Cuckoo Filter: Practically Better Than Bloom

Bin Fan, David G. Andersen, Michael Kaminsky[†], Michael D. Mitzenmacher[‡]
Carnegie Mellon University, [†]Intel Labs, [‡]Harvard University
{binfan,dga}@cs.cmu.edu, michael.e.kaminsky@intel.com, michaelm@eecs.harvard.edu

ABSTRACT

In many networking systems, Bloom filters are used for high-speed set membership tests. They permit a small fraction of false positive answers with very good space efficiency. However, they do not permit deletion of items from the set, and previous attempts to extend "standard" Bloom filters to support deletion all degrade either space or performance.

We propose a new data structure called the *cuckoo filter* that can replace Bloom filters for approximate set membership tests. Cuckoo filters support adding and removing items dynamically while achieving even higher performance than Bloom filters. For applications that store many items and target moderately low false positive rates, cuckoo filters have lower space overhead than space-optimized Bloom filters. Our experimental results also show that cuckoo filters outperform previous data structures that extend Bloom filters to support deletions substantially in both time and space.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures; E.4 [**Data**]: Data Compaction and Compression

Keywords

Cuckoo hashing; Bloom filters; compression

1. INTRODUCTION

Many databases, caches, routers, and storage systems use *approximate set membership* tests to decide if a given item is in a (usually large) set, with some small false positive probability. The most widely-used data structure for this test is the Bloom filter [3], which has been studied extensively due to its memory efficiency. Bloom filters have been used to: reduce the space required in probabilistic routing tables [25]; speed longest-prefix matching for IP addresses [9]; improve network state management and monitoring [24, 4]; and encode multicast forwarding information in packets [15], among many other applications [6].

A limitation of standard Bloom filters is that one cannot remove existing items without rebuilding the entire filter (or possibly introducing generally less desirable false negatives). Several approaches extend standard Bloom filters to support deletion, but with significant space or performance overhead. *Counting Bloom filters* [12] have been suggested for multiple applications [24, 25, 9], but they generally use 3–4× space to retain the same false positive rate as a space-optimized Bloom filter. Other variants include *d-left counting Bloom filters* [5], which are still 1.5× larger, and *quotient filters* [2], which provide significantly degraded lookup performance to yield comparable space overhead to Bloom filters.

This paper shows that supporting deletion in approximate set membership tests need not impose higher overhead in space or performance compared to standard Bloom filters. We propose the *cuckoo filter*, a practical data structure that provides four major advantages.

1. It supports adding and removing items dynamically;
2. It provides higher lookup performance than traditional Bloom filters, even when close to full (e.g., 95% space utilized);
3. It is easier to implement than alternatives such as the quotient filter; and
4. It uses less space than Bloom filters in many practical applications, if the target false positive rate ϵ is less than 3%.

A cuckoo filter is a compact variant of a cuckoo hash table [21] that stores only *fingerprints*—a bit string derived from the item using a hash function—for each item inserted, instead of key-value pairs. The filter is densely filled with fingerprints (e.g., 95% entries occupied), which confers high space efficiency. A set membership query for item x simply searches the hash table for the fingerprint of x, and returns true if an identical fingerprint is found.

When constructing a cuckoo filter, its fingerprint size is determined by the target false positive rate ϵ. Smaller values of ϵ require longer fingerprints to reject more false queries. Interestingly, while we show that cuckoo filters are practically better than Bloom filters for many real workloads, they are asymptotically worse: the minimum fingerprint size used in the cuckoo filter grows logarithmically with the number of entries in the table (as we explain in Section 4). As a consequence, the per-item space overhead is higher for larger tables, but this use of extra space confers a lower false positive

CoNEXT'14, Dec 02-05 2014, Sydney, Australia
ACM 978-1-4503-3279-8/14/12.
http://dx.doi.org/10.1145/2674005.2674994

filter type	**space cost**	**cache misses per lookup**	**deletion support**
Bloom	1	k	no
blocked Bloom	1x	1	no
counting Bloom	3x ∼ 4x	k	yes
d-left counting Bloom	1.5x ∼ 2x	d	yes
quotient	1x ∼ 1.2x	≥ 1	yes
cuckoo	≤1x	2	yes

Table 1: Properties of Bloom filters and variants. Assume standard and counting Bloom filters use k hash functions, and d-left counting Bloom filters have d partitions.

rate. For practical problems with a few billion items or fewer, a cuckoo filter uses less space *while supporting deletion* than a non-deletable, space-optimized Bloom filter when $\epsilon < 3\%$.

Cuckoo filters are substantially different from regular hash tables because only fingerprints are stored in the filter and the original key and value bits of each item are no longer retrievable. Because full keys are not stored, a cuckoo filter cannot even perform standard cuckoo hashing to insert new items, which involves moving existing keys based on their hash values. This difference means that the standard techniques, analyses, and optimizations that apply to cuckoo hashing do not necessarily carry over to cuckoo filters.

Technical contributions made by this paper include

- Applying partial-key cuckoo hashing—a variant of standard cuckoo hashing—to build cuckoo filters that support dynamic addition and deletion of items (Section 3).
- Exploring the reason why partial-key cuckoo hashing ensures high table occupancy for most real-world applications (Section 4).
- Optimizing cuckoo filters to outperform Bloom filters in space efficiency (Section 5).

2. BACKGROUND AND RELATED WORK

2.1 Bloom Filters and Variants

We compare standard Bloom filters and the variants that include support for deletion or better lookup performance, as summarized in Table 1. These data structures are evaluated empirically in Section 7. Cuckoo filters achieve higher space efficiency and performance than these data structures.

Standard Bloom filters [3] provide a compact representation of a set of items that supports two operations: `Insert` and `Lookup`. A Bloom filter allows a tunable false positive rate ϵ so that a query returns either "definitely not" (with no error), or "probably yes" (with probability ϵ of being wrong). The lower ϵ is, the more space the filter requires.

A Bloom filter consists of k hash functions and a bit array with all bits initially set to "0". To insert an item, it hashes this item to k positions in the bit array by k hash functions, and then sets all k bits to "1". Lookup is processed similarly, except it reads k corresponding bits in the array: if all the bits are set, the query returns true; otherwise it returns false. Bloom filters do not support deletion.

Bloom filters can be very space-efficient, but are not optimal [20]. For a false positive rate ϵ, a space-optimized Bloom filter uses $k = \log_2(1/\epsilon)$ hash functions. Such a Bloom filter can store each item using $1.44 \log_2(1/\epsilon)$ bits, which depends only on ϵ rather than the item size or the total number of items. The information-theoretic minimum requires $\log_2(1/\epsilon)$ bits per item, so a space-optimized Bloom filter imposes a 44% space overhead over the information-theoretic lower bound.

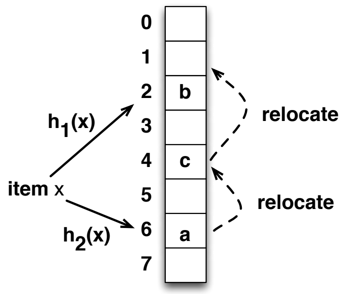
The information theoretic optimum is essentially achievable for a *static* set by using fingerprints (of length $\lceil 1/\epsilon \rceil$ bits) and a perfect hash table [6]. To efficiently handle deletions, we replace a perfect hash function with a well-designed cuckoo hash table.

Counting Bloom filters [12] extend Bloom filters to allow deletions. A counting Bloom filter uses an array of counters in place of an array of bits. An insert increments the value of k counters instead of simply setting k bits, and a lookup checks if each of the required counters is non-zero. The delete operation decrements the values of these k counters. To prevent *arithmetic overflow* (i.e., incrementing a counter that has the maximum possible value), each counter in the array must be sufficiently large in order to retain the Bloom filter's properties. In practice, the counter consists of four or more bits, and a counting Bloom filter therefore requires 4× more space than a standard Bloom filter. (One can construct counting Bloom filters to use less space by introducing a secondary hash table structure to manage overflowing counters, at the expense of additional complexity.)

