the fingerprint f_x is in bucket 1, and therefore report that x is in the set.

Algorithm 3: ASCF_Delete (x)

```
1: h0 : fx = G(x);
2: h1 = (h0 + H(fx)) mod m/2 + m/2;
3: search fx in bucket h0 or h1;
4: if find fx in bucket h0 or h1 then {
5:     remove a copy of fx from this bucket;
6:     return true; }
7: return false;
```

C. Delete Algorithm

Algorithm 3 shows the delete algorithm of the ASCF. The deletion procedure of the ASCF is also simple. When deleting an item x from the ASCF, Lines 1 to 2 compute x's fingerprint f_x and the two candidate buckets h_0 and h_1 by (1) and (2). Then, Line 3 searches the fingerprint f_x in both two candidate buckets h_0 and h_1 to check whether at least one matching fingerprint is found in these two buckets or not. Lines 4 to 7 remove one copy of matching fingerprints when f_x is found in bucket h_0 or h_1.

We note that this deletion is absolutely safe even though two items have the same fingerprint. This is because there are two duplicate fingerprints stored in the ASCF when two items have the same fingerprint. Deleting one of the duplicate fingerprints does not yield a false negative, as another duplicate fingerprint is still in the filter. For instance, as shown in Fig. 5, we assume that two items y and a have the same fingerprint (i.e., f_y==f_a) and there are two duplicate fingerprints f_y and f_a stored in the ASCF. When y is deleted, we first compute y's fingerprint f_y and the two candidate bucket indexes $h_0(y)$=1 and $h_1(y)$=3. Then, we search f_y in buckets 1 and 3 and find that f_y exists in these two buckets because of f_y==f_a. Finally, we remove one duplicate fingerprint f_a from bucket 1, and retain another duplicate fingerprint f_y in bucket 3. When y is queried, we check to find that the fingerprint f_y is in bucket 3, which may yield a false positive but not a false negative. When a is queried, we check to find that the fingerprint f_a is in bucket 3 because of f_y==f_a, and therefore report that a is correctly in the set. In summary, such deletion has no impact on the false positive rate and performance of the ASCF.

D. Time and Space Complexities Analysis

We now analyze the time and space complexities of the ASCF. As shown in Table I, we assume that an ASCF consists of k=2 blocks each with m/k buckets for storing up to n items by using k=2 hash functions. Each bucket contains b=4 slots, each storing an f-bit fingerprint. In addition, f denotes the fingerprint size in bits, ε denotes a target false positive rate, and σ denotes a load factor in a full filter.

Table II compares the time complexities of the CBF, CF, and ASCF in terms of the number of memory accesses for a lookup. We see that the ASCF and CF have two memory accesses per lookup, and they are faster than the CBF with $\log_2(1/\varepsilon)$ memory accesses per lookup for a target false positive rate ε. For instance, the CBF requires 10 memory accesses per lookup for ε=10^{-3}, which results in lower lookup performance than the ASCF and CF. In addition, we show that the ASCF and CF have the same insert and delete performance. This is because they require two memory accesses on average for an insertion or deletion.

TABLE I. NOTATIONS USED IN THE PAPER.

Notation	Description
k	number of hash functions per item
b	number of slots per bucket
m	number of buckets in a filter
M	power of two that just contains m buckets
n	number of stored items in a filter
f	fingerprint size in bits
c	bits per item
ε	target false positive rate
σ	load factor in a full filter

TABLE II. NUMBER OF MEMORY ACCESSES PER LOOKUP AND BITS PER ITEM REQUIRED BY THE CBF, CF, AND ASCF.

