The Vertical Cuckoo Filters: A Family of Insertion-friendly Sketches for Online Applications

Pengtao Fu, Lailong Luo, Shangsen Li, Deke Guo, Geyao Cheng, Yun Zhou
Science and Technology on Information Systems Engineering Laboratory
National University of Defense Technology, Changsha Hunan 410073, P.R. China
{fupengtao, luolailong09, lishangsen, dekeguo, chenggeyao13, zhouyun}@nudt.edu.cn

***Abstract*—Cuckoo filter (CF) and its variants are emerging as replacements of Bloom filters in various networking and distributed systems to support efficient set representation and membership testing. Cuckoo filters store item fingerprints directly with two candidate buckets and a reallocation scheme is implemented to mitigate the bucket overflow problem for higher space utilization. Such a reallocation scheme, once triggered, however, can be time-consuming. This shortcoming makes the existing CFs not applicable for insertion-intensive scenarios such as online applications wherein the items join and leave frequently. To this end, in this paper, we propose the Vertical Cuckoo filter (VCF) which extends the standard Cuckoo filter by providing more candidate buckets to each item. Another challenging issue with such a design is how to ensure that the candidate buckets can be indexed by each other such that no additional hash computation and item access are necessary during fingerprint reallocation. Therefore, we present the vertical hashing, which indexes the candidate buckets with the fingerprint and given bitmasks. We further generalize and improve the VCF by realizing k (≥ 4) candidate buckets and avoiding unnecessary computation. The comprehensive experiments indicate that VCF outperforms its same kinds in terms of space utilization and insertion throughput, with a slight compromise of lookup speed.**

***Index Terms*—Cuckoo filter, vertical hashing, space utilization, insertion throughput, false positive rate, insertion-intensive**

I. Introduction

Many caches, routers and storage systems in networking and distributed systems, relying on space-efficient data structures to decide whether a given item exists in a large set of items [1] [2]. The most well-known data structures are Bloom filter (BF) [3] and its variants, which provide high memory efficiency at the cost of a few false positives. However, standard Bloom filters cannot delete existing items without rebuilding the entire filter. A body of work try to fix this drawback, including Counting Bloom filters (CBF) [4], d-left counting Bloom filters (dlCBF) [5], quotient filters [6], TinySet [7] and etc. These efforts, however, often suffer from degradation in either space or time efficiency, though Rank-Indexed Hashing [8] makes a minor improvement on space-saving.

To this end, Cuckoo filter (CF) [9] is proposed for better space utilization and faster deletion as a light-weight probabilistic data structure. Unlike Bloom filters, the Cuckoo filter stores the item fingerprints directly in two candidate buckets. If both candidates are fully occupied, CF kicks out a stored fingerprint from the candidate buckets to accommodate the coming item while reallocating the victim to its alternative bucket. Such a reallocation process ends successfully when no more victims are triggered or failed if the number of such reallocations reaches a given threshold. Efforts have been made to further improve its performance in terms of false positive rate (Adaptive Cuckoo filter [10]), flexibility (the consistent Cuckoo filter [11]), key set extension (Dynamic Cuckoo filter [12]), theoretical guarantee (the simplified Cuckoo filter [13]), lookup throughput (Vacuum filter [14], Morton filter [15]), etc.

However, the state-of-the-art Cuckoo filters rely on the reallocation process for higher space utilization, making item insertion time-consuming, especially when the filter is nearly full [16]. Such designs are not friendly for insertion-intensive online applications where items insert and delete frequently. An intuitive insight, as stated in D-ary Cuckoo filter (DCF) [17], is to explore more candidate buckets so that the filter can find an empty bucket slot without reallocation. Nevertheless, the d-ary cuckoo hashing is so cumbersome that a lot of calculation overhead and time consumption are caused in the insertion, lookup, and deletion processes.

This paper presents the Vertical Cuckoo filter (VCF), a new Cuckoo filter design that also implements more than two candidate buckets for each item. Standard CF can achieve higher space utilization (i.e., load factor) by extending the bucket size, while VCF does better in a vertical direction, i.e., increasing the number of candidate buckets. So we dub this design the Vertical Cuckoo filter. We propose a novel indexing scheme called vertical hashing to ease the reallocation process, which easily derives out the candidate buckets with item fingerprint and predefined bitmasks. This design benefits the VCF from two aspects. First, the candidate buckets for an arbitrary item can be indexed with each other without additional hash computation nor access to item content. Second, VCF needs less hash computation for item insertion than the existing Cuckoo filters. Because CF must execute one hash computation when reallocating an evicted fingerprint. In this manner, VCF achieves higher insertion throughput by avoiding unnecessary fingerprint reallocation and hash computation.