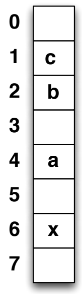
Blocked Bloom filters [22] do not support deletion, but provide better spatial locality on lookups. A blocked Bloom filter consists of an array of small Bloom filters, each fitting in one CPU cache line. Each item is stored in only one of these small Bloom filters determined by hash partitioning. As a result, every query causes at most one cache miss to load that Bloom filter, which significantly improves performance. A drawback is that the false positive rate becomes higher because of the imbalanced load across the array of small Bloom filters.

***d*-left Counting Bloom filters** [5] are similar to the approach we use here. Hash tables using *d-left hashing* [19] store fingerprints for stored items. These filters delete items by removing their fingerprint. Compared to counting Bloom filters, they reduce the space cost by 50%, usually requiring $1.5 - 2\times$ the space compared to a space-optimized non-deletable Bloom filter. Cuckoo filters achieve better space efficiency than d-left counting Bloom filters as we show, and have other advantages, including simplicity.

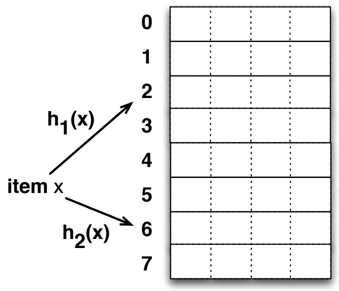
Quotient filters [2] are also compact hash tables that store fingerprints to support deletion. Quotient filters uses a technique similar to linear probing to locate a fingerprint, and thus provide better spatial locality. However, they require additional meta-data to encode each entry, which requires



(a) before inserting item *x*



(b) after item *x* inserted



(c) A cuckoo filter, two hash per item and functions and four entries per bucket

Figure 1: Illustration of cuckoo hashing

10 ~ 25% more space than a comparable standard Bloom filter. Moreover, all of its operations must decode a sequence of table entries before reaching the target item, and the more the hash table is filled, the longer these sequences become. As a result, its performance drops significantly when the occupancy of the hash table exceeds 75%.

Other Variants: Other variants have been proposed to improve Bloom filters, either in space and/or performance. Rank-Indexed Hashing [14] builds linear chaining hash tables to store compressed fingerprints. Although similar to and somewhat more space efficient than *d*-left counting Bloom filters, updating the internal index that reduces the chaining cost is very expensive, making it less appealing in dynamic settings. Putze et al. proposed two variants of Bloom filters [22]. One is the previously discussed *Blocked Bloom filter*; the other, called a *Golomb-Compressed Sequence* stores all items' fingerprints in a sorted list. Its space is near-optimal, but the data structure is static and requires non-constant lookup time to decode the encoded sequence. It is therefore not evaluated with other filters in this paper. Pagh et al. proposed an asymptotically space-optimal data structure [20] based on Cleary [8]. This data structure, however, is substantially more complex than its alternatives and does not appear amenable to a high performance implementation. In contrast, cuckoo filters are easy to implement.

2.2 Cuckoo Hash Tables

Cuckoo Hashing Basics: A basic cuckoo hash table [21] consists of an array of buckets where each item has two candidate buckets determined by hash functions $h_1(x)$ and $h_2(x)$. The lookup procedure checks both buckets to see if either contains this item. Figure 1(a) shows the example of inserting a new item x in to a hash table of 8 buckets, where x can be placed in either buckets 2 or 6. If either of x's two buckets is empty, the algorithm inserts x to that free bucket and the insertion completes. If neither bucket has space, as is the case in this example, the item selects one of the candidate buckets (e.g., bucket 6), kicks out the existing item (in this case "a") and re-inserts this victim item to its own alternate location. In our example, displacing "a" triggers another relocation that kicks existing item "c" from bucket 4 to bucket 1. This procedure may repeat until a vacant bucket is found as illustrated in Figure 1(b), or until a maximum number of displacements is reached (e.g., 500 times in our implementation). If no vacant bucket is found, this hash table is considered too full to insert. Although cuckoo hashing may execute a sequence of displacements, its amortized insertion time is $O(1)$.

Cuckoo hashing ensures high space occupancy because it refines earlier item-placement decisions when inserting new items. Most practical implementations of cuckoo hashing extend the basic description above by using buckets that hold multiple items, as suggested in [10]. The maximum possible load when using k hash functions and buckets of size b assuming all hash functions are perfectly random has been analyzed [13]. With proper configuration of cuckoo hash table parameters (explored in Section 5), the table space can be 95% filled with high probability.

Using Cuckoo Hashing for Set-membership: Recently, standard cuckoo hash tables have been used to provide set membership information in a few applications. To support transactional memory, Sanchez et al. proposed to store the read/write set of memory addresses of each transaction in a cuckoo hash table, and to convert this table to Bloom filters when full [23]. Their design used standard cuckoo hash tables, and thus required much more space than cuckoo filters. Our previous study in building high-speed and memory-efficient key-value stores [17, 11] and software-based Ethernet switches [26] all applied cuckoo hash tables as internal data structures. That work was motivated by and also focused on improving hash table performance by an optimization called *partial-key cuckoo hashing*. However, as we show in this paper, this technique also enabled a new approach to build a Bloom filter replacement which has not been studied before. As a result, this paper also applies partial-key cuckoo hashing, but more importantly it offers an in-depth analysis of using this technique specifically to serve set membership tests (rather than key-value queries) and further compares the performance of cuckoo filters with alternative set membership data structures.

Challenges in Cuckoo Filter: To make cuckoo filters highly space efficient, we use a multi-way associative cuckoo hash table to provide high-speed lookup and high table occupancy (e.g., 95% hash table slots filled); to further reduce the hash table size, each item is first hashed into a constant-sized fingerprint before inserted into this hash table. The challenge of applying this data structure is to redesign the insert process and carefully configure the hash table to minimize space usage per item:

- First, storing only fingerprints in the hash table prevents inserting items using the standard cuckoo hashing approach. Because in cuckoo hashing the insertion algorithm must be able to relocate existing fingerprints to their alternative locations. A space-inefficient but straightforward solution is to store each inserted item in its entirety (perhaps externally to the table); given the original item ("key"), calculating its alternate location is easy. In contrast, cuckoo filters use *partial-key cuckoo hashing* to find an item's alternate location based on only its fingerprint (Section 3).
- Second, cuckoo filter associates each item with multiple possible locations in the hash table. This flexibility in where to store an item improves table occupancy, but retaining the same false positive rate when probing more possible locations on each lookup requires more space for longer fingerprints. In Section 5, we present our analysis to optimize the balance between the table occupancy and its size to minimize the average space cost per item.

3. CUCKOO FILTER ALGORITHMS

In this paper, the basic unit of the cuckoo hash tables used for our cuckoo filters is called an *entry*. Each entry stores one fingerprint. The hash table consists of an array of *buckets*, where a bucket can have multiple entries.

This section describes how cuckoo filters perform `Insert`, `Lookup` and `Delete` operations. Section 3.1 presents *partial-key cuckoo hashing*, a variant of standard cuckoo hashing that enables cuckoo filters to insert new items *dynamically*. This technique was first introduced in previous work [11], but there the context was improving the lookup and insert performance of regular cuckoo hash tables where full keys were stored. In contrast, this paper focuses on optimizing and analyzing the space efficiency when using partial-key cuckoo hashing with only fingerprints, to make cuckoo filters competitive with or even more compact than Bloom filters.

