Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters

Minmei Wang†, Mingxun Zhou‡, Shouqian Shi†, and Chen Qian† *
mwang107@ucsc.edu, zhoumingxun@pku.edu.cn, sshi27@ucsc.edu and cqian12@ucsc.edu
† University of California, Santa Cruz, USA
‡ Peking University, China.

ABSTRACT

We present vacuum filters, a type of data structures to support approximate membership queries. Vacuum filters cost the smallest space among all known AMQ data structures and provide higher insertion and lookup throughput in most situations. Hence they can be used as the replacement of the widely used Bloom filters and cuckoo filters. Similar to cuckoo filters, vacuum filters also store item fingerprints in a table. The memory-efficiency and throughput improvements are from the innovation of a table insertion and fingerprint eviction strategy that achieves both high load factor and data locality without any restriction of the table size. In addition, we propose a new update framework to resolve two difficult problems for AMQ structures under dynamics, namely duplicate insertions and set resizing. The experiments show that vacuum filters can achieve 25% less space in average and similar throughput compared to cuckoo filters, and 15% less space and >10x throughput compared to Bloom filters, with same false positive rates. AMQ data structures are widely used in various layers of computer systems and networks and are usually hosted in platforms where memory is limited and precious. Hence the improvements brought by vacuum filters can be considered significant.

PVLDB Reference Format:
Minmei Wang, Mingxun Zhou, Shouqian Shi, Chen Qian. Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters. *PVLDB*, 13(2): 197-210, 2019.
DOI: https://doi.org/10.14778/3364324.3364333

1. INTRODUCTION

Approximate membership queries (AMQs) rely on space-efficient data structures to decide whether a queried data item is in a large set of items. These data structures (called the AMQ structures hereafter), such as the well-known Bloom filters [13], are essential components of numerous practical computer software and systems, such as Google Bigtable [18], Apache Cassandra [28], Google Chrome, the content distribution network Akamai [33], Bitcoin [5], and Ethereum [9]. The most attractive feature of the AMQ structures is their **memory efficiency**, with the trade-off on allowing few false positives. Compared to an error-free representation of a set of items, such as a hash table, an AMQ structure can work on devices with limited memory resource (network routers, switches, and IoT devices), or in a higher level of the memory hierarchy (cache vs main memory, or main memory vs disk). For example, Bloom filters have been extensively used in reducing disk I/O [13, 18], avoiding unnecessary remote content lookups [24, 33], network functions [21, 44, 46, 31, 50], services on mobile and IoT devices [17, 30], and many data management applications including distributed joins and semi-joins [37], indexing [11], auxiliary metadata [18, 20], and query processing problems [29]. The recently proposed cuckoo filters [23] improves Bloom filters in ideal-case memory-efficiency and enabling deletions.

This paper presents *vacuum filters*, a type of AMQ data structures that cost the **smallest space among all known methods**, i.e., more memory-efficient than Bloom filters, cuckoo filters, and other AMQ structures, when the false positive rate $\epsilon < 3\%$. The name is from vacuum packing which uses the least space to pack items. To understand the space and false positive rate tradeoffs of AMQ data structures, we show the empirical results to better illustrate our idea and contributions. Fig. 1 shows the memory cost of the most representative data structures for AMQs (in bits per item), including the Bloom filters [13], Cuckoo filters [23], and vacuum filters (this work). The false positive ϵ is set to 0.01%, a common requirement of many applications [23, 16]. Compared to Bloom, vacuum filters can reduce the memory

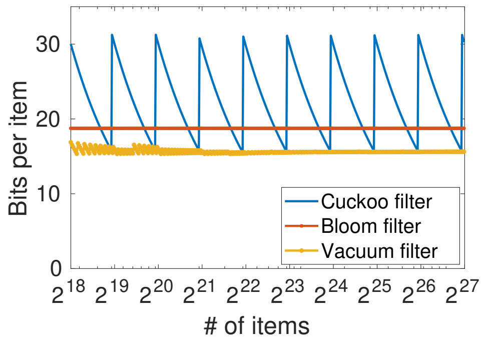


Figure 1: Bits per item vs. # of items

*M. Wang and M. Zhou contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 13, No. 2
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3364324.3364333

Table 1: Comparison of AMQ structures in space, throughput, and supporting dynamics. D: deletions; I: duplicate insertions; R: set resizing. Results are based on the experiments when false positive rate ϵ from 0.01% to 0.1%.

Data structure	Space	Thrpt	Dynamic
Bloom F. [13]	1x	1x	I
Dele. BF [43]	∼1.13x	∼1.95x	D,I
Blocked BF [41]	∼1.12x	∼5.8x	I
Count. BF [24]	4x	<1x	D, I
Count. Quotient F. [40]	0.87x to 1.45x	∼4.7x	D, I, R
Cuckoo F. [23]	0.84x to 1.5x	∼10x	D
Morton F. [16]	0.88x to 1.1x	∼6x	D
Vacuum F.	0.84x to 0.91x	∼11x	D, I, R

cost by more than 3 bits per item, resulting in $\sim$ 15% space saving. Compared to a cuckoo filter, a vacuum filter only costs 50% space in the worst cases and saves 25% space on average. Vacuum filters cost less memory than Bloom filters and other AMQ structures when $\epsilon < 3\%$. The advantage of space-efficient becomes larger when the target ϵ is smaller. Although Bloom filters cost less memory when $\epsilon > 3\%$, 3% is considered too much for most AMQ designs. It is common that applications require $\epsilon < 0.1\%$ or even $< 0.01\%$, such as the examples in [23, 16, 46, 31, 30, 12]. In addition, vacuum filters and cuckoo filters provides similar lookup throughput and both are much faster than other AMQ structures. We show a brief comparison among many popular AMQ structures in both space and lookup throughput in Table 1. Vacuum filters show the space and throughput advantages compared to all other methods.

Since AMQ data structures have been so widely used for many fields of computing technologies, we argue that $> 20\%$ space reduction compared to Bloom and cuckoo filters (average case) is a **significant contribution**. Considering that AMQ data structures are used in fast memory (SRAM, TCAM, cache) that are expensive and power-hungry, such memory efficiency becomes especially important to reduce device cost and avoid unnecessary device updates. In addition, fast memory is usually shared by multiple applications. For example, the SRAM on network switches and routers needs to supports network functions including forwarding tables [46, 48], multicast [31], traffic measurement [47, 50], packet caching [10], and load balancing [25, 35]. Reducing the AMQ cost benefits a variety of functions.

In addition, many practical applications require AMQ data structures to support dynamics, including insertions, deletions and set resizing. A well-known limitation of Bloom filters is that they cannot support deletions. To allow deletions, cuckoo filters store duplicate fingerprints. As a result, a cuckoo filter may crash due to table overflow when inserting duplicate items, which frequently happens in practical applications, as explained in Sec. 2. Moreover, neither Bloom nor cuckoo filters allow set resizing. The only AMQ method known for set resizing is quotient filters [12], which cost more space than Bloom and provide lower throughput than cuckoo.

Our important observation is that an AMQ data structure cannot support both deletions and duplicate insertions unless it allows reconstruction from the complete set. The major problem of current AMQ reconstruction is that it has to be executed in a large but slow memory because a reconstruction needs to access the complete item set. During that time, the AMQ structure running in the fast memory is unable to answer queries in order to achieve consistency. We resolve this problem by proposing a new update framework for vacuum filters called IUPR (Instant Updates and Periodical Reconstructions) to support deletions, duplicate insertions, and set resizing. IUPR is a good fit for legacy memory hierarchy and network architectures. For example, the vacuum filter can be run in the main memory of a query server, while the construction can be conducted on a back-end storage server to access the set of items. In the widely adopted software defined networking (SDN) paradigm [2, 38], the vacuum filter is used in network switches with limited memory, the construction program can be run in the SDN controller on a server, and the vacuum filter updates are achieved via standard APIs such as P4 [15].

In a nutshell, the vacuum filter has unique advantages: **1)** its memory cost is the smallest among existing methods; **2)** its query throughput is higher than most other solutions, only slightly lower than that of cuckoo in very few cases; and **3)** it supports practical dynamics using the memory hierarchy in practice. No existing method can achieve all of them. Since AMQ data structures have been widely adopted and memory efficiency is their most essential feature, the $> 20\%$ space reduction is **fundamental improvement rather than a small increment**.

The remaining paper is organized as follows. Section 2 presents the related work. Section 3 presents the detailed design of vacuum filters. We show the theoretical analysis results in Section 4. Section 5 presents the method to deal with dynamics. The implementation and evaluation results are shown in Section 6. We conclude this work in Section 7.

2. RELATED WORK

This section introduces existing AMQ data structures.