Moreover, we further improve VCF from two aspects. First, we generalize the number of candidate buckets for each item in VCF from the fixed 4 to a flexible k (≥ 4). By doing so, VCF provides a methodology to replace independent hash functions used by other sketches while still guaranteeing the

Deke Guo and Lailong Luo are the corresponding authors.

Table I
COMPARISON OF DATA STRUCTURES.

Data structure	Space	Throughput	Deletion
BF [3]	$1\times$	$1\times$	no
CBF [4]	$4\times$	$< 1\times$	yes
CF [9]	$\leq 1\times$	$\sim 10\times$	yes
4-ary CF [17]	$\leq 0.98\times$	$\sim 3.78\times$	yes
VCF	$\leq 0.98\times$	$\sim 20\times$	yes

randomness of the output. Second, we put forward two VCF variants to realize a proper trade-off between false positive rate and load factor. Specifically, the first variant realizes dynamical adjustments of performance by changing the bitmasks in VCF. While the second variant changes the threshold of fingerprint hash value to achieve almost the same effect as the first one.

We show the empirical results through a brief comparison among several related approximate membership lookup (AMQ) structures in both space utilization and insertion throughput in Table I. VCF has the most remarkable advantages compared to all other methods. The contributions of this paper can be summarized as follows:

- We present the vertical hashing strategy and use it to design VCF, a novel redesign of the Cuckoo filter that can provide multiple candidate buckets for each item.
- We further improve the VCF by generalizing the number of candidate buckets to k (≥ 4) and present two methods to make performance metrics tunable.
- We conduct theoretical analysis and comprehensive experiments to compare VCF with the existing methods. The numerical results demonstrate that VCF can cut the insertion time by half than CF and lead to an increment of space utilization, with an acceptable compromise of lookup speed.

The rest of this paper is organized as follows. Section II reviews the background and the related work; Section III describes the vertical hashing and the design principle of VCF. This section also shows how to generalize VCF to k-VCF; Section IV describes two variants of VCF. Section V presents a theoretical analysis of VCF in different situations and proves that VCF always offers the best performance. Section VI evaluates our design with comprehensive experiments compared with DCF and CF. Finally, Section VII concludes this work.

II. RELATED WORK

A. Bloom Filter and Its Variants

Bloom filter. As the most well-known AMQ data structure, Bloom filter (BF) provides a compact representation of a set of items, which supports set membership insertion and lookup effectively. A BF represents items with a bit array of length m, and all m bits are initially set as 0. For item insertion of x_i in the set $S = \{x_1, x_2, \ldots, x_n\}$, k independent hash functions $\{h_1, h_2, \ldots, h_k\}$ are employed to map the item to k positions in the array. The mapped positions, i.e., $h_j(x_i)$ where $j \in \{1, \ldots, k\}$, in the bit array are all set to 1. To achieve a set membership lookup, the BF checks the k corresponding positions in the bit array. If all bits are 1, the lookup returns true; otherwise, it returns false. BF is a space-efficient probabilistic data structure [18] with a known false positive probability ξ, and no false negative probability for set membership queries. To be specific, an item may be affirmed as the set member when its k hashed positions in the bit array are set as 1, due to the unavoidable hash conflicts. The false positive rate is $\xi_{BF} = (1 - e^{-kn/m})^k$. BF requires more space overhead to achieve lower false positives. Naturally, standard BF does not support item deletions. The reason is that resetting the corresponding bits from 1 to 0 directly may cause false negative results for other items.

Counting Bloom filter. Counting Bloom filter (CBF) [4] extends the BF by adding a counter for each item of the data structure. In this way, these counters help CBF to support item deletion without affecting the existence of other items [19]. CBF also supports constant-time membership lookup.

d-left Counting Bloom filter. dlCBF proposes to replace the general hash functions in CBF with a d-left hash function [20] [21]. Compared with CBF, dlCBF achieves half space-saving with the same false positive and two magnitude reduction of false positive rate with the same spatial scale.