Filter Type	Number of Memory Accesses Per Lookup	Bits Per Item
CBF	$\log_2(1/\varepsilon)$	$4\times1.44\log_2(1/\varepsilon)$
CF	2	$\log_2(8/\varepsilon)/\sigma \sim 2\times\log_2(8/\varepsilon)/\sigma$
ASCF	2	$\log_2(8/\varepsilon)/\sigma$

Next, we compare the space complexity of the ASCF with that of the CF. Since the false positive rate dominates the space complexity, we first analyze the false positive rates achieved by the ASCF and CF. For a query on an item, the ASCF needs to search $k\times b$ slots in its k candidate buckets to check whether there is at least one matching f-bit fingerprint in these buckets or not. Therefore, the false positive rate ε of the ASCF is computed by

$$\varepsilon=1-(1-\frac{1}{2^f})^{k\times b} \approx k\times b/2^f \tag{4}$$

Eq. (4) shows that the ASCF has the same false positive rate as the CF [23]. To achieve a target false positive rate ε, we compute the minimum fingerprint size f in bits required by the ASCF as:

$$f=\log_2(k\times b/\varepsilon)=\log_2(8/\varepsilon) \tag{5}$$

Eq. (5) shows that the fingerprint size f is only dominated by the target false positive rate ε in the ASCF. Therefore, we compute the space complexity of the ASCF in bits per item named c_{ASCF} as:

$$c_{\mathrm{ASCF}}=\frac{m\times b\times f}{n}=\frac{f}{\sigma}=\log_2(8/\varepsilon)/\sigma \tag{6}$$

where σ is a load factor (i.e., 0.95) of the ASCF. Eq. (6) shows that the ASCF has constant space cost per item.

In contrast, the practical CF requires the number of buckets to be a power of two (i.e., M in Table I), satisfying $m\le M\le 2m$. So we compute bits per item required by the CF named c_{CF} by

$$c_{\mathrm{CF}}=\frac{M\times b\times f}{n}\le\frac{2m\times b\times f}{n}=2\times c_{\mathrm{ASCF}} \tag{7}$$

TABLE III. FINGERPRINT SIZE IN BITS REQUIRED BY THE DLCBF, RCBF, CF, CQF, AND ASCF.

False Positive Rate	Fingerprint Size in Bits				
	dlCBF	RCBF	CF	CQF	ASCF
10^{-2}	12	7	10	7	10
10^{-3}	15	10	13	10	13
10^{-4}	18	14	17	14	17
10^{-5}	22	17	20	17	20
10^{-6}	25	20	23	20	23

TABLE IV. SPACE COST PER ITEM REQUIRED BY THE ASCF AND PREVIOUS FILTERS.

False Positive Rate	Bits Per Item						
	BF	CBF	dlCBF	RCBF	CF	CQF	ASCF
10^{-2}	9.6	38.3	18.7	13.3	20.0	12.2	10.5
10^{-3}	14.4	57.4	22.7	16.3	26.0	16.2	13.7
10^{-4}	19.1	76.5	26.7	20.3	34.0	21.5	17.9
10^{-5}	23.9	95.7	32.0	23.3	40.0	25.5	21.1
10^{-6}	28.7	114.8	36.0	26.3	46.0	29.5	24.2

Eq. (7) shows that the ASCF improves up to 2x space cost per item over the CF. Table II compares the space complexities of the CBF, CF and ASCF in terms of bits per item. We see that the ASCF requires less space than the CF and CBF. For instance, given $\varepsilon=10^{-3}$ and $\sigma=0.95$, the ASCF requires 13.6 bits per item while the CF requires 26.0 bits per item and the CBF requires 57.4 bits per item.

In summary, the ASCF has the same time complexity as the CF while requiring up to 2x less space than the CF. In addition, the ASCF outperforms other filters (e.g., the CBF) in both time and space complexities, which is validated in the next section.

IV. EVALUATION

We conducted simulation experiments to evaluate the ASCF and compare with previous six representative filters, including the standard BF [1], CBF [4], dlCBF [17], RCBF [19], CQF [22], and CF [23]. In this section, we first describe the experimental methodology, and then provide the results on synthetic datasets.

A. Experimental Methodology

For a fair evaluation, we select optimal values of parameters for each filter to achieve the best performance as follows.

For both the standard BF and CBF, we use $k=\log_2(1/\varepsilon)$ hash functions to achieve a target false positive rate ε. We allocate $n\times1.44\times\log_2(1/\varepsilon)$ bits to the standard BF for a set of up to n items. We allocate $4n\times1.44\times\log_2(1/\varepsilon)$ bits to the CBF because it uses four bits for a counter to support dynamic deletions.