Bloom Filter. Bloom filters (BFs) [13] are the most well-known AMQ data structures. A Bloom filter represents a set of n items $S = x_1, x_2, ..., x_n$ by an array of m bits. Each item is mapped to k bits in the array uses k independent hash functions $h_1, h_2, ..., h_k$ and every mapped bit at location $h_i(x)$ is set to 1. To lookup whether an item x_i is in the set, the Bloom filter checks the values in the $h_i(x)$-th bit. If all bits are 1, the Bloom filter reports `true`. Otherwise, it reports `false`. A Bloom filter yields *false positives*. The false positive rate is $\epsilon = (1-e^{-kn/m})^k = (1-p)^k$. A Blocked Bloom filter (BBF) [41] divides a Bloom filter into multiple small blocks, each block fits into one cache-line. A BBF is cache-efficient because it only needs one cache miss for every query. One limitation of Bloom filter is that it cannot support deletions. Counting Bloom filters [24] allow deletions, which replace every bit by a counter to store the numbers of setting these bits to 1. Introducing counters significantly increases memory cost. The deletable Bloom filter (DlBF) [43] supports deletion by adding and maintaining a collision bitmap. Items can be deleted with a probability.

Quotient Filters. A quotient filters (QF) [12] uses a hash table to store the fingerprints of the inserted items. The hash table contains 2^q continuous entries. Every entry comprises one slot to store the fingerprint of an item and some extra flag bits to handle hash collision. The space cost of QF is larger than that of BFs. Counting Quotient Filter [40](CQF) improves QF from throughput and memory usage perspectives. It also supports deletions of the inserted items. However, both QF and CQF's hash tables should have the

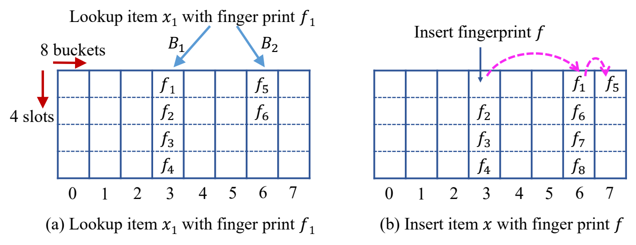

Lookup item x_1 with finger print f_1
8 buckets
4 slots
B_1
B_2
f_1 f_2 f_3 f_4 f_5 f_6
0 1 2 3 4 5 6 7
(a) Lookup item x_1 with finger print f_1
Insert fingerprint f
f_1 f_5 f_2 f_6 f_3 f_7 f_4 f_8
0 1 2 3 4 5 6 7
(b) Insert item x with finger print f

Figure 2: Example of a cuckoo filter

number of entries to be a power of two, which hurts the memory efficiency.

Cuckoo Filters. The recently proposed cuckoo filters (CFs) [23] improves Bloom filters in two aspects. First, in ideal cases, cuckoo filters cost smaller memory than the space-optimized Bloom filter when the target false positive rate $\epsilon < 3\%$. Second, cuckoo filters support deletion operations without extra memory overhead. A cuckoo filter is a table of m *buckets*, each of which contains 4 *slots*. Every slot stores an l-bit *fingerprint* f_x of an item x. For every item x, the cuckoo filter stores its fingerprint f_x in one of two candidate buckets with indices $B_1(x)$ and $B_2(x)$:

$$B_1(x) = H(x) \bmod m$$
$$B_2(x) = \texttt{Alt}(B_1(x), f_x)$$

where H is a uniform hash function and function $\text{Alt}(B, f) = B \oplus H'(f)$, where H' is another uniform hash function. It is easy to prove: $B_1(x) = \texttt{Alt}(B_2(x), f_x)$ which means, using f_x and one of the two bucket indices $B_1(x)$ and $B_2(x)$, we are able to compute the other index. To lookup an item x, we check whether the fingerprint f_x is stored in two buckets $B_1(x)$ and $B_2(x)$ as shown in Fig. 2(a).

Cuckoo filter insertion. For each item x, the cuckoo filter stores its fingerprint f_x in an empty slot in Bucket $B_1(x)$ or $B_2(x)$ if there is an empty slot, as in Fig. 2(a). If neither $B_1(x)$ nor $B_2(x)$ has an empty slot, the cuckoo filter performs the **Eviction** process. It randomly chooses a non-empty slot in bucket B (B is one of $B_1(x)$ and $B_2(x)$). The fingerprint f' stored in the slot will be removed and replaced by f_x. Then f' will be placed to a slot of the alternate bucket $\texttt{Alt}(B, f')$ of f', as shown in Fig. 2(b). If the alternate bucket is also full, the cuckoo filter recursively evicts an existing fingerprint f'' in Bucket $\texttt{Alt}(B, f')$ to place f', and looks for an alternate slot for f''. When the number of recursive evictions reaches a threshold, this insertion is failed and a reconstruction of the whole filter is required.

Though there have been continuous studies of the variants of cuckoo filters [16, 36, 45], we observe that **two fundamental limitations** prevent cuckoo filters from being widely used as a replacement of Bloom filters.

1. The claimed advantage of cuckoo filters in memory-efficiency can only be achieved in ideal situations, i.e., the number of items is around 3.8×2^x for an integer x. In a generalized case where the number of items could be arbitrary, cuckoo filters may need as much as $\sim 50\%$ extra memory in the worst case and $\sim 25\%$ extra memory in average.
2. To support deletions, cuckoo filters will store duplicate fingerprints and may crash due to table overflow when inserting duplicate items. In practical applications, inserting duplicate items are ubiquitous and cannot be detected by cuckoo filters.
3. Cuckoo filters do not support incremental expansion of the item set, a requirement by many applications.

Dynamic cuckoo filters. A dynamic cuckoo filter (DCF) [19] uses a number of linked homogeneous CFs, can support extension of the key set. Since a lookup needs to check all linked CFs, a DCF has lower throughput and higher false positive rate compared to a CF.

Morton filters. Morton Filters (MFs) [16] are variants of CFs. The main design goal of MF is to provides higher throughput for special hierarchical memory systems. MFs introduce virtual buckets and divide logical buckets into memory-aligned blocks. To support high throughput, MF contains extra bits - overflow flags and bucket counters in every block. MFs are claimed faster than CFs on the ARM architecture. MFs only support certain lengths of fingerprints (hence certain false positive rates), which significantly restricts its application range. Besides, Morton Filters cannot use the semi-sorting optimization in CFs [23].

Other variants of cuckoo filter. The adaptive cuckoo filter [36] reduces the false positive rate by maintain a cuckoo hash table in a slow memory. It changes a stored fingerprint when a false positive is detected. The D-ary cuckoo filter [45] aims to provide higher space utilization by increasing the number of candidate buckets for each key. However, it increases the time cost of insertions and lookups.

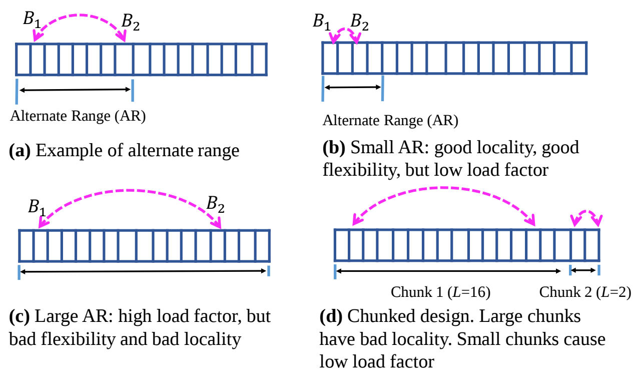
3. DESIGN OF VACUUM FILTERS

3.1 Problem statement

A vacuum filter is an AMQ data structure for a set of items, which supports **insertion**, **lookup**, and **deletion** operations. The construction of a vacuum filter can be implemented as serial insertions of all items in the given set. When executing the lookup operation for a queried item x, the vacuum filter should return either `positive`, indicating that x is in the set, or `negative`, indicating that x is not in the set. Similar to most other AMQ data structures [13, 12, 23], a vacuum filter may report false positive results, but never report false negative results. A vacuum filter utilizes a table-based structure to store fingerprints, similar to those used in quotient filters [12], cuckoo filters [23], and Morton filters [16]. Each fingerprint is a brief representation of the key of an item. If the fingerprint of a queried item is found in certain buckets of the table, the AMQ structure returns `positive`. Compared to cuckoo filters, the typical table-based AMQ structures, vacuum filters have the following main advantages by resolving several challenges that are not addressed in prior methods. 1) Vacuum filters are more space-efficient than cuckoo, Bloom, quotient, and Morton filters. Compared to cuckoo filters, vacuum filters reduce the space cost by $> 25\%$ on average. 2) Vacuum filters provide higher lookup and insertion throughput, due to better data locality. 3) Vacuum filters can replace cuckoo filters in most applications.

3.2 Where to gain extra space-efficiency and throughput

Both vacuum and cuckoo filters use the table structure: A table has m buckets and each bucket has 4 slots to store fingerprints, as shown in Fig. 2. When a new fingerprint is inserted and its both buckets are full, there should be a



(a) Example of alternate range

(b) Small AR: good locality, good flexibility, but low load factor

(c) Large AR: high load factor, but bad flexibility and bad locality

(d) Chunked design. Large chunks have bad locality. Small chunks cause low load factor

Figure 3: Selecting a proper AR is non-trivial

way to evict an existing fingerprint to its alternate bucket by using the function Alt(). If the alternate bucket is again full, another fingerprint in the alternate bucket will be recursively evicted. The most important feature of Alt() is to ensure that $B_2 = \texttt{Alt}(B_1, f) \bmod m$ and $B_1 = \texttt{Alt}(B_2, f) \bmod m$, for a fingerprint f and its alternate buckets B_1 and B_2. In addition, compared to the function Alt() used in cuckoo, we want to achieve **two desired properties** of Alt() in the new design.