B. Cuckoo Filter and Its Variants

Cuckoo filter. The recently proposed Cuckoo filter is a light-weight probabilistic data structure to support item deletions and constant-time membership queries. Structurally, a CF is a table of m buckets, each of which contains b slots. CF achieves satisfactory space efficiency for a given false positive rates [9] when $b = 4$. Every slot stores an f-bit fingerprint η_x of an item x, where $\eta_x = h_0(x) \ mod \ 2^f$. Standard CF provides two candidate buckets $B_1(x)$ and $B_2(x)$ for each item x through the partial cuckoo hashing technique [22]:

$$\begin{aligned} B_1(x) &= hash(x), \\ B_2(x) &= B_1(x) \oplus hash(\eta_x) \end{aligned} \tag{1}$$

The XOR operation in Equ. 1 ensures that the cyclic access of two candidate buckets for an item x can be achieved by utilizing only its fingerprint. To insert an item x, CF computes its fingerprint η_x and indexes of the two alternate buckets $B_1(x)$ and $B_2(x)$. Thereafter, the fingerprint η_x is stored in bucket $B_1(x)$ or $B_2(x)$ if there is an empty slot. Otherwise, if both candidates are occupied, the CF performs the eviction. It randomly kicks out a fingerprint in one of the candidates and then reinserts η_x in the slot that has just been vacated. At the same time, CF calculates the other candidate bucket of the victim fingerprint and tries to reinsert it into that bucket. The CF recursively performs the eviction until a bucket has an available slot or the number of such relocations reaches a given threshold MAX. In the process of membership lookup of set A, CF checks whether the fingerprint η_x is stored in the two candidate buckets $B_1(x)$ and $B_2(x)$. If that fingerprint is found, CF judges $x \in A$, otherwise, $x \notin A$. However, for an item $y \notin A$, CF may find a fingerprint that is the same as f-bit fingerprint of y in one slot of candidate buckets. The reason is that the potential hash collisions of fingerprints cause false positive errors. Theoretically, the false positive rate of CF is bounded as $\xi_{CF} = 1 - (1 - \frac{1}{2^f})^{2b} \approx \frac{b}{2^{f-1}}$ when the size of a fingerprint is f and each bucket has b slots.

Dynamic Cuckoo filters. Dynamic Cuckoo filter [12] supports the extension of the key set by using many linked homogeneous CFs. However, each lookup needs to check all linked CFs, which causes a dynamic Cuckoo filter with lower lookup throughput and a higher false positive rate.

Vacuum filters. Vacuum Filters (VF) [14] divides the whole table into multiple equal-size chunks and ensures two candidate buckets of each item in the same chunk. VF addresses the issue that CF can only achieve its claimed advantage in memory-efficiency when the size of the table is restricted to a power of two, with a slight improvement in space utilization and lookup throughput compared with CF.

Morton filters. Morton Filter (MF) [15] is designed to provide high lookup throughput for unique hierarchical memory systems. MF introduces a compressed block format that permits storing a logically sparse filter compactly in memory. Besides the fingerprint storage array, each MF block contains overflow flags and bucket counters by extending extra bits. MF is claimed faster than CF on the ARM architecture.

D-ary Cuckoo filter. By allowing that each item has more than two candidate buckets, D-ary Cuckoo Filter (DCF) [17] achieves efficient space utilization. DCF introduces the base-d digit-wise XOR operation that satisfies the limitation of only fingerprints available. Two keys X and Y need to be converted to their base-d form X_d and Y_d at first, as XOR here is defined in the base-d system. The result will be guaranteed to cycle back to X if it performs the base-d digit-wise XOR operation on Y for d times. Take $d = 4$ (each item has four candidate buckets) as an example, the operation XOR here guarantees:

$$X_4 = X_4 \oplus Y_4 \oplus Y_4 \oplus Y_4 \oplus Y_4 \tag{2}$$

However, Dynamic Cuckoo filter performs worse than CF in false positive rate and lookup efficiency. The performance improvements made by VF are negligible. MF only supports certain lengths of fingerprints (hence specific false positive rates). DCF needs to convert each index to base-d form and then convert it back to binary form, increasing time and calculation consumption of item insertion and lookup. In insertion-intensive scenarios, both comprehensive high-speed operations and maximized space utilization are essential for a proper filter. In contrast, these CF variants suffer from severe performance degradation that significantly restricts their applications in such scenarios. So we propose the Vertical Cuckoo Filter to solve these problems with excellent comprehensive performance and incredibly high insertion throughput.