For the dlCBF, we use four hash functions to map an item to four blocks, each with the same number of buckets. Each bucket contains eight slots each with a two-bit counter and a fingerprint of size $\log_2(24/\varepsilon)$ bits. We allocate $n\times(2+\log_2(24/\varepsilon))/\sigma$ buckets to the dlCBF and set the load factor to be $\sigma=0.75$.

For the RCBF, we use one hash function to map an item to one of blocks, each with 64 buckets. Each bucket contains one slot that stores a fingerprint of size $\log_2(\sigma/\varepsilon)$ bits. We allocate n/σ buckets to the RCBF and set the load factor to be $\sigma=0.9$.

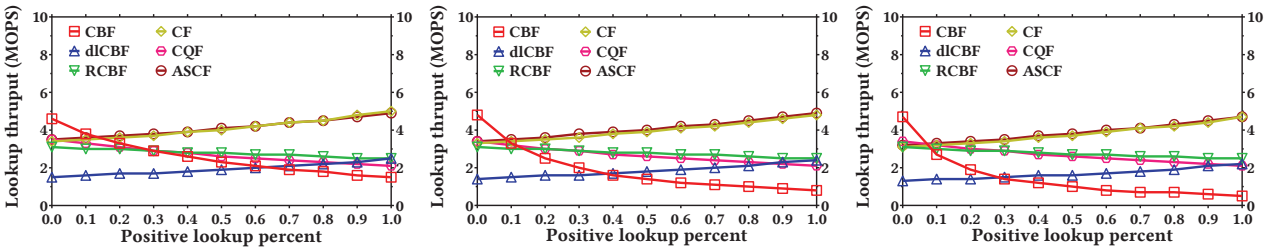
For the CQF, we use one hash function to map an item to one of blocks each with 64 buckets and a 136-bit metadata. Each bucket contains one slot that stores a fingerprint of size $\log_2(1/\varepsilon)$ bits. We allocate $n\times(2.125+\log_2(1/\varepsilon))/\sigma$ buckets to the CQF and set the load factor to be $\sigma=0.75$ for avoiding the overflow.

For both the CF and ASCF, we use two hash functions to map an item to two candidate buckets. Each bucket contains four slots, each with a fingerprint of size $\log_2(1/\varepsilon)$ bits. We allocate a power of two (i.e., M) buckets to the CF and $n\times\log_2(8/\varepsilon)/\sigma$ buckets to the ASCF, and set the same load factor to be $\sigma=0.95$.

We conducted two categories of experiments to evaluate the performance of each filter. In the first experiment, we run each filter that demands a maximum number of stored items. In the second experiment, we run each filter that demands a maximum memory size. We generate synthetic datasets of 64-bit integers for testing each filter. We use MurmurHash [29], a popular fast hash function to generate a random 64-bit hash value as an item. These experiments run on a server with Intel Xeon E5-2640 v3 CPU@2.6GHz (8 cores, 20MB L3 cache) and 96GB DDR3 main memory with one single thread. We measure the accuracy and performance metrics of each filter: the false positive rate, space cost, and lookup and update throughput. The results are averaged over ten trials.

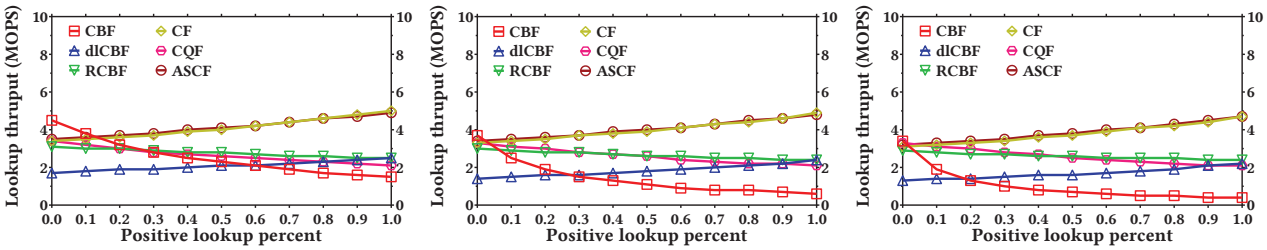
B. Results on Filters with Limited Numbers of Stored Items