3.1 Insert

As previously stated, with standard cuckoo hashing, inserting new items to an existing hash table requires some means of accessing the original existing items in order to determine where to relocate them if needed to make room for the new ones (Section 2.2). Cuckoo filters, however, only store fingerprints and therefore there is no way to restore and rehash the original keys to find their alternate locations. To overcome this limitation, we utilize a technique called *partial-key cuckoo hashing* to derive an item's alternate location based on its fingerprint. For an item x, our hashing scheme calculates the indexes of the two candidate buckets as follows:

$$\begin{aligned} h_1(x) &= \text{hash}(x), \\ h_2(x) &= h_1(x) \oplus \text{hash}(x\text{'s fingerprint}). \end{aligned} \tag{1}$$

The xor operation in Eq. (1) ensures an important property: $h_1(x)$ can also be calculated from $h_2(x)$ and the fingerprint using the same formula. In other words, to displace a key originally in bucket i (no matter if i is $h_1(x)$ or $h_2(x)$), we directly calculate its alternate bucket j from the current bucket index i and the fingerprint stored in this bucket by

$$j = i \oplus \text{hash}(\text{fingerprint}). \tag{2}$$

Hence, an insertion only uses information in the table, and never has to retrieve the original item x.

In addition, the fingerprint is hashed before it is xor-ed with the index of its current bucket to help distribute the items uniformly in the table. If the alternate location were calculated by "$i \oplus$fingerprint" without hashing the fingerprint, the items kicked out from nearby buckets would land close to each other in the table, if the size of the fingerprint is small compared to the table size. For example, using 8-bit fingerprints the items kicked out from bucket i will be placed to buckets that are at most 256 buckets away from bucket i, because the xor operation would alter the eight low order bits of the bucket index while the higher order bits would not change. Hashing the fingerprints ensures that these items can be relocated to buckets in an entirely different part of the hash table, hence reducing hash collisions and improving the table utilization.

Using partial-key cuckoo hashing, cuckoo filters add new items dynamically by the process shown in Algorithm 1. Because these fingerprints can be significantly shorter than the

Algorithm 1: `Insert(x)`

```
$f$ = fingerprint($x$);
$i_1$ = hash($x$);
$i_2$ = $i_1 \oplus$ hash($f$);
if bucket[$i_1$] or bucket[$i_2$] has an empty entry then
    add $f$ to that bucket;
    return Done;
// must relocate existing items;
$i$ = randomly pick $i_1$ or $i_2$;
for $n = 0$; $n <$ MaxNumKicks; $n$++ do
    randomly select an entry $e$ from bucket[$i$];
    swap $f$ and the fingerprint stored in entry $e$;
    $i = i \oplus$ hash($f$);
    if bucket[$i$] has an empty entry then
        add $f$ to bucket[$i$];
        return Done;
// Hashtable is considered full;
return Failure;
```

size of h_1 or h_2, there are two consequences. First, the total number of different possible choices of (h_1, h_2) as calculated by Eq. (1) can be much smaller than using a perfect hash to derive h_1 and h_2 as in standard cuckoo hashing. This may cause more collisions, and in particular previous analyses for cuckoo hashing (as in [10, 13]) do not hold. A full analysis of partial-key cuckoo hashing remains open (and beyond this paper); in Section 4, we provide a detailed discussion of this issue and consider how to achieve high occupancy for practical workloads.

Second, inserting two different items x and y that have the same fingerprint is fine; it is possible to have the same fingerprint appear multiple times in a bucket. However, cuckoo filters are not suitable for applications that insert the same item more than $2b$ times (b is the bucket size), because the two buckets for this duplicated item will become overloaded. There are several solutions for such a scenario. First, if the table need not support deletion, then this issue does not arise, because only one copy of each fingerprint must be stored. Second, one could, at some space cost, associate counters with buckets, and increment/decrement them appropriately. Finally, if the original keys are stored somewhere (perhaps in slower external storage), one could consult that record to prevent duplicate insertion entirely, at the cost of slowing down insertion if the table already contains a (false positive) matching entry for the bucket and fingerprint. Similar requirements apply to traditional and d-left counting Bloom filters.

Algorithm 2: `Lookup(x)`

```
$f$ = fingerprint($x$);
$i_1$ = hash($x$);
$i_2$ = $i_1 \oplus$ hash($f$);
**if** bucket[$i_1$] or bucket[$i_2$] has $f$ **then**
    **return** True;
**return** False;
```

3.2 Lookup

The lookup process of a cuckoo filter is simple, as shown in Algorithm 2. Given an item x, the algorithm first calculates x's fingerprint and two candidate buckets according to Eq. (1). Then these two buckets are read: if any existing fingerprint in either bucket matches, the cuckoo filter returns true, otherwise the filter returns false. Notice that this ensures no false negatives as long as bucket overflow never occurs.

Algorithm 3: `Delete(x)`

```
$f$ = fingerprint($x$);
$i_1$ = hash($x$);
$i_2$ = $i_1 \oplus$ hash($f$);
**if** bucket[$i_1$] or bucket[$i_2$] has $f$ **then**
    remove a copy of $f$ from this bucket;
    **return** True;
**return** False;
```

3.3 Delete

Standard Bloom filters cannot delete, thus removing a single item requires rebuilding the entire filter, while counting Bloom filters require significantly more space. Cuckoo filters are like counting Bloom filters that can delete inserted items by removing corresponding fingerprints from the hash tables on deletion. Other filters with similar deletion processes prove more complex than cuckoo filters. For example, d-left counting Bloom filters must use extra counters to prevent the "false deletion" problem on fingerprint collision[1], and quotient filters must shift a sequence of fingerprints to fill the "empty" entry after deletion and maintain their "bucket structure".[2]

The deletion process of cuckoo filters illustrated in Algorithm 3 is much simpler. It checks both candidate buckets for a given item; if any fingerprint matches in any bucket, one copy of that matched fingerprint is removed from that bucket.

Deletion does not have to clean the entry after deleting an item. It also avoids the "false deletion" problem when two items share one candidate bucket and also have the same fingerprint. For example, if both item x and y reside in bucket i_1 and collide on fingerprint f, partial-key cuckoo hashing ensures that they both can also reside in bucket i_2 because $i_2 = i_1 \oplus \text{hash}(f)$. When deleting x, it does not matter if the process removes the copy of f added when inserting x or y. After x is deleted, y is still perceived as a set member because there is a corresponding fingerprint left in either bucket i_1 and i_2. An important consequence of this is that the false-positive behavior of the filter is unchanged after deletion. (In the above example, y being in the table causes false positives for lookups of x, by definition: they share the same bucket and fingerprint.) This is the expected false-positive behavior of an approximate set membership data structure, and its probability remains bounded by ϵ.

Note that, to delete an item x safely, it must have been previously inserted. Otherwise, deleting a non-inserted item might unintentionally remove a real, different item that happens to share the same fingerprint. This requirement also holds true for all other deletion-supporting filters.

[1] A *naive implementation* of d-left counting Bloom filters has a "false deletion" problem. A d-left counting Bloom filter maps each item to d candidate buckets, each from a partition of the table, and stores the fingerprint in one of its d buckets. If two different items share one and only one bucket, and they have the same fingerprint (which is likely to happen when the fingerprints are small), directly deleting the fingerprint from this specific bucket may cause a wrong item to be removed. To address this issue, d-left counting Bloom filters use an additional counter in each table entry and additional indirections [5].

[2] In quotient filters, all fingerprints having the same quotient (i.e., the q most significant bits) must be stored in contiguous entries according to their numerical order. Thus, removing one fingerprint from a cluster of fingerprints must shift all the fingerprints after the hole by one slot, and also modify their meta-data bits [2].