III. Design of Vertical Cuckoo Filters

This section introduces a novel indexing method called vertical hashing to ease the reallocation when inserting an item. After that, the Vertical Cuckoo filter (VCF) is proposed based on the vertical hashing. Generalizations of both vertical hashing and VCF are also attached in this section.

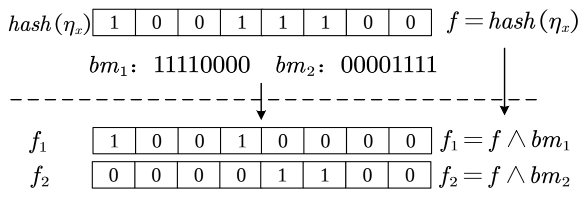


Figure 1. An illustrative example of the fragmentation of the fingerprint's hash value.

A. Vertical Hashing

1) Standard vertical hashing: To ease the fingerprint reallocation and computation, we present vertical hashing to index the candidate buckets with each other. It divides the fingerprint's hash value into multiple fragments using bitmasks (bm_s). The vertical hashing implement cyclical access to multiple buckets, which can be indexed by each other by performing xor operations on these fragments. Specifically, the implementation process is composed of two main steps.

First, for an f-bit fingerprint of each item to insert, vertical hashing divides its fingerprint into two fragments by performing AND operations on the hash value with two bitmasks, respectively. Each AND operation with a bitmask only keeps half of the fingerprint. In Fig. 1, we use an 8-bit fingerprint as an example, i.e., $f = 8$, and the fingerprint's hash value of the inserted item is $hash(\eta_x) = 10011100$ (expressed in binary). Thereafter, we derive out the two hash value fragments f_1 and f_2 by performing AND operations between the $hash(\eta_x)$ and bm_1 and bm_1, respectively. Thereby, the first fragment f_1 can be calculated as $f_1 = hash(\eta_x) \wedge bm_1 = 10010000$.

Second, based on the two fragments of fingerprint, we derive four candidate buckets that can be indexed by each other directly. Here is the detail:

$$\begin{aligned}
&B_1(x) = hash(x),\\
&B_2(x) = B_1(x) \oplus f_1 = B_1(x) \oplus hash(\eta_x) \wedge bm_1,\\
&B_3(x) = B_1(x) \oplus f_2 = B_1(x) \oplus hash(\eta_x) \wedge bm_2,\\
&B_4(x) = B_1(x) \oplus f = B_1(x) \oplus hash(\eta_x)
\end{aligned} \tag{3}$$

Theorem 1: Only when $bm_1 = \neg bm_2$, vertical hashing can circularly access all four candidate buckets for any item via its current location, fingerprint and bitmasks, without additional hash computation nor the access of its content.

Proof of Theorem 1: To displace an item x originally in a bucket B_i, vertical hashing directly calculates its other alternate buckets B_m, B_n and B_j from the current bucket index B_i and the fingerprint η_x stored in this bucket by:

$$\begin{aligned}
B_m &= B_i \oplus hash(\eta_x) \wedge bm_1,\\
B_n &= B_i \oplus hash(\eta_x) \wedge bm_2,\\
B_j &= B_i \oplus hash(\eta_x)
\end{aligned} \tag{4}$$

When $bm_1 = \neg bm_2$, substituting B_i of Equ. 4 with $B_2(x)$ in Equ. 3, we can derive that $B_m = B_1(x)$, $B_n = B_4(x)$ and $B_j = B_3(x)$. Let $B_i = B_1(x)$, $B_i = B_3(x)$ or $B_i = B_4(x)$, we have the same conclusion. ■

We further consider a special case wherein $hash(\eta_x)$ is shaped like $****0000$ or $0000****$ ($* = 1$ *or* 0) in the example of Fig. 1. Then we may only have two candidate buckets since $B_i, B_j = B_m$ *or* B_n. Assuming that the fingerprint size of VCF is f, we can give the probability of obtaining different alternate buckets successfully as follows.

$$P = 1 + 2^{-f} - 2^{1-\frac{f}{2}} \tag{5}$$

The probability that an item has only two candidate buckets is small. Precisely, in our case, only one-eighth of the insertions in Fig. 1 have two candidate buckets. With the increase of fingerprint length, this ratio will decrease rapidly. Besides, failing to extend the number of buckets does not affect the insertion, deletion, or query of VCF.