In this experiment, we first insert synthetic store sets of 2^{20}, 2^{22}, and 2^{24} 64-bit integers into each filter with different target false positive rates (i.e., 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, and 10^{-6}). Then, we generate synthetic query sets each of which is 10 times the size of each store set to test each filter. Each query set has different positive lookup percentages ranging from 0% to 100%. The positive lookup percentage is X%, denoting that X% of queried



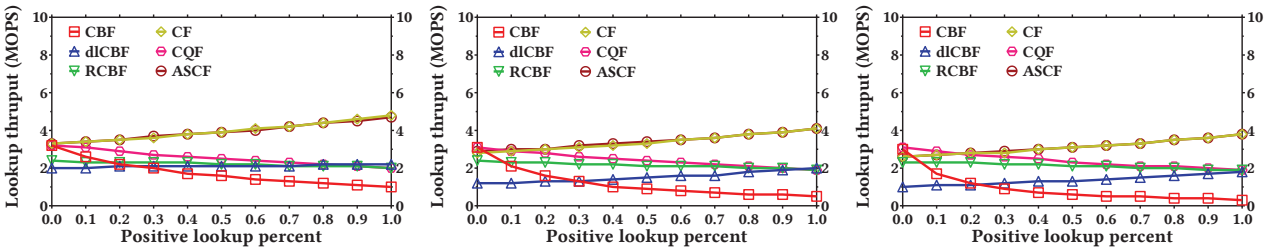
(a) False positive rate 10^{-2}. (b) False positive rate 10^{-4}. (c) False positive rate 10^{-6}.

Fig. 6. Lookup throughput for different positive lookup percentages when the number of items stored in each filter is 2^{20}.



(a) False positive rate 10^{-2}. (b) False positive rate 10^{-4}. (c) False positive rate 10^{-6}.

Fig. 7. Lookup throughput for different positive lookup percentages when the number of items stored in each filter is 2^{22}.



(a) False positive rate 10^{-2}. (b) False positive rate 10^{-4}. (c) False positive rate 10^{-6}.

Fig. 8. Lookup throughput for different positive lookup percentages when the number of items stored in each filter is 2^{24}.

items is in the set while 1-X% is not in the set. Finally, we delete all items from each filter to test the delete performance.

We note that the fingerprint-based variants of CBFs (e.g., the dlCBF, RCBF, CF, CQF, and ASCF) require non-byte-aligned fingerprints to achieve target false positive rates. Table III shows the size of non-byte-aligned fingerprints required by the dlCBF, RCBF, CF, CQF, and ASCF for different false positive rates. We see that the ASCF and CF have the same fingerprint size for same target false positive rates. To achieve high performance, we use an SIMD instruction *_bextr_u32* to extract a non-byte-aligned fingerprint from a 32-bit integer in these filters.

Space Cost: Table IV shows the space cost per item required by the ASCF and previous filters. We see that the ASCF requires lower space cost per item than previous filters for low false positive rates $\varepsilon \leq 10^{-3}$, the usual case for most of applications. Specifically, the ASCF reduces up to 17.9% space cost per item over the CQF, up to 47.4% (1.9x) over the CF, up to 20.9% over the RCBF, up to 43.6% over the dlCBF, up to 78.9% over the CBF, and even up to 15.6% over the non-deletable standard BF. This is because the ADD/SUB operations do not require the total number of buckets in the ASCF to be a power of two, which

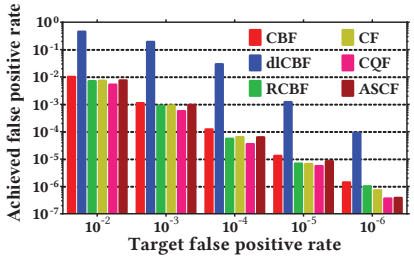
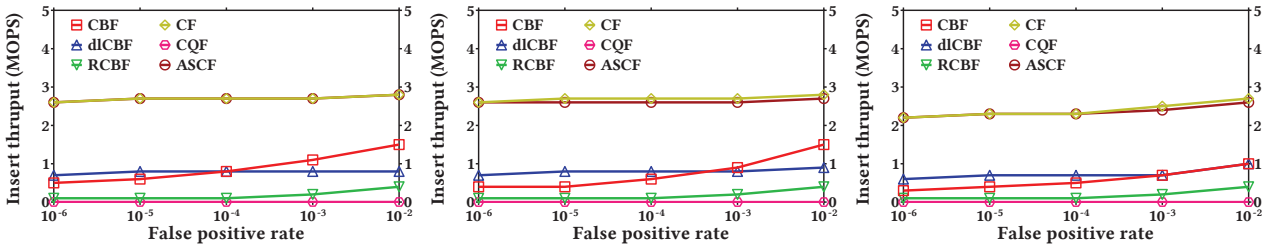