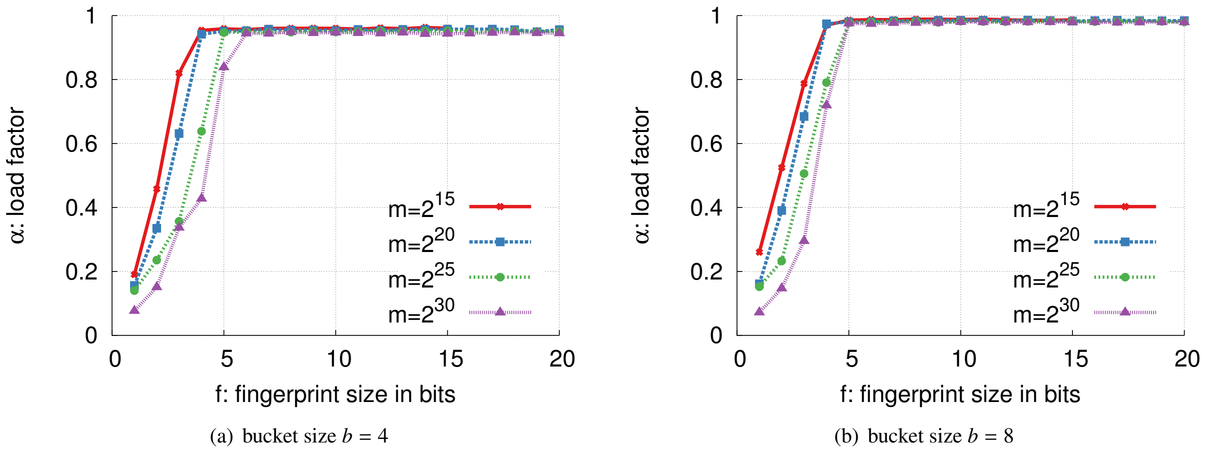


Figure 2: Load factor α achieved by using f-bit fingerprint using partial-key cuckoo hashing, in tables of different sizes ($m = 2^{15}, 2^{20}, 2^{25}$ and 2^{30} buckets). Short fingerprints suffice to approach the optimal load factor achieved by using two fully independent hash functions. $\alpha = 0$ means empty and 1 means completely full. Each point is the minimal load factor seen in 10 independent runs.

4. ASYMPTOTIC BEHAVIOR

Here we show that using partial-key cuckoo hashing to store fingerprints in a cuckoo filter to leads to a lower bound on fingerprint sizes that grows slowly with the filter size. This contrasts with other approaches (such as fingerprints and perfect hashing for a static Bloom filter, previously discussed) where the fingerprint size depended only on the desired false positive probability. While this might seem like a negative, in practice the effect seems unimportant. Empirically, a filter containing up to 4 billion items can effectively fill its hash table with loads of 95% when each bucket holds four fingerprints that are 6 bits or larger.

The notation used for our analysis in this and the next section is:

ϵ	target false positive rate
f	fingerprint length in bits
α	load factor ($0 \le \alpha \le 1$)
b	number of entries per bucket
m	number of buckets
n	number of items
C	average bits per item

Minimum Fingerprint Size: Our proposed partial-key cuckoo hashing derives the alternate bucket of a given item based on its current location and the fingerprint (Section 3). As a result, the candidate buckets for each item are not independent. For example, assume an item can be placed in bucket i_1 or i_2. According to Eq. (1), the number of possible values of i_2 is at most 2^f when using f-bit fingerprints. Using 1-byte fingerprints, given i_1 there are only up to $2^f = 256$ different possible values of i_2. For a table of m buckets, when $2^f < m$ (or equivalently $f < \log_2 m$ bits), the choice of i_2 is only a subset of all m buckets of the entire hash table.

Intuitively, if the fingerprints are sufficiently long, partial-key cuckoo hashing could still be a good approximation to standard cuckoo hashing; however, if the hash table is very large and stores relatively short fingerprints, the chance of insertion failures will increase due to hash collisions, which may reduce the table occupancy. This situation may arise when a cuckoo filter targets a large number of items but only a moderately low false positive rate. In the following, we determine analytically a lower bound of the probability of insertion failure.

Let us first derive the probability that a given set of q items collide in the same two buckets. Assume the first item x has its first bucket i_1 and a fingerprint t_x. If the other $q-1$ items have the same two buckets as this item x, they must [3] (1) have the same fingerprint t_x, which occurs with probability $1/2^f$; and (2) have their first bucket either i_1 or $i_1 \oplus h(t_x)$, which occurs with probability $2/m$. Therefore the probability of such q items sharing the same two buckets is $\left(2/m \cdot 1/2^f\right)^{q-1}$.

Now consider a construction process that inserts n random items to an empty table of $m = cn$ buckets for a constant c and constant bucket size b. Whenever there are $q = 2b+1$ items mapped into the same two buckets, the insertion fails. This probability provides a lower bound for failure (and, we believe, dominates the failure probability of this construction process, although we do not prove this and do not need to in order to obtain a lower bound). Since there are in total $\binom{n}{2b+1}$ different possible sets of $2b+1$ items out of n items, the expected number of groups of $2b+1$ items colliding during

[3]Here we assume hashing the fingerprints (i.e., $t_x \rightarrow h(t_x)$) is collision-free, because in practice t_x is a few bits and $h(t_x)$ is much longer.

the construction process is

$$\binom{n}{2b+1}\left(\frac{2}{2^f \cdot m}\right)^{2b} = \binom{n}{2b+1}\left(\frac{2}{2^f \cdot cn}\right)^{2b} = \Omega\left(\frac{n}{4^{bf}}\right) \quad (3)$$

We conclude that 4^{bf} must be $\Omega(n)$ to avoid a non-trivial probability of failure, as otherwise this expectation is $\Omega(1)$. Therefore, the fingerprint size must be $f = \Omega(\log n/b)$ bits.

This result seems somewhat unfortunate, as the number of bits required for the fingerprint is $\Omega(\log n)$; recall that Bloom filters use a constant (approximately $\ln(1/\epsilon)$ bits) per item. We might therefore have concerns about the scalability of this approach. As we show next, however, practical applications of cuckoo filters are saved by the b factor in the denominator of the lower bound: as long as we use reasonably sized buckets, the fingerprint size can remain small.

Empirical Evaluation: Figure 2 shows the load factor achieved with partial-key cuckoo hashing as we vary the fingerprint size f, bucket size b, and number of buckets in the table m. For the experiments, we varied the fingerprint size f from 1 to 20 bits. Random 64-bit keys are inserted to an empty filter until a single insertion relocates existing fingerprints more than 500 times before finding an empty entry (our "full" condition), then we stop and measure achieved load factor α. We run this experiment ten times for filters with $m = 2^{15}, 2^{20}, 2^{25}$, and 2^{30} buckets, and measured their minimum load over the ten trials. We did not use larger tables due to the memory constraint of our testbed machine.

As shown in Figure 2, across different configurations, filters with $b = 4$ could be filled to 95% occupancy, and with $b = 8$ could be filled to 98%, with sufficiently long fingerprints. After that, increasing the fingerprint size has almost no return in term of improving the load factor (but of course it reduces the false positive rate). As suggested by the theory, the minimum f required to achieve close-to-optimal occupancy increases as the filter becomes larger. Also, comparing Figure 2(a) and Figure 2(b), the minimum f for high occupancy is reduced when bucket size b becomes larger, as the theory also suggests. Overall, short fingerprints appear to suffice for realistic sized sets of items. Even when $b = 4$ and $m = 2^{30}$, so the table could contain up to four billion items, once fingerprints exceed six bits, α approaches the "optimal load factor" that is obtained in experiments using two fully independent hash functions.