2) Generalized vertical hashing: We further enhance the basic vertical hashing by extending the two bitmasks to $k-2$ ($k > 4$) bitmasks. Suppose that there are $k-2$ random different bitmasks with the same size as the hash value of fingerprints, then all of the k candidate buckets for a given item x can be derived as follows:

$$\begin{aligned}
&B_1(x) = hash(x),\\
&B_2(x) = B_1(x) \oplus hash(\eta_x) \wedge bm_1,\\
&B_3(x) = B_1(x) \oplus hash(\eta_x) \wedge bm_2,\\
&\ldots\ldots\\
&B_{k-1}(x) = B_1(x) \oplus hash(\eta_x) \wedge bm_{k-2},\\
&B_k(x) = B_1(x) \oplus f = B_1(x) \oplus hash(\eta_x)
\end{aligned} \tag{6}$$

Theorem 2: According to Equ. 6, for an item x, its any candidate bucket $B_e(x)$ ($1 < e < k$) can be derived out through any other candidate bucket $B_g(x)$ ($1 < g < k$) with the equation as follows.

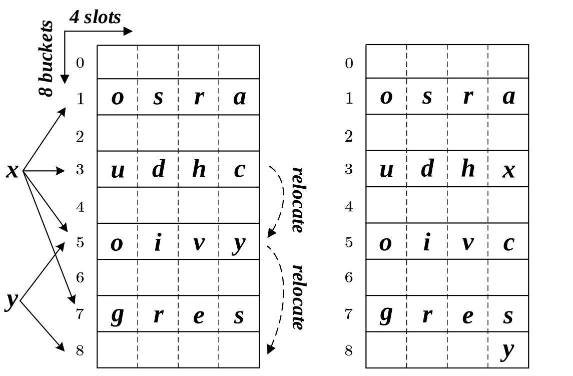
$$B_e(x) = B_g(x) \oplus hash(\eta_x) \wedge bm_g \oplus hash(\eta_x) \wedge bm_e \tag{7}$$

Proof of Theorem 2: Since $B_g(x) = B_1(x) \oplus hash(\eta_x) \wedge bm_g$, we have $B_g(x) \oplus hash(\eta_x) \wedge bm_g = B_1(x)$. Substituting this equation into Equ. 7, it is evident that $B_e(x) = B_1(x) \oplus hash(\eta_x) \wedge bm_e$. It proves that the generalized vertical hashing can also circularly access all $k-1$ candidate buckets of $B_e(x)$ with the bm_e as prior knowledge. ■

B. The Vertical Cuckoo Filters

Compared to partial cuckoo hashing, vertical hashing provides more candidate buckets for a given item, so each item has a greater probability of finding an empty slot during the insertion process, reducing the hash computation of fingerprint reallocations. Moreover, vertical hashing is far more concise in calculation than base-d digit-wise XOR operation in DCF. Therefore, using vertical hashing can help VCF obtain higher space utilization and insertion throughput than CF and faster insertion, lookup, and deletion than DCF.

Like CF, VCF also uses slots as the basic unit of the cuckoo hash tables, and each slot stores only one fingerprint. Several such slots constitute a bucket, and these ranked buckets construct a hash table for VCF. By using vertical hashing, VCF can generally index four candidate buckets with the fingerprint and bitmasks. However, VCF may only get two candidate buckets as the insertion of item y shown in Fig. 2.



(a) Before inserting item x. (b) After inserting item x.

Figure 2. An illustrative example of VCF insertion.

Insertion. The insertion of VCF is shown in Fig. 2. Suppose x is the item to be inserted, and the sequence numbers of its four candidate buckets are 1, 3, 5, and 7, as shown in Fig. 2(a). If any of these four candidate buckets have an empty slot, we can store x there. Otherwise, if all candidate buckets (1, 3, 5, and 7) are occupied, VCF will randomly choose one of the candidate buckets, say bucket 3, and evict an item randomly from it, e.g., item c. Then we reinsert this victim item to its other alternate buckets. In this example, the victim c will trigger another relocation of item y in bucket 8. Then y will be reallocated to its alternative candidate bucket, i.e., bucket 5. This allocation procedure continues until an empty slot is available or the times of such reallocations reach the predefined threshold. The final storage state is shown in Fig. 2(b). Once the eviction times reach the threshold, the VCF is considered too full to insert more items. Thus, the VCF insertion returns an insertion failure. The details are shown in Algorithm 1.