1. The fingerprints should be spread evenly over all buckets by Alt() to achieve high table *load factor* without any restriction on the number of buckets. The load factor is the proportion of filled slots in a table – an important metric for memory efficiency.

2. The two alternate buckets should be stored with a certain level of *locality*: they could co-locate in a same cacheline, page, or big page to reduce the memory access cost.

Note cuckoo filters use $\texttt{Alt}(B, f) = B \oplus H'(f)$, which relies on the assumption that m must be a power of two. It may cause significant space waste. In a case when we need 1025 buckets, we have to use $m = 2048$, almost 50% space waste. Simply replacing $\oplus$ with $+$ does not resolve this problem. It can work with an arbitrary m but insertions may easily fail with a low load factor of the table when m is not a power of two. Also there is very little locality using this Alt() function. Two buckets can be arbitrarily far away across the table. Hence each insertion operation may access the memory in different cachelines and pages, resulting in many cacheline and translation lookaside buffer (TLB) missings. Each lookup or deletion operation needs to access two buckets in different cachelines and pages. Assuming the structure runs in the main memory, these cache missings are considered the bottleneck operations of an AMQ structure. We run a set of experiments with $m = 2^{28}$ and every bucket occupies 8 bytes. The results show the two alternate buckets of cuckoo filters never fit into the same 64-byte cache line or the same 4K-byte page.

The alternate function of vacuum filters achieves these two goals. We firstly model the problem as a "Balls into Bins" problem [42] and introduce the concept of *alternate ranges* to balance the tradeoff between the load factor and data locality. Based on these results, we propose an optimized *multi-range alternate function* that achieves both good data locality and high load factor.

3.3 Alternate Ranges

To maintain a certain level of locality, we may divide the whole table into multiple equal-size chunks, each of which includes L consecutive buckets and L is a power of two. Hence m is a multiple of L, instead of being restricted to a power of two as in cuckoo filters. The two candidate buckets of each item should be in the same chunk. For each item x, we compute the indices of the two alternate buckets using $B_1(x) = H(x) \bmod m$ and:

$$B_2(x) = \texttt{Alt}(B_1(x), f) = B_1(x) \oplus (H'(f) \bmod L) \qquad (1)$$

Note we change the alternate function slightly. It is easy to prove that $\lfloor B_1(x)/L \rfloor = \lfloor B_2(x)/L \rfloor$, which means the two buckets fall into the same chunk. This method is denoted as chunked-CF. Hence by knowing the fingerprint f and one of the alternate buckets, we can always compute the other alternate bucket using Equation 1, because we also have $B_1(x) = \texttt{Alt}(B_2(x), f)$. For a better illustration, we call the length of chunk, L, as the *alternate range* (AR), shown in Fig. 3(a).

Determining a proper size L of the alternate range is challenging. If the AR is small, as shown in Fig. 3(b), the filter provides good data locality since the two alternate buckets are very close to each other and likely to be in the same cache line or page (called *co-located*). However, a small AR can cause *fingerprint gathering*. Fingerprint gathering means all alternate buckets of many fingerprints are in a small range of buckets. The search space of the eviction process is limited, hence eviction loops are likely to happen and the insertions can easily fail. A large AR (Fig. 3(c)) can avoid fingerprint gathering and provide high load factor, but its locality becomes bad. Besides, the flexibility of the table is limited, because the number of buckets should a multiple of L – extra buckets may be used and the space cost increases. We show the experimental results of the load factor in Fig. 4, the rate of two alternate buckets in a same cacheline in Fig. 5, and the rate of two alternate buckets in a same page in Fig. 6, by varying the AR size L. The dilemma is obvious from the results: small ARs cause low load factors and large ARs cause bad locality.

Algorithm 1: LoadFactorTest(n, α, r, L)

```
$m = \lceil n/4\alpha L \rceil L$ // the number of buckets;
$N = 4rm\alpha$ // the number of inserted items ;
$c = m/L$ // the number of chunks;
$P = 0.97 \times 4L$ // the capacity lower bound of each chunk;
D = EstimatedMaxLoad$(N, c)$ ;
if $D < P$ then
    return Pass;
else
    return Fail;
end
```

An existing work of chunked hash table [34] fixes the number of chunks to 256 empirically. We improve this design by calculating the minimum AR size based on the number of items n and the target load factor α. The first algorithm is to test whether a specific alternate range, L, can achieve the target load factor α given the number of items n, as shown in Algorithm 1. r is a parameter that shows the ratio of inserted items in the total number of items. Given n, the target load factor α, and the number of slots per bucket b (4 in our case), we can calculate the number of buckets m. The whole table of buckets is separated into $c = m/L$ chunks.

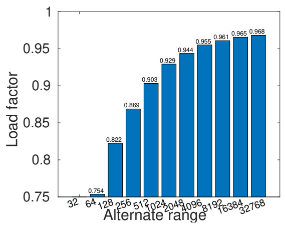


Figure 4: Load factor vs. ARs

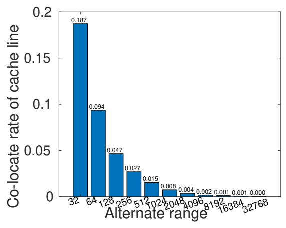


Figure 5: Rate of two alternate buckets in a same cacheline

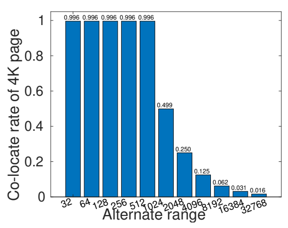


Figure 6: Rate of two alternate buckets in a same page

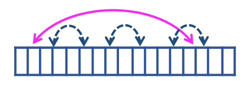

Keys have independent ARs. Most keys use small ARs. A small fraction use large ARs. Achieves high load factor, good locality, and flexibility

Figure 7: Vacuum design for alternate ranges

The items are randomly distributed to the chunks. We can model this problem as the "Balls into Bins" problem [42]. The goal is that all these chunks will not be overwhelmed by the inserted items. The "Balls into Bins" model provides an estimation of the maximum load and we use a loose estimation from [42] to calculate the maximum load with high probability:

$$\texttt{EstimatedMaxLoad}(n, c) = \frac{n}{c} + \frac{3}{2}\sqrt{2\frac{n}{c} \cdot \log(c)} \qquad (2)$$

If the estimated maximum load is smaller than the capacity of each chunk P, we consider this L is good to use and the algorithm returns 'Pass'. Otherwise it returns 'Fail'.

Then the second algorithm selects the minimum AR size that can pass the load factor test to achieve good locality, shown in Algorithm 2. For example, when $m = 2^{25}$ and expected load factor $\alpha = 0.95$, the algorithm will select $L = 32768$ as the AR size L.

Algorithm 2: `RangeSelection`(n, α, r)

```
$L = 1$ ;
while LoadFactorTest$(n, \alpha, r, L) \neq Pass$ do
    $L \leftarrow L \times 2$
end
return $L$
```

Table 2: Determine the # of AR to balance the load factor and locality. 100 random tests for 2^{27} items. One failed insertion means the final load factor is lower than 95% in a run.

Metric \ # of ARs	1	2	4	8
Average Load Factor	96.6%	96.3%	96.1%	95.3%
Insertion Fail Rate ($\alpha = 95\%$)	0%	0%	1%	7%
Co-locate in a 64B cacheline	0.00%	4.69%	7.03%	9.86%
Co-locate in a 4KB page	0.83%	50.61%	75.00%	81.03%
Co-locate in a 4MB page	95.32%	98.44%	99.22%	99.61%

3.4 Multi-Range Alternate Function

We shall not stop with a fixed AR size as we showed that small ARs cause low load factors and large ARs cause bad locality. To achieve the best of both worlds, we propose a multi-range alternate function. Our idea is inspired by the road network, which usually consists of a large portion of short local roads and a small portion of long highways. We allow every item to have an *independent alternate range*. Also, most items have small alternate ranges to achieve locality and a small number of items have large alternate ranges to avoid low load factor, as shown in Fig. 7.

For example, given the target load factor $\alpha = 95\%$, we may allow 25% items to use a large AR and 75% items to use a much smaller AR. Denote the large AR as L_0 and small AR as L_1. The AR size calculated by Algorithm 2 can be considered as the upper bound of L_0 to achieve α. Hence we set $L_0 = \texttt{RangeSelection}(n, \alpha, 1)$. Next, we consider the small AR L_1. Note that a large chunk with size L_0 and a small chunk with size L_1 may overlap as shown in Fig. 7. Hence the items with AR L_0 may exist in a small chunk but will not always take the space of the small chunk – it can be evicted to a bucket outside the small chunk. We can re-use the range selection algorithm for these 75% of items of AR L_1 to compute $L_1 = \texttt{RangeSelection}(n, \alpha, 0.75)$.

Moreover, we can use more than two ARs. For example, if we want to have K different ARs, we can set those ARs by the calculation method above. Denote those ARs as L_0 to L_{K-1}. Let $L_0 \geq L_1 \geq \ldots L_{K-1}$. For the i–th AR, $\alpha * (1 - i/K)$ of items should have their ARs smaller than or equal to L_i. So we set $L_i = \texttt{RangeSelection}(n, \alpha, 1 - i/K)$.