Insights: The lower bound of fingerprint size derived in Eq. (3), together with the empirical results shown in Figure 2, give important insights into the cuckoo filter. While in theory the space cost of cuckoo filters is "worse" than Bloom filters—$\Omega(\log n)$ versus a constant—the constant terms are very important in this setting. For a Bloom filter, achieving $\epsilon = 1\%$ requires roughly 10 bits per item, regardless of whether one thousand, one million, or billion items are stored. In contrast, cuckoo filters require longer fingerprints to retain the same high space efficiency of their hash tables, but lower false positive rates are achieved accordingly. The $\Omega(\log n)$ bits per fingerprint, as also predicted by the theory, grows slowly if the bucket size b is sufficiently large. We find that, for practical purposes, it can be treated as a reasonable-sized constant for implementation. Figure 2 shows that for cuckoo filters targeting a few billion items, 6-bit fingerprints are sufficient to ensure very high utilization of the hash table.

As a heuristic, partial-key cuckoo hashing is very efficient, as we show further in Section 7. Several theoretical questions regarding this technique, however, remain open for future study, including proving bounds on the cost of inserting a new item and studying how much independence is required of the hash functions.

5. SPACE OPTIMIZATIONS

The basic algorithms for cuckoo filter operations `Insert`, `Lookup`, and `Delete` presented in Section 3 are independent of the hash table configuration (e.g., how many entries each bucket has). However, choosing the right parameters for cuckoo filters can significantly affect space efficiency. This section focuses on optimizing the space efficiency of cuckoo filters, through parameter choices and additional mechanisms.

Space efficiency is measured by the average number of bits to represent each item in a full filter, derived by the table size divided by the total number of items that a filter can effectively store. Recall that, although each entry of the hash table stores one fingerprint, not all entries are occupied: there must be some slack in the table for the cuckoo filter or there will be failures when inserting items. As a result, each item effectively costs more to store than a fingerprint: if each fingerprint is f bits and the hash table has a load factor of α, then the amortized space cost C for each item is

$$C = \frac{\text{table size}}{\text{\# of items}} = \frac{f \cdot (\text{\# of entries})}{\alpha \cdot (\text{\# of entries})} = \frac{f}{\alpha} \quad \text{bits.} \quad (4)$$

As we will show, both f and α are related to the bucket size b. The following section studies how to (approximately) minimize Eq. (4) given a target false positive rate ϵ by choosing the optimal bucket size b.

5.1 Optimal Bucket Size

Keeping a cuckoo filter's total size constant but changing the bucket size leads to two consequences:

(1) Larger buckets improve table occupancy (i.e., higher $b \rightarrow$ higher α). With $k = 2$ hash functions, the load factor α is 50% when the bucket size $b = 1$ (i.e., the hash table is directly mapped), but increases to 84%, 95% or 98% respectively using bucket size $b = 2$, 4 or 8.

(2) Larger buckets require longer fingerprints to retain the same false positive rate (i.e., higher $b \rightarrow$ higher f). With larger buckets, each lookup checks more entries and thus has a higher chance to see fingerprint collisions. In the worst case of looking up a non-existent item, a query must probe two buckets where each bucket can have b entries. (While not all of these buckets may be filled, we analyze here

	bits per item	**load factor α**	**avg. # memory references / lookup** positive query	negative query
space-optimized Bloom filter	$1.44\log_2(1/\epsilon)$	–	$\log_2(1/\epsilon)$	2
cuckoo filter	$(\log_2(1/\epsilon)+3)/\alpha$	95.5%	2	2
cuckoo filter w/ semi-sort	$(\log_2(1/\epsilon)+2)/\alpha$	95.5%	2	2

Table 2: Space and lookup cost of Bloom filters and three cuckoo filters.

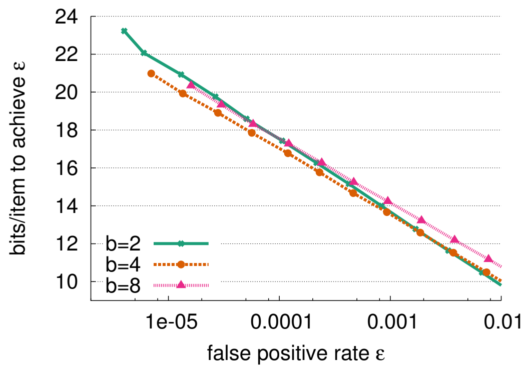


Figure 3: Amortized space cost per item vs. measured false positive rate, with different bucket size $b = 2, 4, 8$. Each point is the average of 10 runs

the worst case in which they are; this gives us a reasonably accurate estimate for a table that is 95% full.) In each entry, the probability that a query is matched against one stored fingerprint and returns a false-positive successful match is *at most* $1/2^f$. After making $2b$ such comparisons, the upper bound of the total probability of a false fingerprint hit is

$$1 - (1 - 1/2^f)^{2b} \approx 2b/2^f, \tag{5}$$

which is proportional to the bucket size b. To retain the target false positive rate ϵ, the filter ensures $2b/2^f \le \epsilon$, thus the minimal fingerprint size required is approximately:

$$f \ge \lceil \log_2(2b/\epsilon) \rceil = \lceil \log_2(1/\epsilon) + \log_2(2b) \rceil \quad \text{bits.} \tag{6}$$

Upper Bound of Space Cost As we have shown, both f and α depend on the bucket size b. The average space cost C by Eq. (4) is bound by:

$$C \le \lceil \log_2(1/\epsilon) + \log_2(2b) \rceil / \alpha, \tag{7}$$

where α increases with b. For example, when $b = 4$ so $1/\alpha \approx 1.05$, Eq. (7) shows cuckoo filters are asymptotically better (by a constant factor) than Bloom filters, which require $1.44\log_2(1/\epsilon)$ bits or more for each item.

Optimal bucket size b To compare the space efficiency by using different bucket sizes b, we run experiments that first construct cuckoo hash tables by partial-key cuckoo hashing with different fingerprint sizes, and measure the amortized space cost per item and their achieved false positive rates. As shown in Figure 3, the space-optimal bucket size depends on the target false positive rate ϵ: when $\epsilon > 0.002$, having two entries per bucket yields slightly better results than using four entries per bucket; when ϵ decreases to $0.00001 < \epsilon \le 0.002$, four entries per bucket minimizes space.

In summary, we choose (2, 4)-cuckoo filter (i.e., each item has two candidate buckets and each bucket has up to four fingerprints) as the default configuration, because it achieves the best or close-to-best space efficiency for the false positive rates that most practical applications [6] may be interested in. In the following, we present a technique that further saves space for cuckoo filters with $b = 4$ by encoding each bucket.

5.2 Semi-sorting Buckets to Save Space

This subsection describes a technique for cuckoo filters with $b = 4$ entries per bucket that saves one bit per item. This optimization is based on the fact that the order of fingerprints within a bucket does not affect the query results. Based on this observation, we can compress each bucket by first sorting its fingerprints and then encoding the sequence of sorted fingerprints. This scheme is similar to the "semi-sorting buckets" optimization used in [5].