Lookup. The item lookup of VCF is shown in Algorithm 2. For a given item x, the algorithm first calculates its fingerprint and four candidate buckets according to Equ. 3. When any existing fingerprint in these bucket matches, the VCF returns true; otherwise, it returns false. VCF has no false negatives, just like CF. Notice that if VCF fails to provide four candidate buckets, only two buckets are involved in the lookup process.

Deletion. The deletion process of VCF is detailed in algorithm 3. Similar to the lookup process, it checks all four candidate buckets for the given item first. If any matched fingerprint is found, one copy of that fingerprint will be removed from the bucket. Similar to CF, VCF also overcomes the mis-deletion problem. To be specific, even if several items have the same candidate buckets and also have the same fingerprint in one of these shared buckets, VCF can delete all or part of these items safely as long as they have been previously inserted.

Algorithm 1: Insert (x) in VCF

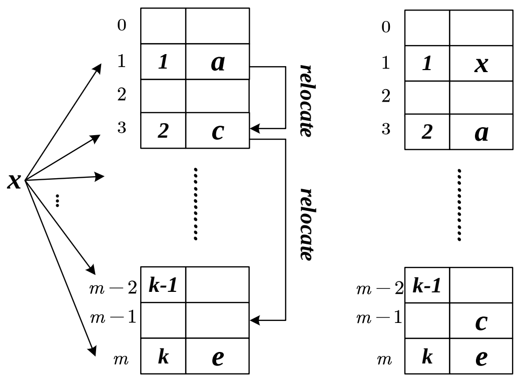
```
1  define $bm_1$ and $bm_2$;
2  $f$=fingerprint($x$);
3  $i_1$ = hash($x$);
4  $i_2$ = $i_1 \oplus$ hash($f$) $\wedge$ $bm_1$;
5  $i_3$ = $i_1 \oplus$ hash($f$) $\wedge$ $bm_2$;
6  $i_4$ = $i_1 \oplus$ hash($f$);
7  if Bucket[$i_1$] or Bucket[$i_2$] or Bucket[$i_3$] or Bucket[$i_4$] has an empty slot then
8      add $f$ to that bucket;
9      return Done;
10 // must relocate existing items;
11 $B_i$ = randomly pick $i_1$, $i_2$, $i_3$ or $i_4$;
12 for s=0; s<MaxNumKicks; $s++$ do
13     randomly select a slot $e$ from Bucket[$B_i$];
14     swap $f$ and the fingerprint stored in slot $e$;
15     $B_m$ = $B_i \oplus$ hash($f$) $\wedge$ $bm_1$;
16     $B_n$ = $B_i \oplus$ hash($f$) $\wedge$ $bm_2$;
17     $B_j$ = $B_i \oplus$ hash($f$);
18     if Bucket[$z$] ($z = B_m$, $B_n$ or $B_j$) has an empty slot then
19         add $f$ to Bucket[$z$];
20         return Done;
21     $B_i$ = randomly pick $B_m$, $B_n$ or $B_j$;
22 // Hashtable is considered full;
23 return Failure
```

Algorithm 2: Lookup (x) in VCF

```
1 $f$=fingerprint($x$);
2 $i_1$ = hash($x$);
3 $i_2$ = $i_1 \oplus$ hash($f$) $\wedge$ $bm_1$;
4 $i_3$ = $i_1 \oplus$ hash($f$) $\wedge$ $bm_2$;
5 $i_4$ = $i_1 \oplus$ hash($f$);
6 if Bucket[$i_1$] or Bucket[$i_2$] or Bucket[$i_3$] or Bucket[$i_4$] has $f$ then
7     return True;
8 return False
```

Algorithm 3: Delete (x) in VCF

```
1 $f$=fingerprint($x$);
2 $i_1$ = hash($x$);
3 $i_2$ = $i_1 \oplus$ hash($f$) $\wedge$ $bm_1$;
4 $i_3$ = $i_1 \oplus$ hash($f$) $\wedge$ $bm_2$;
5 $i_4$ = $i_1 \oplus$ hash($f$);
6 if Bucket[$i_1$] or Bucket[$i_2$] or Bucket[$i_3$] or Bucket[$i_4$] has $f$ then
7     remove a copy of $f$ from that bucket;
8     return True;
9 return False
```



(a) Before inserting item x. (b) After inserting item x.

Figure 3. An illustrative example of k-VCF insertion.