Fig. 9. False positive rates achieved for different target false positive rates when the maximum number of items is stored in each filter.

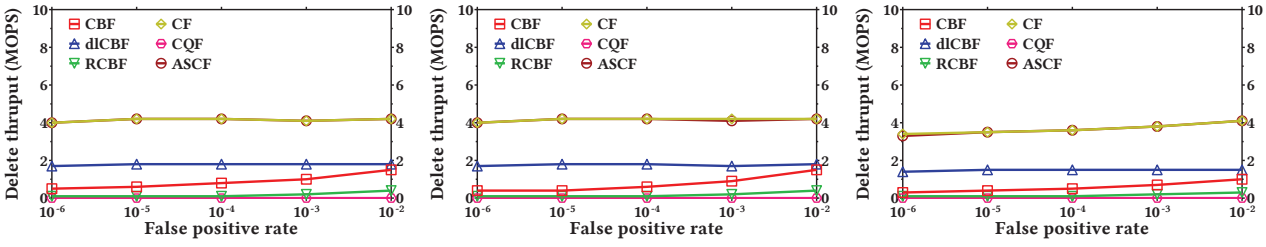
results in high space efficiency. We note that the non-deletable standard BF requires lower space cost per item than the ASCF when the false positive rate is 10^{-2}.

Lookup Throughput: Fig. 6, Fig. 7, and Fig. 8 show the lookup throughput in million operations per second (MOPS) for different positive lookup percentages when the number of items stored in each filter is 2^{20}, 2^{22}, and 2^{24}, respectively. The lookup throughput of each filer varies as the positive lookup percentage varies from 0% to 100%. From these three figures, we see that



(a) 2^{20} items. (b) 2^{22} items. (c) 2^{24} items.

Fig. 10. Insert throughput for different target false positive rates when the maximum number of items stored in each filter is 2^{20}, 2^{22}, and 2^{24}.



(a) 2^{20} items. (b) 2^{22} items. (c) 2^{24} items.

Fig. 11. Delete throughput for different target false positive rates when the maximum number of items stored in each filter is 2^{20}, 2^{22}, and 2^{24}.

TABLE V. MAXIMUM NUMBERS OF ITEMS STORED IN DIFFERENT FIXED-SIZE FILTERS.

Fixed Filter Size	False Positive Rate	Maximum Number of Stored Items					
		CBF	dlCBF	RCBF	CF	CQF	ASCF
8MB	10^{-2}	1,753,626	2,795,724	3,355,443	3,984,588	2,776,917	3,984,588
	10^{-4}	876,813	2,795,713	3,355,443	3,984,588	2,776,917	3,984,588
	10^{-6}	584,542	2,795,722	3,355,443	3,984,588	2,776,917	3,984,588

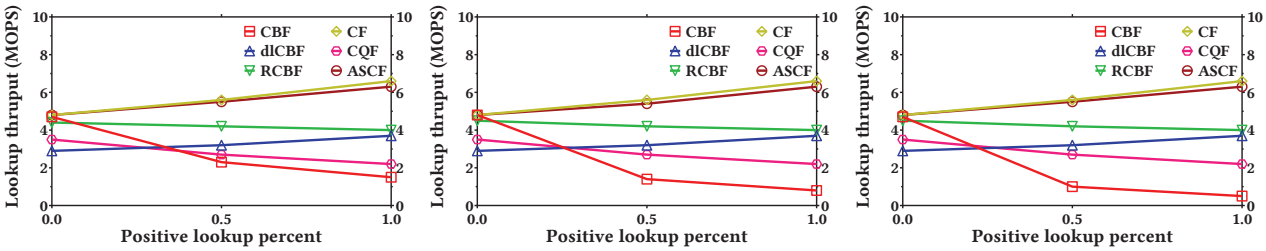
TABLE VI. MINIMUM BITS PER ITEM REQUIRED BY DIFFERENT FIXED-SIZE FILTERS.