How many ARs should we use eventually? From the implementation perspective, we set the number of ARs as a power of two, so we can assign ARs to the items by their least significant bits. We test different numbers of ARs to find the best configuration and show the results in Table 2. The result shows that when we use one AR, it cannot achieve good data locality (co-locate rates). With more ARs, the fail cases happen more frequently. To balance both the load factor and locality, we use 4 different ARs in the final design of vacuum filters.

The final design of the alternate function `Alt`() is presented in Algorithm 3. All items will be divided into four (roughly) equal-size groups and each group uses a AR size determined by the calculation of `RangeSelection`().We double the smallest alternate range $L[3]$ to avoid fingerprint gathering that will cause insertion failures. Due to the loose maximum load estimation, it causes relatively bad performance when there is a small number of keys. Thus, we design another alternate function that is shown as Algorithm 4 when the number of keys is smaller than 2^{18}.

3.5 Optimization for insertion

The insertion algorithm is slightly different from the recursive eviction process introduced in cuckoo filters [23]. To further optimize space utilization and throughput, we introduce optimization in the insertion process.

The proposed recursive eviction is different from that in cuckoo hashing [39]. The hash table can be viewed as an

Algorithm 3: Vacuum Filter : `Alt`(B, f)

```
if $L$ is not initialized then
    for $i = 0$; $i < 4$; $i{+}{+}$ do
        $L[i] = \texttt{RangeSelection}(n, 0.95, 1 - i/4)$
    end
    $L[3] = L[3] \times 2$ // Enlarge $L[3]$ to avoid failures ;
end
$l = L[f \bmod 4]$ // Current alternate range ;
$\Delta = H'(f) \bmod l$ ;
return $B \oplus \Delta$ ;
```

Algorithm 4: Vacuum Filter : `Alt`(B, f)

```
$\Delta' = H'(f) \bmod m$;
$B' = (B - \Delta') \bmod m$ ;
$B' = (m - 1 - B' + \Delta') \bmod m$;
return $B'$;
```

undirected graph, where each bucket is a vertex and each item can be considered as an edge that connects the two alternate buckets of the item. Cuckoo hashing uses a "random eviction" scheme to look for an empty slot, which can be considered as a random depth-first search (DFS) of the graph. The random DFS scheme is easy to implement. However, as table occupancy grows, the random DFS scheme needs to examine more buckets and thus require more eviction steps to find an empty slot. A breadth-first-search (BFS) scheme for cuckoo hashing is proposed in [32]. For a full bucket of fingerprints, evicting each fingerprint will start a possible eviction path. The BFS scheme has a broader searching space and thus may reduce the number of evictions to find an empty slot. However, it needs to maintain an extra queue to find the path. In vacuum filters, we combine the advantages of both BFS and DFS. When searching for an empty slot, we look ahead one step. Specifically, when we have to evict an existing fingerprint from the two full buckets of the inserting item, we traverse all alternate buckets of the 8 fingerprints. If there is an empty slot among these buckets, then we can evict the corresponding fingerprint to the empty slot and finish the eviction process. This optimization increases the success rate of insertion and hence improve both the load factor and insertion throughput. In addition we do not need to maintain an extra queue. Algorithm 5 shows the vacuum filter insertion algorithm. Besides, semi-sorting is a technique to reduce the space cost of storing fingerprints, introduced in an early work [14]. Vacuum filters provide an option of using semi-sorting, similar to cuckoo filters [23].

3.6 Lookup and deletion

To lookup an item x, it first computes two candidate buckets $B_1(x) = H(x)$ and $B_2(x) = \texttt{Alt}(B_1(x), f)$. If the fingerprint f_x matches one fingerprint stored in the two buckets, the algorithm returns `positive`. Otherwise it returns `negative`. When we generate the first candidate bucket for an item, we need a modulo operation to map the 32-bit hash values of the bucket indices to $[0, m-1]$. In cuckoo filters, when m is a power of two, the modulo can be replaced by a simple bit-wise `AND` operation to increase the speed of calculation – and hence improve the lookup and insertion throughput. However, when m is not a power of two as in vacuum filters, the bit-wise `AND` operation fails, which forces us to use another method.

Algorithm 5: Vacuum Filter : Insert(x)

```
$f = H'(x)$ // $H'$ is the fingerprint function;
$B_1 = H(x)$, $B_2 = \texttt{Alt}(B_1, f)$;
if $B_1$ or $B_2$ has an empty slot then
    put $f$ into the empty slot ;
    return Success;
end
Randomly select a bucket $B$ from $B_1$ and $B_2$ ;
for $i = 0$; $i <$ MaxEvicts; $i$++ do
    // Extend Search Scope
    foreach fingerprint $f'$ in $B$ do
        if Bucket $\texttt{Alt}(B, f')$ has an empty slot then
            put $f$ to the original slot of $f'$ ;
            put $f'$ to the empty slot ;
            return Success ;
        end
    end
    Randomly select a slot $s$ from bucket $B$ ;
    Swap $f$ and the fingerprint stored in the slot $s$ ;
    $B = \texttt{Alt}(B, f)$ ;
end
return Fail ;
```

What we need is a map function, which maps 32-bit hash value to $[0, m-1]$ uniformly to avoid the collision. We adopt the method from [1]: map$(x, m) = (x \cdot m) \gg 32$. If x is uniformly distributed on $[0, 2^{32}-1]$, then the first multiplication scale it to a distribution over $[0, (2^{32} - 1) * m]$. Then the right shift operation compress the interval to $[0, m-1]$. This method generates a roughly uniform distribution with only two light instruction, which is comparably fast to the `AND` operation.

The deletion algorithm for item x is simple. If there is a fingerprint equal to f_x in the two candidate buckets of x, the vacuum filter removes this fingerprint.

4. ANALYSIS RESULTS

We present the analysis results on the load factor, false positive rate, space cost and time cost.

4.1 Load factor

We give the theoretical analysis for static ARs. The analysis for the multi-range alternate function is more complicated and we will leave it as future work. In the static range case, items will be randomly distributed into different chunks. Each chunk can be viewed as a single sub-table and all evictions of an insertion happen in a single chunk. Therefore, we need two steps of analysis - we first calculate the expected load factor of each chunk, and then check whether the load factor of each chunk can satisfy a certain value of overall load factor of the entire vacuum filter.

Upper bound of the number of items in each chunk. Let the AR be L and the number of buckets is m. There are $c = m/L$ chunks and n items to insert. We utilize the results from the well-studied "Balls into Bins" model [42], in which n balls are uniformly randomly distributed to c bins. In [42], the author gives an upper bound about the number of balls in any bin.

THEOREM 1. *Let M be the random variable that counts the maximum number of balls in any bin. Then* $\texttt{Pr}[M > k_a] = o(1)$, *where $a > 1$ and $k_a = \frac{n}{c} + a\sqrt{2\frac{n}{c}\log c}$, if $n \gg c\log c$.*

In our case, n and c satisfy $n \gg c \log c$. Let $a = \frac{3}{2}$. We have the maximum number of items in any chunk will be smaller than $\frac{n}{c} + \frac{3}{2}\sqrt{2\frac{n}{c}\log c}$ with high probability.

Expected load factor of vacuum filters. In [22], the author provides the analysis of the load factor of cuckoo filters. However, it requires the number of slots per bucket to be 10.916, which does not fit our design. **To our knowledge, there is no theoretical result of the load factor that fits the four-slot table design.**

Hence we use experiments to test the maximum load factor for those chunks. The experiment shows that the load factor of a single chunk can achieve 97% with probability greater than 99%. Hence in the `RangeSelection` algorithm, we fix the overall load factor as 95% and select the AR value such that the upper bound of the number of items does not exceed 97% of the chunk capacity, which ensures that even if the number of items of a single chunk reaches the upper bound, the insertions will not fail with $> 99\%$ probability. As a result, the final design of vacuum filters can achieve high load factor (95%) with $> 99\%$ probability.

4.2 False positive rate and space cost

Two factors influence the false positive rate of a vacuum filter: 1) the length of fingerprint l; and 2) the number of slots b in each bucket (usually set to 4). We call an item 'alien item', if it is not in the target item set. In a vacuum filter, the probability that a query of an alien item matches one stored fingerprint (a false-positive match) is at most $1/2^l$. The probability of false positive rate can be computed after $2b$ comparisons. We also need to consider the load factor α of the table of the vacuum filter. Then the expected number of comparisons is $2b\alpha$. The probability of no false hit is $(1-1/2^l)^{2b\alpha}$. Thus, the upper bound of the total probability of false positive rate is

$$\epsilon = 1-(1-1/2^l)^{2b\alpha} \approx 2b\alpha/2^l \tag{3}$$

We can derive the necessary fingerprint length for a given target false positive ϵ:

$$l \geq \lceil \log_2(2b\alpha/\epsilon) \rceil \tag{4}$$

For a given number of items n, the whole memory consumption M_V is

$$M_V = (n/\alpha)\lceil \log_2(2b\alpha/\epsilon) \rceil \tag{5}$$

where the unit is bit.