C. Generalization of Vertical Cuckoo Filters

VCF can be generalized to k-VCF that has k ($k > 4$) candidate buckets for each given item, using the generalized vertical hashing and additional mark bits. k-VCF can also circularly access all of the k candidate buckets.

However, k-VCF does not satisfy *Theorem 1* like VCF, so it must add the mark bits to label the bitmasks, i.e., e^{th} of bm_e in Equ. 7 for each slot. Consequently, each slot must have two fields, the fingerprint field and the counter field. For example, k-VCF needs to add extra three bits as a counter field based on a fingerprint when $k = 7$. Take Fig. 3 as an example, to insert an item x into the k-VCF where all of the candidate buckets are full, the item eviction process is triggered. Then k-VCF will randomly choose a victim bucket, such as bucket 1 in Fig. 3(a), as well as a random slot in that bucket. After accessing both the fingerprint field and the counter field of bucket 1, k-VCF will compute another candidate bucket by Equ. 7, such as bucket 3 in Fig. 3(a). This insertion procedure will end until a bucket with an empty slot is found, such as bucket $m-1$ in Fig. 3(a) or the times of such displacements reach the predefined threshold.

Most current sketch data structures, such as Count-Min Sketch [23], Adaptive Sketch [24], and Level Hashing [25], have to execute two or more hash calculations to index the corresponding blocks. By contrast, k-VCF only requires one hash computation, i.e., to derive out the item fingerprint. Therefore, k-VCF can provide a much faster lookup in most cases. Besides, most concurrent cuckoo hash tables fail to efficiently address the problem of endless loops during item insertion due to the potential hash collisions [26]. However, as the number of candidate buckets increases, the probability of such endless loop formation in both VCF and k-VCF can be significantly reduced.

IV. Variants of Vertical Cuckoo Filters

VCF improves the load factor by increasing the candidate buckets while also causes an increase in false positives. Although the easiest way to reduce the false positive rate is to reduce the number of slots of buckets in VCF, this method will result in a more significant compromise of space utilization. Extensive experiments have proved that VCF uses buckets of size four cannot improve CF with buckets of size two or three under the same table size. Therefore, reducing the number of slots in candidate buckets of VCF is not an advisable choice.

To this end, all VCFs and their variants mentioned in this paper keep four slots per candidate bucket. We introduce two innovative methods called Inversed Vertical Cuckoo Filters (IVCF) and Differentiated Vertical Cuckoo Filters (DVCF) to make a proper trade-off between load factor and false positive rate. As k-VCF is different from VCF and CF structurally, IVCF and DVCF are not applied to k-VCF.

A. The Inversed Vertical Cuckoo Filters

IVCF changes the bitmask forms in VCF and realizes the trade-off among false positive rate, load factor, and throughput of insertion, lookup, and deletion. As long as the two bitmasks satisfy $bm_1 = \neg bm_2$, such as 0101010 and 1010101, they can be implemented in VCF. When all bits in bm_1 are 0 or 1, VCF will be degraded as CF and only provide two candidate buckets, as mentioned in Section III-A.

Assume that VCF use f-bits fingerprints, and there are l $(0 < l < f)$ 0s and $f - l$ 1s in bm_1. We can formulate the probability P that VCF will provide 4 candidate buckets as follows.

$$P = 1 - \frac{2^l + 2^{f-l} - 1}{2^f} \approx 1 - 2^{l-f} - 2^{-l}, \qquad (8)$$

where l is the number of 0s in bm_1.

From the above equation, as long as the value of $|2l - f|$ decreases while f remains unchanged, the value of P will increase. A larger f will also cause a higher P with the same $|2l - f|$. For example, when $f = 8$, the probability P will approximate to $\frac{7}{8}$ when $l = 4$, while P decreases rapidly to $\frac{1}{2}$ when $l = 7$. In addition, suppose that VCF increases the length of fingerprint to 16 bits and $2l - f$ remains 0, i.e. $f = 16$ and $l = 8$, then $P \approx 0.9922$. With such a probability, VCF can provide 4 candidate buckets for most items.

The failure to provide four candidate buckets may lower the overall load factor for item insertion and lower the false positive rate for membership testing. In other words, larger P causes a higher load factor and also a higher false positive rate, and vice versa. Therefore, we can implement a proper trade-off between load factor and false positive rate by altering the P by adjusting the bitmasks. We name this method the Inversed Vertical Cuckoo Filters, meaning to convert VCF to CF through a lower P. As the length of fingerprint f is unchanged in IVCF, we can only change the number of 0-bits in bm_1, i.e., l, to achieve the performance trade-off. However, IVCF cannot arbitrarily look for trade-offs because IVCF can only set the bitmasks into a few fixed forms, causing P to be discrete, such as $P \approx \{0, 0.49, 0.73, 0.84, 0.87\}$ when $f = 8$.