Fixed Filter Size	False Positive Rate	Minimum Bits Per Item					
		CBF	dlCBF	RCBF	CF	CQF	ASCF
8MB	10^{-2}~10^{-6}	38.3~114.8	24.0	22.2	16.8	24.2	16.8

the ASCF achieves the same lookup throughput as the CF. This is because the ADD/SUB operations used by the ASCF almost perform as fast as the XOR operation used by the CF. In addition, the ASCF outperforms other filters except for the CBF when the positive lookup percentage is no larger than 10%. We see that when the positive lookup percentage increases from 0% to 10%, the lookup throughput of the ASCF increases while that of the CBF decreases. This is because the ASCF needs to search fewer slots for a positive lookup than for a negative lookup while in contrast the CBF needs to search more slots (i.e., counters) for a positive lookup. Moreover, we see that the target false positive rate has little impact on the lookup throughput of the ASCF. This is because the false positive rate dominates the fingerprint size but not the lookup performance of the ASCF.

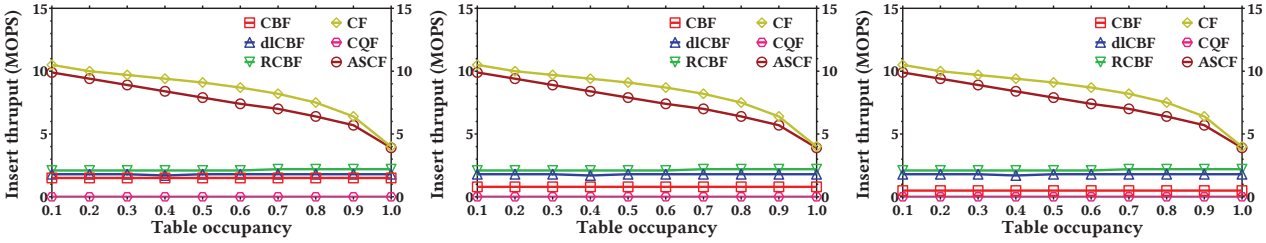
False Positive Rates: Fig. 9 shows the false positive rates achieved for target false positive rates ranging from 10^{-2} to 10^{-6} when the maximum number of items is stored in each filter. We see that the ASCF almost achieves the same false positive rates as the CQF, CF, and RCBF, and reduces 12.1% to 72.6% false positive compared to the CBF. We note that the dlCBF achieves pretty larger false positive rates than other filters. This is because it uses short fingerprints for high space efficiency, which incurs the increase in false positive rates.

Update Throughput: Fig. 10 and Fig. 11 show the insert and delete throughput in MOPS for different false positive rates when the maximum number of items is stored in each filter. We see that the ASCF achieves the same update throughput as the CF. Specifically, on average, the ASCF and CF have 2.6 MOPS insert throughput and 4.0 MOPS delete throughput. In addition, the ASCF achievers higher update throughput than other filters. Specifically, the ASCF has higher insert (or delete) throughput than the CBF by up to 7.0x (or 11.1x), dlCBF by up to 3.7x (or 2.7x), RCBF by up to 40.7x (or 67.2x), and CQF by up to 10849x (or 24214x), respectively. Moreover, the target false positive rate has little impact on the insert and delete throughput of the ASCF. This is because it requires two memory accesses to two buckets for an insertion or deletion, which is independent on a target false positive rate.