The theoretical number of bits per item for vacuum filters (VFs) is $\frac{\log_2(2b\alpha)+\log_2(1/\epsilon)}{\alpha}$. Since the semi-sorting technology use one bit less per item, the space cost per item for vacuum filters with semi-sorting (VFs-ss) is $\frac{\log_2(2b\alpha)-1+\log_2(1/\epsilon)}{\alpha}$.

Given $\alpha = 0.95$ and $b = 4$, the space cost per item for VFs and VFs-ss are $3.07 + 1.05\log_2(1/\epsilon)$ and $2.07 + 1.05\log_2(1/\epsilon)$. Cuckoo filters (CFs) have the same space cost in their best cases, and their average and worst cases need 25% and 50% more space respectively. For counting quotient filters (CQFs) [4], at the same load factor, the space cost per item is $2.24 + 1.05\log_2(1/\epsilon)$. Similar to CFs, CQFs require the number of buckets to be a power of two. Hence the space cost in the average and worse cases is higher than this value. For Bloom filters, the space cost per item is $1.44\log_2(1/\epsilon)$. When $\epsilon < 3\%$, VFs have the lowest space cost compared to other AMQ structures.

4.3 Time cost

The time cost for each lookup or deletion of VFs is constant – either of them only needs at most 2 memory accesses.

The analysis of the time cost for each insertion is more complicated. Since all fingerprints are uniformly distributed among the buckets, the probability that a bucket of b slots is full is α^b. Assume the searching process in the insertion algorithm follows the Bernoulli distribution with a successful rate $1-\alpha^b$. Let p be the number of traversed buckets of an insertion. Then the expectation of p under the load factor α, $E(p_\alpha)$ can be estimated by the equation

$$E(p_\alpha) = (1-\alpha^b) + \alpha^b(E(p_\alpha)+1) \tag{6}$$

We have $E(p_\alpha) = 1/(1-\alpha^b)$. For the average insertion cost E for serial insertions from load factor 0 to α, we can integrate $E(p_x)$ from 0 to α,

$$E = \int_0^\alpha E(p_x)dx = \int_0^\alpha 1/(1-x^b)dx \tag{7}$$

With $b = 4$ and $\alpha = 0.95$, we have $E = 1.3$. Our experimental results show that the average traversed number of buckets is about 1.58, which is close to the theoretical result.

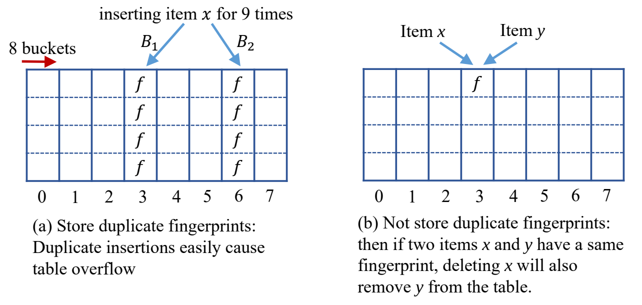

inserting item x for 9 times
8 buckets
B_1
B_2
f
0 1 2 3 4 5 6 7
(a) Store duplicate fingerprints: Duplicate insertions easily cause table overflow
Item x
Item y
(b) Not store duplicate fingerprints: then if two items x and y have a same fingerprint, deleting x will also remove y from the table.

Figure 8: Duplicate insertions are difficult to handle

5. VACUUM FILTERS UNDER DYNAMICS

5.1 Problem statement

Dynamics of the item set include item insertions, deletions, and set resizing. Note if the item insertion rate is approximately equal to the deletion rate, set resizing will not happen. However, if the insertion and deletion rates are not equal in some applications, the set size may change after a period of time. For example, by keeping inserting items to the set, the set may become too larger to fit the current table at a certain time. Bloom filters can deal with insertions but not deletions. Cuckoo filters can handle insertions and deletions when there is no *duplicate insertion*. Duplicate insertions mean a same item may be inserted to the data structure for multiple times. If set resizing or duplicate insertions happen, cuckoo filters may fail or at least become sub-optimal.

Duplicate insertions exist in many practical applications. For example, a proxy server in a content distribution network may use an AMQ structure to represent the current set of cached content in the local network [33, 17]. When a proxy server caches a content, it notifies all other servers about the cached content. Obviously, a same content may be cached in different servers by multiple times, hence there will be duplicate insertions. We show the dilemma in dealing with duplicate insertions in Fig. 8. As shown in Fig. 8(a), if we allow to store duplicate fingerprints, then after 9 times of inserting the same item x, the table will overflow and fail. As shown in Fig. 8(b), if we do not allow to store duplicate fingerprints, then two items x and y may have a same fingerprint. Deleting x will also remove y from the table, resulting in future false negatives!

In addition to duplicate insertions, neither cuckoo filters nor counting Bloom filters can deal with set resizing. In fact, no existing AMQ structures work well under duplicate insertions or set resizing.

5.2 Instant updates and periodical reconstructions

We plan to solve these problems in a practical model. Most applications make use of a memory hierarchy to construct the AMQ data structures, in which the full data set is stored in a slow and large memory and the AMQ is running in a fast and small memory. For example, the slow memory can be disks and the fast memory can be main memory [13]; the slow memory can be a server and the fast memory can be network devices that use ASICs [17, 48, 49]. The AMQ structure (vacuum filter in our case) is used to response to membership lookups that should be processed in a fast speed. In addition, the vacuum filter should *instantly* update itself when there is any insertion or deletion. However, after a period of time and a number of updates, the vacuum filter may be sub-optimal, e.g., its size does not fit the current item size. At this time, a new version of vacuum filter is constructed by another process (or another machine) using the slow memory. The old vacuum filter running in the fast memory will be replaced with the new version. This method is called IUPR (Instant Updates and Periodical Reconstructions). For every time of reconstruction, duplicate insertions and set resizing are resolved. Hence the vacuum filter can be recovered from the sub-optimal condition.

5.3 Instant updates of vacuum filters

One requirement for the self updates on the vacuum is a fast speed to avoid interruption of responding AMQ queries. Our goal is to provide fast updates and the update operations include insertions and deletions. For deletion operations, the requirement is to delete the items which must have been previously inserted [23]. Insertion operations are more complicated to deal with compared with deletion operations. It is because a full vacuum filter needs to be extended to insert more items.

To achieve this goal, we design the Dynamic Vacuum Filter (DVF) inspired by the Dynamic cuckoo filter (DCF) [19] for fast self updates of IUPR. A DCF uses a linked chain of cuckoo filters for some extended space.

However, directly applying DCFs to vacuum filters faces some problems. 1) A DCF is based on the standard cuckoo filter, the load factor of each cuckoo filter is low when inserting a certain number of items, which is not memory-efficient. 2) The design of a DCF incurs high cost for the lookup process. Assuming there are s chained cuckoo filters, the lookup process needs $2s$ memory accesses, which increases the cost when s grows big. 3) The false positive rate increases to $2bs/2^l$ compared to $2b/2^l$ of the original cuckoo filter. 4) A DCF requires that every linked cuckoo filter has the same size of buckets. Thus the number of buckets in each cuckoo filter is decided by the first cuckoo filter. If the initial number of buckets is small, this design could incur many small cuckoo filters when more items come, significantly increasing the cost for the operations. However, in many applications it is hard to pre-determine a proper number of buckets at the system initialization. Therefore DCFs cannot be directly used for vacuum filters.

A DVF uses a linked chain of vacuum filters. A vacuum filter achieves about 95% memory utilization rate without any constraint on the number of inserting items. Thus, the DVF can resolve the first problem of DCFs. Another feature of the DVF is that we do not require all the linked vacuum filters have the same number of buckets, which makes the DVF more flexible in dynamic environments. The number of buckets in each vacuum filter is independently decided.

Insertions. The status of a vacuum filter may be full or not when there are new items to be inserted. Two conditions are used to decide whether the filter is full. 1) The load factor of the filter reaches a defined threshold, e.g., 96%; 2) The current insertion fails. If the filter is not considered full, we can easily insert the item. When the filter is considered full, we create another table and order all tables in a sequence with ascending order of the time of creation. Then the future items will be inserted into the newly created table, called the tail table. If there are multiple tables in the list, during item insertions, we check the status of the tail table instead of traversing all tables in the list to save time. Obviously the lookup and insertion performance will downgrade by chaining more tables. However, IUPR includes the periodical reconstruction phase, which will recover the chained tables to a unified vacuum filter table including the fingerprints of all current items. The process benefits from the property of a vacuum filter that it can be in any size without restricting the value of m.

Lookups. To lookup of an item x, the vacuum filter needs to traverse the tables in the chain. If no table has a matched fingerprint, the vacuum filter reports `negative`.

Deletions. The deletion of an item x is simple. The DVF performs a membership query of x in the whole vacuum filter list. If a corresponding fingerprint of x is found, we delete the fingerprint. One situation caused by deletion operations is that some vacuum filters in the list may get low load factors after deleting a number of items. The sub-optimal tables will be resolved by each reconstruction.

During each update operation, the system cannot support the membership query. Since each of these updates can be finished with in $O(1)$ time in average similar to those in cuckoo hashing [39]. Thus lookup operations will not be significantly interrupted. On the other hand, a reconstruction to create a new version of the vacuum filter may take a longer time. Hence we propose to let a different machine or process to run the reconstruction program *concurrently* with the lookup process.