The following example illustrates how the compression saves space. Assume each bucket contains $b = 4$ fingerprints and each fingerprint is $f = 4$ bits (more general cases will be discussed later). An uncompressed bucket occupies $4 \times 4 = 16$ bits. However, if we sort all four 4-bit fingerprints stored in this bucket (empty entries are treated as storing fingerprints of value "0"), there are only 3876 possible outcomes in total (the number of unique combinations with replacement). If we precompute and store these 3876 possible bucket-values in an extra table, and replace the original bucket with an index into this precomputed table, then each original bucket can be represented by a 12-bit index ($2^{12} = 4096 > 3876$) rather than 16 bits, saving 1 bit per fingerprint.[4]

Note that this permutation-based encoding (i.e., indexing all possible outcomes) requires extra encoding/decoding tables and indirections on each lookup. Therefore, to achieve high lookup performance it is important to make the encoding/decoding table small enough to fit in cache. As a result, our "semi-sorting" optimization only apply this technique for tables with buckets of four entries. Also, when fingerprints are larger than four bits, only the four most significant bits of

[4]If the cuckoo filter does not need to support deletion, then it can ignore duplicate entries in the fingerprint list, creating the potential for saving an additional fraction of a bit per entry.

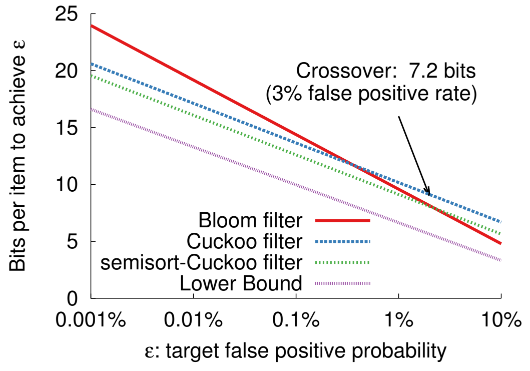


Figure 4: False positive rate vs. space cost per element. For low false positive rates ($< 3\%$), cuckoo filters require fewer bits per element than the space-optimized Bloom filters. The load factors to calculate space cost of cuckoo filters are obtained empirically.

each fingerprint are encoded; the remainder are stored directly and separately.

6. COMPARISON WITH BLOOM FILTER

We compare Bloom filters and cuckoo filters using the metrics shown in Table 2 and several additional factors.

Space Efficiency: Table 2 compares space-optimized Bloom filters and cuckoo filters with and without semi-sorting. Figure 4 further shows the bits to represent one item required by these schemes, when ϵ varies from 0.001% to 10%. The information theoretical bound requires $\log_2(1/\epsilon)$ bits for each item, and an optimal Bloom filter uses $1.44\log_2(1/\epsilon)$ bits per item, for a 44% overhead. Thus cuckoo filters with semi-sorting are more space efficient than Bloom filters when $\epsilon < 3\%$.

Number of Memory Accesses: For Bloom filters with k hash functions, a *positive query* must read k bits from the bit array. For space-optimized Bloom filters that require $k = \log_2(1/\epsilon)$, as ϵ gets smaller, positive queries must probe more bits and are likely to incur more cache line misses when reading each bit. For example, $k = 2$ when $\epsilon = 25\%$, but k is 7 when $\epsilon = 1\%$, which is more commonly seen in practice. A *negative query* to a space optimized Bloom filter reads two bits on average before it returns, because half of the bits are set. Any query to a cuckoo filter, positive or negative, always reads a fixed number of buckets, resulting in (at most) two cache line misses.

Value Association: Cuckoo filters can be extended to also return an associated value (stored external to the filter) for each matched fingerprint. This property of cuckoo filters provides an approximate table lookup mechanism, which returns $1 + \epsilon$ values on average for each existing item (as it can match more than one fingerprint due to false positive hits) and on average ϵ values for each non-existing item. Standard Bloom filters do not offer this functionality. Although variants like Bloomier filters can generalize Bloom filters to represent functions, these structures are more complex and require more space than cuckoo filters [7].

Maximum Capacity: Cuckoo filters have a load threshold. After reaching the maximum feasible load factor, insertions are non-trivially and increasingly likely to fail, so the hash table must expand in order to store more items. In contrast, one can keep inserting new items into a Bloom filter, at the cost of an increasing false positive rate. To maintain the same target false positive rate, the Bloom filter must also expand.

Limited Duplicates: If the cuckoo filter supports deletion, it must store multiple copies of the same item. Inserting the same item $kb + 1$ times will cause the insertion to fail. This is similar to counting Bloom filters where duplicate insertion causes counter overflow. However, there is no effect from inserting identical items multiple times into Bloom filters, or a non-deletable cuckoo filter.

7. EVALUATION

Our implementation[5] consists of approximately 500 lines of C++ code for standard cuckoo filters, and 500 lines for the support of the "semi-sorting" optimization presented in Section 5.2. In the following, we denote a basic cuckoo filter as "CF", and a cuckoo filter with semi-sorting as "ss-CF". In addition to cuckoo filters, we implemented four other filters for comparison:

- Standard Bloom filter (BF) [3]: We evaluated standard Bloom filters as the baseline. In all of our experiments, the number of hash functions k are configured to achieve the lowest false positive rate, based on the filter size and the total number of items. In addition, a performance optimization is applied to speed up lookups and inserts by doing less hashing [16]. Each insert or lookup only requires two hash functions $h_1(x)$ and $h_2(x)$, and then uses these two hashes to simulate the additional $k - 2$ hash functions in the form of

$$g_i(x) = h_1(x) + i \cdot h_2(x).$$

- Blocked Bloom filter (blk-BF) [22]: Each filter consists of an array of blocks and each block is a small Bloom filter. The size of each block is 64 bytes to match a CPU cache line in our testbed. For each small Bloom filter, we also apply the same optimization of simulating multiple hash functions as in standard Bloom filters.
- Quotient filter (QF) [2]: We evaluated our own implementation of quotient filters[6]. This filter stores three extra bits for each item as the meta-data to help locate items. Because the performance of quotient filters degrades as they become more loaded, we set the maximum load factor to be 90% as evaluated in [2].
- d-left counting Bloom filter (dl-CBF) [5]: The filter is configured to have $d = 4$ hash tables. All hash tables

[5]https://github.com/efficient/cuckoofilter

[6]The source code from the original authors was not publicly available for comparison due to license issues.

metrics	CF	ss-CF	BF	blk-BF	QF	dl-CBF
# of items (million)	127.78	**128.04**	123.89	123.89	120.80	103.82
bits per item	12.60	**12.58**	13.00	13.00	13.33	15.51
false positive rate	0.19%	**0.09%**	0.19%	0.43%	0.18%	0.76%
constr. speed (million keys/sec)	5.00	3.13	3.91	**7.64**	1.91	4.78

Table 3: Space efficiency and construction speed. All filters are 192MB. Entries in bold are the best among the row.

have the same number of buckets; each bucket has four entries.

We emphasize that the standard and blocked Bloom filters do not support deletion, and thus are compared as a baseline.

Experiment Setup: All the items to insert are pre-generated 64-bit integers from random number generators. We did not eliminate duplicated items because the probability of duplicates is very small.

On each query, all filters first generate a 64-bit hash of the item using CityHash [1]. Each filter then partitions these 64 bits in this hash as it needed. For example, Bloom filters treat the high 32 bits and low 32 bits as the first two independent hashes respectively, then use these two 32-bit values to calculate the other $k-2$ hashes. The time to compute the 64-bit hash (about 20 ns per item in our testbed) is included in our measurement. All experiments use a machine with two Intel Xeon processors (L5640@2.27GHz, 12MB L3 cache) and 32 GB DRAM.

Metrics: To fully understand how different filters realize the trade-offs in function, space and performance, we compare above filters by the following metrics:

- *Space efficiency*: measured by the filter size in bits divided by the number of items a full filter contains.
- *Achieved false positive rate*: measured by querying a filter with non-existing items and counting the fraction of positive return values.
- *Construction rate*: measured by the number of items that a full filter contains divided by the time to construct this full filter from empty.
- *Lookup, Insert and Delete throughput*: measured by the average number of operations a filter can perform per second. The value can depend on the workload and the occupancy of the filter.