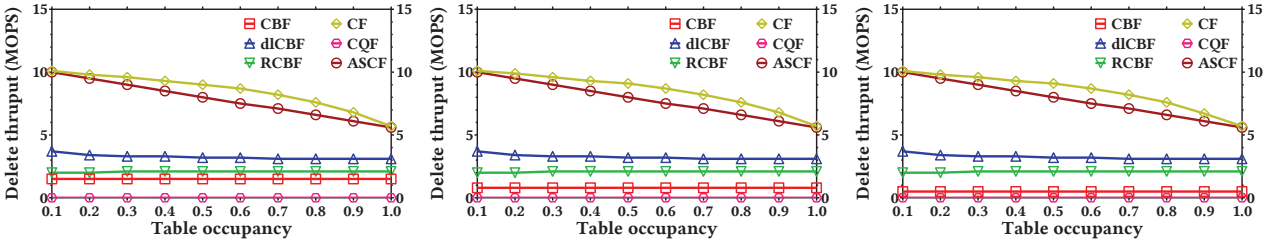
(a) False positive rate 10^{-2}. (b) False positive rate 10^{-4}. (c) False positive rate 10^{-6}.

Fig. 12. Lookup throughput for different positive lookup percentages when the table occupancy is 100% and the memory size of each filter is 8MB.



(a) False positive rate 10^{-2}. (b) False positive rate 10^{-4}. (c) False positive rate 10^{-6}.

Fig. 13. Insert throughput for different table occupancies when the memory size of each filter is 8MB.



(a) False positive rate 10^{-2}. (b) False positive rate 10^{-4}. (c) False positive rate 10^{-6}.

Fig. 14. Delete throughput for different table occupancies when the memory size of each filter is 8MB.

C. *Results on Filters with Limited Memory Size*

In this experiment, we select proper values of parameters to ensure that the CF works with the optimal performance. This is because the CF requires the number of buckets to be a power of two. Therefore, we configure each filter to be the same fixed size (i.e., 8MB) and the fingerprint size to be 16 bits for fitting in an integer in the CF, CQF, and ASCF. In this experiment, we first insert a synthetic store set of n 64-bit integers into each filter with different table occupancies ranging from 10% to 100% for different target false positive rates 10^{-2}, 10^{-4}, and 10^{-6}. The table occupancy is Y%, denoting that there is Y% of items stored in a full filter. Then, we generate a synthetic query set of $10\times n$ 64-bit integers with different positive lookup percentages (i.e., 0%, 50%, and 100%) to test each filter. Finally, we delete all items from each filter to test the delete performance.

Space Cost: Table V shows the maximum number of items stored in each filter with the fixed memory size of 8MB. In this case, the larger number of items is stored in a filter, the lower space cost per item is achieved by a filter. From the table, we see that the ASCF has the same maximum number of stored items as the CF, but has more items than other filters. In addition, Table VI shows the minimum bits per item required by different filters. We see that the ASCF achieves the same minimum space cost per item as the CF, but requires lower space cost per item than other filters. This is because the ASCF and CF have the same number of buckets (i.e., a power of two) and the same load factor (i.e., 0.95). When the number of buckets is not a power of two, the ASCF requires lower space cost per item than the CF (see Table IV).

Lookup Throughput: Fig. 12 shows the lookup throughput in MOPS for different positive lookup percentages when the table occupancy is 100%. We see that the ASCF almost achieves the same lookup throughput as the CF because the ADD/SUB operations have the same performance as the XOR operation. The ASCF achieves higher lookup throughput than other filters. In addition, the ASCF increases the lookup throughput as the positive lookup percentage increases. This is because the ASCF searches fewer slots for a positive lookup than for a negative lookup when the positive lookup percentage becomes larger.

Update Throughput: Fig. 13 and Fig. 14 show the insert and delete throughput in MOPS for different table occupancies. We see that the ASCF almost achieves the same insert and delete throughput as the CF, and outperforms other filters. Specifically, the ASCF achieves up to 12% lower insert and delete throughput than the CF when the table occupancy varies from 10% to 100%. Therefore, the ASCF has high update performance comparable to the CF. In addition, the ASCF achieves higher insert (or delete) throughput than the CBF by up to 18.7x (or up to 18.8x), dlCBF by up to 5.5x (or up to 2.8x), RCBF by up to 4.7x (or up to 4.9x), and CQF by up to 5904x (or up to 5689x), respectively.