5.4 Periodical reconstruction

To resolve the performance downgrading problem caused by fast updates, we introduce the periodical reconstruction process that creates new versions of the vacuum filter.

Each reconstruction can be triggered in two ways. First, after a time interval T, the construction process reconstructs the vacuum filter from the up-to-date data set. Second, it can be triggered by special events, e.g., the lookup throughput of the vacuum filter is lower than a pre-defined threshold, or the number of updates exceeds a pre-defined threshold. During the reconstruction process, the current vacuum filter running in the fast memory still accepts membership queries and performs fast updates. When a new version of the vacuum filter has been constructed, the construction

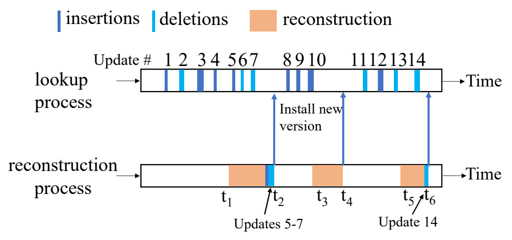


Figure 9: Handling concurrency in IUPR

process will first apply the update operations that happened during the reconstruction time. It is because these operations are not included in the past reconstruction. Then the construction process transmits the new version to the lookup process, and the lookup process replaces the old filter with the new version. Fig. 9 is an example of such parallel reconstruction. It shows the timelines of both lookup and construction processes. At time t_1, a reconstruction is triggered and the new vaccum filter should reflect the original item set and the recent updates #1-4. When this reconstruction finishes, there are three more updates happened during this period, namely updates #5-7. Hence the construction process applies instant updates to the new version for 5-7 and sends the new version to the lookup process. The lookup process has self-updated the updates #1-7 and may experience a downgraded performance. It replaces the old vacuum filter with the new version and is recovered to a better performance. Under these dynamics *the downgraded performance of the vacuum filter can always be recovered* after a period of time. The experimental results is shown in Section 6.3.

In addition, reconstructions help to keep a very small number of chained tables in the vacuum filter. In DCF that the chain could be arbitrarily long. A long chain of table will significantly decrease the lookup and update throughput. Two parameters are taken into consideration to achieve a good tradeoff for the dynamic design. The first one is the time interval of triggering the reconstruction process. The second one is the number of buckets for the newly chained table during fast updates. The reconstruction time interval is suggested to be determined in a per-application basis. If we decide the reconstruction time interval T and predict the insertion rate as r_i and deletion rate as r_d, we set the size of the newly chained table to contain $2T(r_i - r_d)$ items, using 2x as a safety margin.

5.5 Summary of IUPR

The design of IUPR solves the performance downgrading problem caused by duplicate insertions and set resizing. These problems frequently happen in practical applications but has not been well-addressed by existing methods. IUPR is an extra component to further strengthen vacuum filters that work on practical memory hierarchies. *Note even if IUPR is not used, vacuum filters provide better performance than Bloom or cuckoo filters in most concerned metrics including the space cost, throughput, and false positive rate.*

6. PERFORMANCE EVALUATION

6.1 Evaluation methodology

We implement a complete software prototype of vacuum filters including the basic algorithms and the protocols in IUPR. There are two design options of vacuum filters in implementation. One option is that for different lengths of fingerprints, whether we should add padding bits so that no bucket will cross two cachelines. The other option is whether the implementation includes semi-sorting. The implementation of vacuum filters including semi-sorting but no padding is denoted as **VF-ss (No Padding)**. The implementation including padding but no semi-sorting is denoted as **VF**. The implementation including both padding and semi-sorting is denoted as **VF-ss**. We use the implementation of cuckoo filters provided by their authors [3]. The corresponding versions of cuckoo filter are denoted as **CF** and **CF-ss**. For Morton filter (MF), we use the implementation provided by the authors [7]. The implementation of Bloom filters is standardized, we set the number of hash functions to $\lfloor \ln 2(m/n) \rfloor$ in order to achieve the lowest false positive rate [17], where n is the number of items, m is the array size. We also implement the deletable Bloom filters (DIBFs) [43] and blocked Bloom filters (BBFs) [41]. We use open source code for Counting Quotient Filters (CQFs) [4]. The code for Vacuum Filters is available in [8].

Unless otherwise mentioned, the items used for experiments are pre-generated 64-bit distinct integers from random number generators. All experiments are running on a DELL work station with Intel E5-2687W CPU, 3.00 GHz, and 30 MB L3 Cache. The hard-disk is SK-hynix-SC311-S 1TB SSD.

Metrics: We evaluate the following metrics:

- *False positive rate*: measured by querying a filter with non-existing (alien) items and then calculating the fraction of returned positive results. The false positive rate is usually a target value that needs to be achieve by adjusting other parameters.
- *Bits per item*: this metric reflects the memory cost. We count the average number of bits used per item in an AMQ structure to achieve a target false positive rate.
- *Load factor*: measured by the number of bits used to store fingerprints, over the total size of the data structure.
- *Lookup, insertion, and deletion throughput*: measured by the number of lookup/insertion/deletion operations a data structure can process per second.

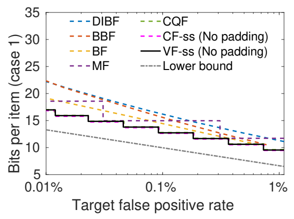
Unless otherwise mentioned, for every result shown in this section, we conduct 10 production runs and compute the average.

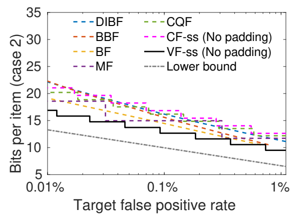
For the evaluation under dynamic environments, we compare the performance between IUPR-VF method and a modified dynamic vacuum filter (linked-VF) method without reconstruction, which will be illustrated in Section 6.3. We also compare IUPR-VF with IUPR-CF.

In addition, we evaluate the gain of replacing Bloom and cuckoo filters with vacuum filters in a real application: checking the revocation status of digital certificates.

6.2 Evaluation of data structures

We compare the performance of vacuum filters (VFs), cuckoo filters (CFs), Bloom filters (BFs), Blocked Bloom filters (BBFs), Deletable Bloom filters (DIBFs), Morton filters (MFs) and Counting Quotient Filter (CQFs). For VFs and CFs, we also compare the performance of their semi-sorting implementation (VF-ss and CF-ss). Note counting


(a) Case 1


(b) Case 2

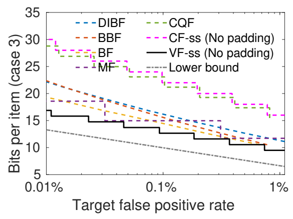

(c) Case 3

Figure 10: Memory cost versus false positives

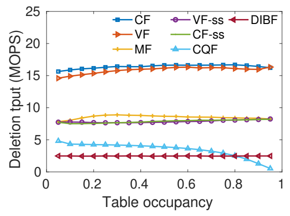
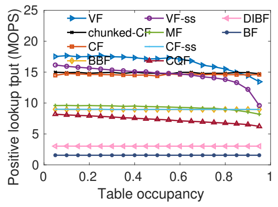


Figure 11: Deletion Throughput.


(a) Lookups for existing items

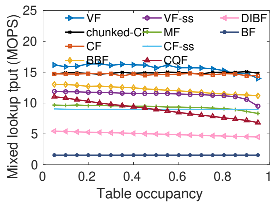
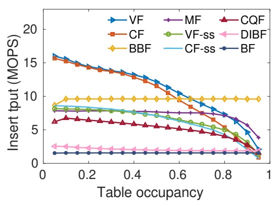

(b) Lookups for mixed items

Figure 12: Performance of lookup throughput


(a) Varying occupancy

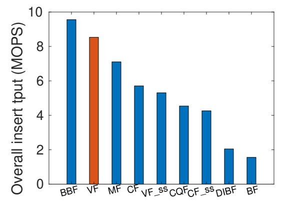

(b) Overall throughput

Figure 13: Performance of insertion throughput

Bloom filters (CBFs) have higher memory cost and lower throughput than BFs by the design. Hence we do not need to compare to CBFs.

6.2.1 *Memory cost and false positives*

We evaluate the memory usage with target false positive rates for VF-ss, CF-ss, BFs, BBFs, DIBFs, MFs and CQFs. We show the results of three cases. In Case 1 the number of items $n = 0.95 \times 2^{25}$. In Case 2, $n = 0.75 \times 2^{25}$. In Case 3, $n = 2^{25}$. Note in all three cases, the bits per item of VFs have no clear difference. However for CFs and MFs, Case 1 is considered the best case, Case 2 is considered the average case, and Case 3 is considered the worst case. The selected numbers of items and the results of three cases are the representative ones.