7.1 Achieved False Positive Rate

We first evaluate the space efficiency and false positive rates. In each run, all filters are configured to have the same size (192 MB). Bloom filters are configured to use nine hash functions, which minimizes the false positive rate with thirteen bits per item. For cuckoo filters, their hash tables have $m = 2^{25}$ buckets each consisting of four 12-bit entries. The d-left counting Bloom filter have the same number of hash table entries, but divided into $d = 4$ partitions. Quotient filter also has 2^{27} entries where each entry stores 3-bit meta-data and a 9-bit remainder.

Each filter is initially empty and items are placed until either the filter sees an insert failure (for CF, and dl-CBF), or it has reached the target capacity limit (for BF, blk-BF, and QF). The construction rate and false positive rate of different filters are shown in Table 3.

Among all filters, the ss-CF achieves the lowest false positive rate. Using about the same amount of space (12.60 bits/item), enabling semi-sorting can encode one more bit into each item's fingerprint and thus halve the false positive rate from 0.19% to 0.09%, On the other hand, semi-sorting requires encoding and decoding when accessing each bucket, and thus the construction rate is reduced from 5.00 to 3.13 million items per second.

The BF and blk-BF both use 13.00 bits per item with $k = 9$ hash functions, but the false positive rate of the blocked filter is 2× higher than the BF and 4× higher than the best CF. This difference is because the blk-BF assigns each item to a single block by hashing and an imbalanced mapping will create "hot" blocks that serve more items than average and "cold" blocks that are less utilized. Unfortunately, such an imbalanced assignment happens across blocks even when strong hash functions are used [18], which increases the overall false positive rate. On the other hand, by operating in a single cache line for any query, the blk-BF achieves the highest construction rate.

The QF spends more bits per item than BFs and CFs, achieving the second best false positive rate. Due to the cost of encoding and decoding each bucket, its construction rate is the lowest.

Finally, the dl-CBF sees insert failures and stops construction when the entire table is about 78% full, thus storing many fewer items. Its achieved false positive rate is much worse than the other filters because each lookup must check 16 entries, hence having a higher chance of hash collisions.

7.2 Lookup Performance

Different Workloads We next benchmark lookup performance after the filters are filled. This section compares the lookup throughput and latency with varying workloads. The workload is characterized by the fraction of *positive queries* (i.e., items in the table) and *negative queries* (i.e., items not in the table), which can affect the lookup speed. We vary the fraction p of positive queries in the input workload from 0% (all queries are negative) to 100% (all queries are positive).

The benchmark result of lookup throughput is shown in Figure 5. Each filter occupies 192 MB, much larger than the L3 cache (20 MB) in our testbed.

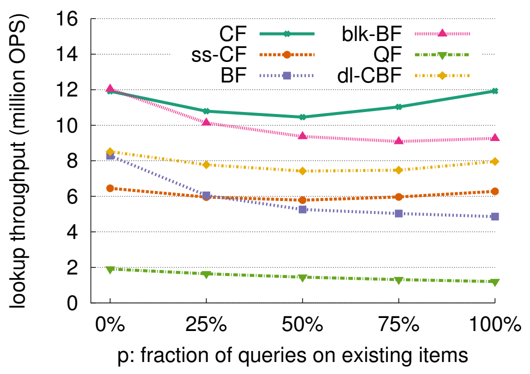


Figure 5: Lookup performance when a filter achieves its capacity. Each point is the average of 10 runs.

The blk-BF performs well when all queries are negative, because each lookup can return immediately after fetching the first "0" bit. However, its performance declines when more queries are positive, because it must read additional bits as part of the lookup. The throughput of BF changes similarly when p increases, but is about 4 MOPS slower. This is because the BF may incur multiple cache misses to complete one lookup whereas the blocked version can always operate in one cache line and have at most one cache miss for each lookup.

In contrast, a CF always fetches two buckets[7], and thus achieves the same high performance when queries are 100% positive and 100% negative. The performance drops slightly when $p = 50\%$ because the CPU's branch prediction is least accurate (the probability of matching or not matching is exactly 1/2). With semi-sorting, the throughput of CF shows a similar trend when the fraction of positive queries increases, but it is lower due to the extra decoding overhead when reading each bucket. In return for the performance penalty, the semi-sorting version reduces the false positive rate by a factor of two compared to the standard cuckoo filter. However, the ss-CF still outperforms BFs when more than 25% of queries are positive.

The QF performs the worst among all filters. When a QF is 90% filled, a lookup must search a long chain of table entries and decode each of them for the target item.

The dl-CBF outperforms the ss-CF, but 30% slower than a BF. It also keeps about the same performance when serving all negative queries and all positive queries, because only a constant number of entries are searched on each lookup.

Different Occupancy In this experiment, we measure the lookup throughput when the these filters are filled at different levels of occupancy. We vary their load factor α of each filter from 0 (empty) to its maximum occupancy. Figure 6 shows the average instantaneous lookup throughput when all queries are negative (i.e., for non-existing items) and all queries are positive (i.e., for existing items).

The throughput of CF and ss-CF is mostly stable across different load factor levels on both negative and positive queries. This is because the total number of entries to read and compare remains constant even as more items are inserted.

In contrast, the throughput of QF decreases substantially when more loaded. This filter searches an increasingly long chain of entries for the target item as the load factor grows.

Both BF and blk-BF behave differently when serving negative and positive queries. For positive queries, they must always check in total k bits, no matter how many items have been inserted, thus providing constant lookup throughput; while for negative queries, when the filter is less loaded, fewer bits are set and a lookup can return earlier when seeing a "0".

The dl-CBF behaves differently from the BF. When all lookups are negative, it ensures constant throughput like the CF, because a total of 16 entries from four buckets must be searched, no matter how many items this filter contains. For positive queries, if there are fewer items inserted, the lookup may return earlier before all four buckets are checked; however, this difference becomes negligible after the dl-CBF is about 20% filled.

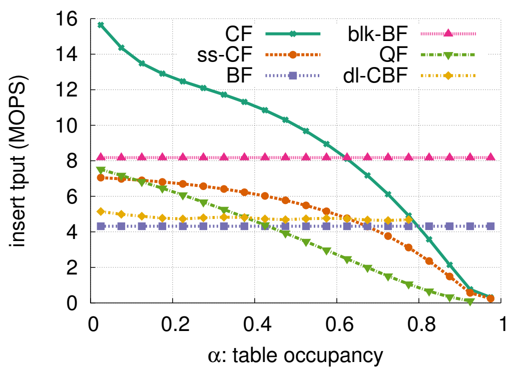


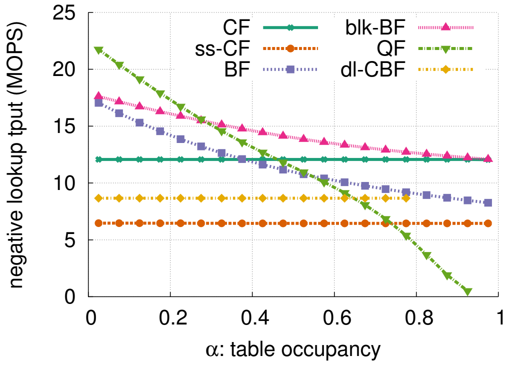
Figure 7: Insert throughput at different occupancy. Insert random keys in a large universe until each data structure achieves its designed capacity. Each point is the average of 10 runs.