V. Conclusion

This paper presents ASCF, a scalable variant of the cuckoo filter for improving the space efficiency while sustaining high performance. The ASCF is a blocked compact cuckoo hash table that uses the addition and subtraction (ADD/SUB) operations, instead of the XOR operation, to compute two candidate bucket indexes for an item's fingerprint. The ADD/SUB operations not only perform fast, but also do not require the total number of buckets in the filter to be a power of two. Therefore, beyond the CF, the ASCF achieves lower space cost as well as the same high performance. Experimental results on simulation show that the ASCF requires less space than previous filters, and reduces up to 1.9x space cost per item over the CF. In addition, the ASCF achieves the same lookup and update throughput as the CF, but outperforms other filters.

References

[1] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. Communications of the ACM, 13(7): 442-426, 1970.

[2] A. Broder and M. Mitzenmacher. Network applications of bloom filters: a survey. Internet Mathematics, 1(4): 488-509, 2004.

[3] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. IEEE Communications Survey & Tutorials, 14(1): 131-155, 2012.

[4] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: a scalable wide-area web cache sharing protocol. In ACM SIGCOMM, 1998.

[5] D. Eppstein, M. Goodrich, F. Uyeda, and G. Varghese. What's the difference?: efficient set reconciliation without prior context. In ACM SIGCOMM, 2011.

[6] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using bloom filters. In ACM SIGCOMM, 2003.

[7] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In ACM SIGCOMM, 2005.

[8] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond bloom filters: from approximate membership checks to approximate state machines. In ACM SIGCOMM, 2006.

[9] M. Yu, A. Fabrikant, and J. Rexford. BUFFALO: bloom filter forwarding architecture for large organizations. In ACM CoNEXT, 2009.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In USENIX OSDI, 2006.

[11] H. Lim, B. Fan, D. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In ACM SOSP, 2011.

[12] B. Fan, D. Andersen, and M. Kaminsky. MemC3: compact and concurrent memcache with dumber caching and smarter hashing. In USENIX NSDI, 2013.

[13] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y. Kim, M. Carey, M. Dreseler, and C. Li. Storage management in AsterixDB. In PVLDB, 2014.

[14] B. Zhu, K. Li, and R. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In USENIX FAST, 2008.

[15] B. Debnath, S. Sengupta, and J. Li. Chunkstash: speeding up inline storage deduplication using flash memory. In USENIX ATC, 2010.

[16] B. Debnath, S. Sengupta, J. Li, D. Lilja, and D. Du. BloomFlash: bloom filter on flash-based storage. In IEEE ICDCS, 2011.

[17] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In ESA, 2006.

[18] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Multilayer compressed counting bloom filters. In IEEE INFOCOM, 2008.

[19] N. Hua, H. Zhao, B. Lin, and J. Xu. Rank-indexed hashing: a compact construction of bloom filters and variants. In IEEE ICNP, 2008.

[20] O. Rottenstreich, Y. Kanizo, and I. Keslassy. The variable-increment counting bloom filter. In IEEE INFOCOM, 2012.

[21] M. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R.Spillane,and E. Zadok. Don't thrash: how to cache your hash on flash. In PVLDB, 2012.

[22] P. Pandey, M. Bender, R. Johnson, and R. Patro. A general-purpose counting filter: making every bit count. In ACM SIGMOD, 2017.

[23] B. Fan, D. Andersen, M. Kaminskyy, and M. Mitzenmacherz. Cuckoo filter: practically better than bloom. In ACM CoNEXT, 2014.

[24] R. Pagh and F. Rodler. Cuckoo hashing. Journal of Algorithms, 51(2): 122–144, 2004.

[25] F. Putze, P. Sanders, and S. Johannes. Cache-, hash- and space-efficient bloom filters. ACM Journal of Experimental Algorithmics, 14(4): 1-18, 2009.

[26] Y. Qiao, T. Li, and S. Chen. One memory access bloom filters and their generalization. In IEEE INFOCOM, 2011.

[27] T. Yang, A. Liu, M. Shahzad, Y. Zhong, Q. Fu, Z. Li, G. Xie, and X. Li. A shifting bloom filter framework for set queries. In PVLDB, 2016.

[28] A. Breslow and N. Jayasena. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. In PVLDB, 2018.

[29] MurmurHash. https://sites.google.com/site/murmurhash.