Fig. 10 shows the results of the memory cost (in the number of bits per item) versus the target false positive rate. We also show the theoretical lower bound in the figures for comparison, which does not exist in practice. We find that VFs achieve **the lowest memory cost in all cases**, providing 2-5 bits saving per item compared to other methods in the average case under a same false positive rate. The plots of memory cost for CFs, MFs, CQFs, and VFs are all in stair-steps, because they need to use the fingerprints whose length is an integer. When the key numbers are near powers of two (such as Case 3), CFs and CQFs have to create a much bigger hash table to store the fingerprints. As a result, the load factor of a CF will be close 50% and the memory cost is high. In the average case (Case 2), VFs still show an advantages of $\sim$5 fewer bits per item compared to CFs. BFs and the variants DIBFs and BBFs show worse memory and the false positive tradeoff than the VFs in all cases. MFs need more memory than VFs in all cases.

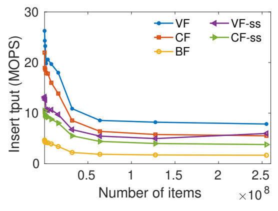
6.2.2 *Operation throughput*

In this section, we fix the fingerprint length for VFs, CFs, MFs, and CQFs to 12 bits. For VF-ss and CF-ss, we select 13 bits per fingerprints for a better alignment. The item number is $2^{27} * 0.95$, and the memory consumption are all fixed to 192MB, which is much bigger than the L3 cache. **Note this setting is a best-case situation for CFs and CQFs. We give advantages to them to show that even under these cases, VFs still outperform these methods. VFs do not rely on these situations. In other situations, VFs may provide more obvious performance gains.**

Lookup throughput. Fig. 12 shows the lookup throughput for two cases - 100% of existing items, and 50%-50% mixed of existing and alien items. Our results show that VF-ss is about 1.1x to 1.8x faster than CF-ss for positive lookups. VFs provide higher throughput than all other methods when the table occupancy rate is lower than 90%. In the implementation, VFs may access fewer than two cachelines, and CFs will always access two cachelines. Intuitively, CFs' throughput will be much slower than VFs'. However, since The Intel Xeon micro-architecture has multiple memory accessing units, the two memory accesses may be done concurrently. As a result, the lookup throughput of CFs will be slightly better than VFs when the occupancy rate is high. The number of chunks for chunked-CF is 256. We can find that chunked-CF has almost the same performance as CF. The lookup throughput of BFs and DBFs is always the lowest.

Deletion throughput. The deletion throughput is shown in Fig. 11. We can find that VFs and CFs have similar deletion throughput, higher than most other methods.

Insertion throughput. Fig. 13(a) shows the insertion throughput with different table occupancies. VFs have the highest throughput compared to other AMQ structures when the table occupancy < 65%. The insertion throughput of all methods except Bloom filters and their variants decreases with higher table occupancy. When the occupancy is higher than 60%, both CFs and VFs require more bucket accesses to find an empty slot. In these cases, the data locality can significantly affect the throughput. Since the semi-sorting optimization requires more complicated operations,


(a) Insertion

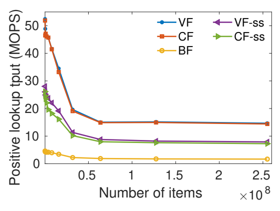
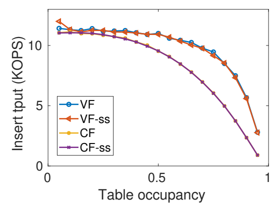

(b) Lookups of existing items

Figure 14: Throughput versus num. of items.


(a) Insertion throughput

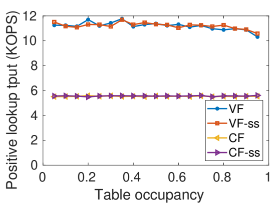

(b) Positive lookup throu

Figure 15: Performance of throughput in SSD.

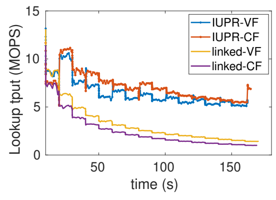
the memory access speed may not be the bottleneck. Hence, VF-ss shows slightly better throughput than CF-ss. BFs need about 9 random memory accesses, which have the slowest insertion speed among all methods. However BBFs are fast in insertions because they cause one cache miss at the most. After inserting all items into the data structures, we compute the overall performance of insertion throughput, shown in Fig. 13(b). We can find that VF achieves best overall insertion throughput except for BBF. The overall insertion throughput of VFs and VF-ss is higher than that of CFs and CF-ss.

Throughput versus the number of items. We evaluate the throughput for different numbers of items to demonstrate the scalability and stability of the methods. Figs. 14(a) and 14(b) shows the results. VFs are slightly better than CFs in all cases.

Throughput in SSD. We evaluate the throughput of different methods in the SSD hard disk, where the memory access speed is much slower. And we limit the memory to 100MB. The results are shown in Fig 15. The results show that the lookup throughput of VF and VF-ss is around 1.86x to 2.13x faster than those of CF and CF-ss. The insertion experiments also show VFs provides higher throughput than CFs regardless of the occupancy.

6.3 Evaluation under dynamics

In this section, we evaluate the performance of vacuum filters in dynamic environments with frequent item set updates. For the comparison purpose, we build two systems. The **Linked-VF** system keeps creating new tables with items updates and links the tables in a chain. The **IUPR-VF** system uses the proposed IUPR update and reconstruction methods. We also replace vacuum filters by standard cuckoo filters to make a comparison between them. We apply the semi-sorting versions of VFs and CFs to save memory cost.We set length of fingerprints to be 13 bits. The vacuum filters and cuckoo filters are empty at the beginning. Updates happen with a stable frequency. The arrival time of the update events follows the Poisson distribution. The parameter λ denotes the average number of updates happening in the given time interval.


(a) Lookup throughput

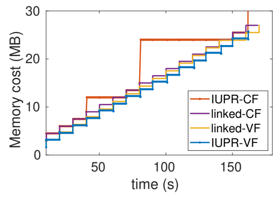

(b) Memory cost

Figure 16: IUPR-VF/CF versus Linked-VF/CF

We evaluate two metrics: lookup throughput and memory cost. For the memory cost, we only consider the memory consumption of the AWQ structures running in the fast memory. We consider two parameters that influence performance. 1) Reconstruction interval: In the implementation, we reconstruct the vacuum filter with a given interval. 2) The number of buckets: to choose a proper number of buckets when creating a new table, we set the number of buckets according to a pre-defined number, an estimated number of insertions in the next time interval.

Overall system performance. Fig. 16 shows the performance comparison of IUPR-VF, linked-VF, IUPR-CF, and linked-CF. In this set of experiments, the reconstruction interval is set to 20 seconds and λ is set to 10^6. From Fig.16(a), we find that IUPR-VF and IUPR-CF have higher lookup throughput than linked-VF and linked-CF over time. The reason is that the systems are always recovered to a good state every 20 seconds by resolving performance downgrading. We find that every 20 seconds, there is a sudden increase of the lookup throughput (in the 20s, 40s, and so on), meaning a replacement of the old version with a new version. However, the lookup throughput of linked-VF and linked-CF continuously decreases due to more linked tables caused by updates. We also find that IUPR-VF and IUPR-CF have a comparable lookup throughput. However, it may require much more memory for IUPR-CF to reconstruct the filter due to the restriction of the filter size as shown in Fig. 16(b).

Varying number of buckets in a new table. We use experiments to show the performance of varying the number of buckets for each new table. Every 10 seconds there will be about 10^6 new items need to be inserted into the data structure. In this experiment setting, every update we create a new table that can contain 10^6, 2×10^6, 3×10^6, and 4×10^6 items respectively. Fig. 17 shows the results of lookup throughput. We find that assigning a larger space when creating a new table has better lookup throughput for the linked-VF because it can decrease the number of table in the list. The drawback is that it consumes more memory when the newly linked vacuum filter is not full of sufficient updates, which can be shown in Fig. 18(a). In Fig. 18, the memory of both linked-VF and IUPR-VF increases about 1.6 MB every 10 seconds due to the new vacuum filter is created. And we find that the influence of the different size of a vacuum filter is small because of the reconstruction process. A larger number of buckets consume more memory during the system operation. In practice, assigning an estimated space for updates with proper reconstruction makes a good trade-off between the memory cost and lookup throughput.