7.3 Insert Performance

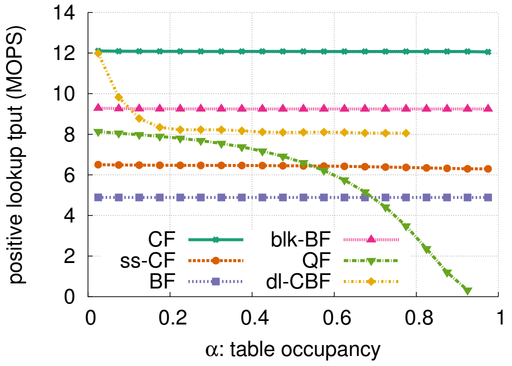
The overall construction speed, measured based on the total number of items a full filter contains and the total time to insert these items, is shown in Table 3. We also study the instantaneous insert throughput across the construction process. Namely, we measure the insert throughput of different filters when they are at levels of load factors, as shown in Figure 7.

In contrast to the lookup throughput shown in Figure 6, both types of CF have decreasing insert throughput when they are more filled (though their overall construction speed is still high), while both BF and blk-BF ensure almost constant insert throughput. The CF may have to move a sequence of existing keys recursively before successfully inserting a new item, and this process becomes more expensive when the load factor grows higher. In contrast, both Bloom filters always

[7] Instead of checking each item's two buckets one by one, our implementation applies a performance optimization that tries to issue two memory loads together to hide the latency of the second read.



(a) Lookup random keys from a large universe



(b) Lookup random keys existing in the filter

Figure 6: Lookup throughput (MOPS) at different occupancy. Each point is the average of 10 runs.

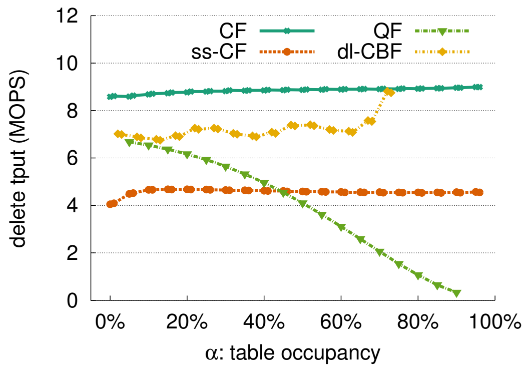


Figure 8: Delete-until-empty throughput (MOPS) at different occupancy. Each point is the average of 10 runs.

set k bits, regardless of the load factor.

The QF also has decreasing insert throughput, because it must shift a sequence of items before inserting a new item, and this sequence grows longer when the table is more filled.

The dl-CBF keeps constant throughput. For each insert, it must only find an empty entry in up to four buckets; if such an entry can not be found, the insert stops without relocating existing items as in cuckoo hashing. This is also why its maximum load factor is no more than 80%.

7.4 Delete Performance

Figure 8 compares the delete performance among filters supporting deletion. The experiment deletes keys from an initially full filter until it is empty. The CF achieves the highest throughput. Both CF and ss-CF provide stable performance through the entire process. The dl-CBF performs the second best among all filters. The QF is the slowest when close to full, but becomes faster than ss-CF when close to empty.

Evaluation Summary: The CF ensures high and stable lookup performance for different workloads and at different levels of occupancy. Its insert throughput declines as the filter is more filled, but the overall construction rate is still faster than other filters except the blk-BF. Enabling semi-sorting makes cuckoo filters more space-efficient than space-optimized BFs. It also makes lookups, inserts, and deletes slower, but still faster than conventional BFs.

8. CONCLUSION

Cuckoo filters are a new data structure for approximate set membership queries that can be used for many networking problems formerly solved using Bloom filters. Cuckoo filters improve upon Bloom filters in three ways: (1) support for deleting items dynamically; (2) better lookup performance; and (3) better space efficiency for applications requiring low false positive rates ($\epsilon < 3\%$). A cuckoo filter stores the fingerprints of a set of items based on cuckoo hashing, thus achieving high space occupancy. As a further key contribution, we have applied *partial-key cuckoo hashing*, which makes cuckoo filters significantly more efficient by allowing relocation based on only the stored fingerprint. Our configuration exploration suggests that the cuckoo filter, which uses buckets of size 4, will perform well for a wide range of applications, although appealingly cuckoo filter parameters can be easily varied for application-dependent tuning.

While we expect that further extensions and optimizations to cuckoo filters are possible and will further provide impetus for their use, the data structure as described is a fast and efficient building block already well-suited to the practical demands of networking and distributed systems.

9. ACKNOWLEDGMENTS

This work is supported by funding from Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC), and National Science Foundation under awards CCF-0964474, CNS-1040801, CCF-1320231, CNS-1228598, and IIS-0964473. We gratefully acknowledge the CoNEXT reviewers for their feedback and suggestions; Rasmus Pagh for discussions on analyzing space usage of cuckoo filters that led to many of the insights about the asymptotic behavior; Hyeontaek Lim for code optimizations; and Iulian Moraru, Andrew Pavlo, Peter Steenkiste for comments on writing.

10. REFERENCES

[1] CityHash. https://code.google.com/p/cityhash/.

[2] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. *PVLDB*, 5(11):1627–1637, 2012.

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

[4] F. Bonomi, M. Mitzenmacher, and R. Panigrahy. Beyond Bloom filters: From approximate membership checks to approximate state machines. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.

[5] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *14th Annual European Symposium on Algorithms, LNCS 4168*, pages 684–695, 2006.

[6] A. Broder, M. Mitzenmacher, and A. Broder. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, volume 1, pages 636–646, 2002.

[7] B. Chazelle, J. Kilian, R. Rubinfeld, A. Tal, and O. Boy. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of SODA*, pages 30–39, 2004.

[8] J. G. Cleary. Compact Hash Tables Using Bidirectional Linear Probing. *IEEE Transactions on Computer*, C-33(9), Sept. 1984.

[9] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using Bloom filters. In *Proc. ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003.

[10] M. Dietzfelbinger and C. Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1):47–68, 2007.

[11] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. 10th USENIX NSDI*, Lombard, IL, Apr. 2013.

[12] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. In *Proc. ACM SIGCOMM*, Vancouver, BC, Canada, Sept. 1998.

[13] N. Fountoulakis, M. Khosla, and K. Panagiotou. The multiple-orientability thresholds for random hypergraphs. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1222–1236. SIAM, 2011.

[14] N. Hua, H. C. Zhao, B. Lin, and J. J. Xu. Rank-Indexed Hashing: A Compact Construction of Bloom Filters and Variants. In *Proc. of IEEE Int'l Conf. on Network Protocols (ICNP) 2008*, Orlando, Florida, USA, Oct. 2008.

[15] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander. Lipsin: Line speed publish/subscribe internetworking. In *Proc. ACM SIGCOMM*, Barcelona, Spain, Aug. 2009.

[16] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms*, 33(2):187–218, 2008.

[17] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[18] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing*, pages 255–312. Kluwer, 2000.

[19] M. Mitzenmacher and B. Vocking. The asymptotics of selecting the shortest of two, improved. In *Proc. the Annual Allerton Conference on Communication Control and Computing*, volume 37, pages 326–327, 1999.

[20] A. Pagh, R. Pagh, and S. S. Rao. An optimal bloom filter replacement. In *Proc. ASM-SIAM Symposium on Discrete Algorithms, SODA*, 2005.

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

[22] F. Putze, P. Sanders, and S. Johannes. Cache-, hash- and space-efficient bloom filters. In *Experimental Algorithms*, pages 108–121. Springer Berlin / Heidelberg, 2007.

[23] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proc. the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 123–133, Washington, DC, USA, 2007.

[24] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: An aid to network processing. In *Proc. ACM SIGCOMM*, pages 181–192, Philadelphia, PA, Aug. 2005.

[25] M. Yu, A. Fabrikant, and J. Rexford. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proc. CoNEXT*, Dec. 2009.

[26] D. Zhou, B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2013.