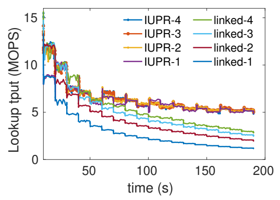
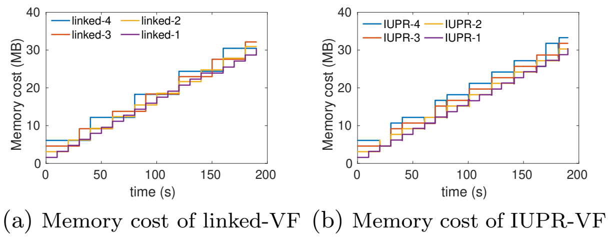


Figure 17: Lookup throughput vs. time (IUPR-x denotes that a VF contains about $x * 10^6$ items)



(a) Memory cost of linked-VF (b) Memory cost of IUPR-VF

Figure 18: Memory cost with different number of buckets

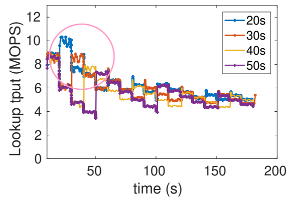


Figure 19: Lookup throughput with different reconstruction intervals

Table 3: Performance of AMQs for certificate checking applications

	bits per item			measured false positive r.			lookup thpt (pos)			lookup thpt (mixed)		
	VF-ss	CF-ss	BF	VF-ss	CF-ss	BF	VF-ss	CF-ss	BF	VF-ss	CF-ss	BF
valid	**12.752**	13.546	13.542	0.081%	**0.076%**	0.117%	**7.593**	7.455	2.177	**7.383**	6.250	4.184
	15.808	16.932	18.750	0.011%	0.0098%	**0.0090%**	**7.127**	7.001	1.473	**7.042**	6.045	2.900
revoked	**12.873**	20.978	13.542	**0.087%**	0.097%	0.112%	14.704	**14.868**	4.149	**13.754**	11.052	8.198
	16.091	26.699	17.709	**0.011%**	0.012%	0.014%	14.166	**14.650**	3.232	**13.797**	10.775	6.580

Varying reconstruction interval. We evaluate the influence of different reconstruction intervals by setting the interval to be 20s, 30s, 40s, and 50s respectively. Fig. 19 shows the results of lookup throughput. A smaller reconstruction interval means reconstructions happen more frequently, which is shown in the pink circle in Fig. 19. We find that the lowest lookup throughput of 20s is higher than that of 30s, 40s, and 50s. We also find that the smallest reconstruction interval achieves best lookup throughput from the figure 19 in the whole process, although it incurs higher computation overhead in the slow memory due to more frequent reconstruction calls. In practice, a real deployment may choose the smallest time interval allowed by the workstation running the update program to achieve a better lookup throughput.

6.4 Case study: checking revoked certificates

Vacuum filters can be used in many real applications [26, 5, 30, 27]. In this section, we study a case of using vacuum filters to check the certificate revocation status, which originally uses Bloom filters.

The TLS protocol, which relies on the public key infrastructure (PKI), is the cornerstone of Internet security. One critical problem for the web's PKI is the overlooked certificate revocation. If an erroneous or compromised certificate should be revoked, otherwise the client software will believe that the certificate is valid until it expires. Attackers can utilize such certificates to conduct the Man-in-the-Middle (MitM) attacks. However, many client applications do not properly check the revocation status of certificates. CRLite [30] proposes a client-server model to check certificate revocation status. The server aggregates the revocation information for all-known revoked certificates and generate a filter based on them. The clients download the filter and use it to check the revocation status. CRLite relies on Bloom filters to achieve low memory cost for running on mobile devices. In this study, we show that replacing Bloom filters with vacuum filters results in smaller memory cost for the same target false positive rate ϵ.

We use the data downloaded from Censys [6], which contain both revoked and non-revoked certificates. In this set of experiments, the total number of certificates is 30 million. The number of unrevoked certificates is 29,725,074, the number of revoked certificates is 274,926. For this application, we build three AWQ data structures to store the revoked and unrevoked certificates respectively. We set the target false positive rate to be 0.1% and 0.01%. For Bloom filters, we set the number of hash functions to achieve the best false positive rate. Table 3 shows the results. We can find that vacuum filters achieve the lowest memory cost with a target false positive rate compared to CFs and BFs in all cases. We also evaluate the lookup throughput with only existing items and the mixed items (containing 50% alien items). We can find that VFs and CFs have similar lookup throughput. BFs are the worst in both existing and mixed lookup throughput.

7. CONCLUSION

Vacuum filter, which is a type of AMQ data structures, is a more memory-efficient and faster replacement of Bloom and cuckoo filters. We made three main contributions in this work: 1) a fingerprint eviction strategy to achieve both high load factors and better data locality; 2) a new insertion algorithm that combines the advantages of both BFS and DFS to achieve a higher load factor and better insertion throughput; 3) an instant updates and periodical reconstructions (IUPR) method to resolve duplicate insertions and set resizing, both of which are considered difficult to handle in previous AMQ data structures. Experimental results and real case study show that vacuum filters require smaller memory in all cases and provide higher throughput in many situations, compared to existing methods.

8. ACKNOWLEDGMENT

M. Wang, S. Shi, and C. Qian were partially supported by NSF Grants 1717948, 1750704, and 1932447. We thank Tong Yang, Bin Fan, and the anonymous reviewers for valuable comments and/or discussions.

9. REFERENCES

[1] A fast alternative to the modulo reduction. Accessed: 2019-04-10.

[2] Campus LAN SDN Strategies: North American Enterprise Survey. `http://www.infonetics.com/pr/2015/Enterprise-LAN-SDN-Survey-Highlights.asp`.

[3] Cuckoo filter code. `https://github.com/efficient/cuckoofilter`, 2017.

[4] Implementation of Counting Quotient Filter. `https://github.com/splatlab/cqf`, 2017.

[5] Bloom filter in Bitcoin. `https://bitcoin.org/en/developer-guide#bloom-filters`, 2019.

[6] Censys. `https://censys.io/certificates`, 2019.

[7] Implementation of Morton Filter. `https://github.com/AMDComputeLibraries/Morton_filter`, 2019.

[8] Implementation of Vacuum Filters. `https://github.com/wuwuz/Vacuum-Filter`, 2019.

[9] The Ethereum Project. `https://www.ethereum.org/`, 2019.

[10] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In *Proc. of ACM SIGCOMM*, 2008.

[11] M. Athanassoulis and A. Ailamaki. BF-tree: approximate tree indexing. *PVLDB*, 7(14):1881–1892, 2014.

[12] 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. *PVDLB*, 5(11):1627–1637, 2012.

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

[14] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting bloom filters. In *Proc. of European Symposium on Algorithms*, 2006.

[15] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[16] A. D. Breslow and N. S. Jayasena. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *PVLDB*, 11(9):1041–1055, 2018.

[17] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*. Citeseer, 2002.

[18] 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. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[19] H. Chen, L. Liao, H. Jin, and J. Wu. The dynamic cuckoo filter. In *Proc. of IEEE ICNP*, 2017.

[20] N. Dayan, M. Athanassoulis, and S. Idreos. Optimal Bloom Filters and Adaptive Merging for LSM-Trees. *ACM Transactions on Database Systems (TODS)*, 43(4):16, 2018.

[21] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest Prefix Matching using Bloom Filters. In *Proc. of ACM SIGCOMM*, 2003.

[22] D. Eppstein. Cuckoo filter: Simplification and analysis. *arXiv preprint arXiv:1604.06067*, 2016.

[23] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *Proc. of the ACM CoNEXT*, 2014.

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

[25] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proc. of ACM SIGCOMM*, 2014.

[26] A. Gervais, S. Capkun, G. O. Karame, and D. Gruber. On the privacy provisions of bloom filters in lightweight bitcoin clients. In *Proc. of the ACM ACSAC*, 2014.

[27] M. Kwon, P. Reviriego, and S. Pontarelli. A length-aware cuckoo filter for faster IP lookup. In *IEEE INFOCOM WKSHPS*, 2016.

[28] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

[29] H. Lang, T. Neumann, A. Kemper, and P. Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *PVLDB*, 12(5):502–515, 2019.

[30] J. Larisch, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers. In *Proc. of IEEE Symposium on Security and Privacy*, 2017.

[31] D. Li, Y. Li, J. Wu, S. Su, and J. Yu. ESM: Efficient and Scalable Data Center Multicast Routing. *IEEE/ACM Transactions on Networking*, 2012.

[32] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proc. of the ACM EuroSys*, 2014.

[33] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM Computer Communication Review*, 2015.

[34] T. Maier, P. Sanders, and S. Walzer. Dynamic space efficient hashing. *Algorithmica*, 81(8):3162–3185, 2019.

[35] R. Mao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of ACM SIGCOMM*, 2017.

[36] M. Mitzenmacher, S. Pontarelli, and P. Reviriego. Adaptive cuckoo filters. In *Proc. of SIAM ALENEX*, 2018.

[37] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, 1990.

[38] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen,

K. Obraczka, and T. Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks . *IEEE Communications Surveys and Tutorials*, 2014.

[39] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 2004.

[40] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. A general-purpose counting filter: Making every bit count. In *Proc. of ACM SIGMOD*, 2017.

[41] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121. Springer, 2007.

[42] M. Raab and A. Steger. Balls into BinsA Simple and Tight Analysis. *Randomization and Approximation Techniques in Computer Science*, pages 159–170, 1998.

[43] C. E. Rothenberg, C. A. Macapuna, F. L. Verdi, and M. F. Magalhaes. The deletable Bloom filter: a new member of the Bloom family. *IEEE Communications Letters*, 14(6):557–559, 2010.

[44] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *Proc. of ACM SIGCOMM*, 2005.

[45] Z. Xie, W. Ding, H. Wang, Y. Xiao, and Z. Liu. D-Ary Cuckoo Filter: A space efficient data structure for set membership lookup. In *Proc. of IEEE ICPADS*, 2017.

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

[47] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *Proc. of USENIX NSDI*, 2013.

[48] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang. A Concise Forwarding Information Base for Scalable and Fast Name Lookups. In *Proc. of IEEE ICNP*, 2017.

[49] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *IEEE/ACM Transactions on Networking*, 2018.

[50] Y. Yu, C. Qian, and X. Li. Distributed collaborative monitoring in software defined networks. In *Proc. of ACM HotSDN*, 2014.