It is proved that different bitmask forms will lead to an unequal probability of scaling the number of candidate buckets. IVCF takes advantage of this property to achieve a flexible trade-off. Besides, the failure to increase the number of buckets does not affect the insertion, lookup, and deletion operations. Consequently, Algorithm 1, Algorithm 2 and Algorithm 3 in Section III-B are also appropriate for IVCF.

Algorithm 4: Insert (x) in DVCF

```
1  define $bm_1$, $bm_2$ and the threshold $\Delta t$;
2  $f$=fingerprint($x$);
3  if $f \in [T/2 - \Delta t, T/2 + \Delta t]$ then
4      $i_1$ = hash($x$), $i_2$ = $i_1 \oplus$ hash($f$) $\wedge$ $bm_1$, $i_3$ = $i_1 \oplus$
       hash($f$) $\wedge$ $bm_2$, $i_4$ = $i_1 \oplus$ hash($f$);
5      if Bucket[$i$]($i = i_{1,2,3\ or\ 4}$) has an empty slot then
6          add $f$ to that bucket;
7          return Done;
8  else
9      $i_1$ = hash($x$), $i_2$ = $i_1 \oplus$ hash($f$);
10     if Bucket[$i_1$] or Bucket[$i_2$] has an empty slot then
11         add $f$ to that bucket;
12         return Done;
13 $B_i$ = randomly pick one from any Bucket[$i$];
14 for s=0; s<MaxNumKicks; s++ do
15     randomly select a slot $e$ from Bucket[$B_i$];
16     swap $f$ and the fingerprint stored in slot $e$;
17     if $f \in [T/2 - \Delta t, T/2 + \Delta t]$ then
18         $B_m$ = $B_i \oplus$ hash($f$) $\wedge$ $bm_1$, $B_n$ = $B_i \oplus$ hash($f$)
           $\wedge$ $bm_2$, $B_j$ = $B_i \oplus$ hash($f$);
19         if Bucket[$z$] ($z = B_m, B_n$ or $B_j$) has an empty
           slot then
20             add $f$ to Bucket[$z$];
21             return Done;
22         $B_i$ = randomly pick $B_m$, $B_n$ or $B_j$;
23     else
24         $B_j$ = $B_i \oplus$ hash($f$);
25         if Bucket[$B_j$] has an empty slot then
26             add $f$ to Bucket[$B_j$];
27             return Done;
28         $B_i$ = $B_j$;
29 return Failure
```

B. The Differentiated Vertical Cuckoo Filters

To solve the problem that IVCF cannot arbitrarily seek trade-offs, we present DVCF, which uses the same bitmasks as standard VCF. At first, DVCF divides the value range of $hash(\eta_x)$ in Equ. 3 into two parts by setting a threshold Δt. The first interval In_1 is $[T/2 - \Delta t, T/2 + \Delta t]$, and the second interval In_2 is $[0, T/2 - \Delta t] \cup [T/2 + \Delta t, T]$. Thereafter, DVCF differentiates the fingerprints lie in these two intervals and provides different number of candidate buckets to them.

As shown in Algorithm 4, to insert an item x, DVCF will first calculate the fingerprint and then determine which interval this fingerprint belongs to. If it belongs to the first interval, i.e., In_1, then DVCF will obtain its four candidate buckets by

Equ. 3; otherwise, DVCF will derive out two candidate buckets by Equ. 1. During each relocation, the judgment about the victim's location is necessary before reinserting this victim.

The insertion method in Algorithm 4 provides 4 candidate buckets for all the coming items which belong to In_1 since $B_i, B_j = B_m$ *or* B_n in Equ. 4 is not satisfied. In other words, DVCF is the combination of VCF and CF by implementing the intervals. Therefore, assume DVCF uses f-bit fingerprints, the proportion p of items with four candidate buckets can be calculated as follows.

$$p = \frac{2\Delta t}{T} = \frac{2\Delta t}{2^f} \tag{9}$$

Algorithm 5: Lookup (x) in DVCF

```
1  $f$=fingerprint($x$);
2  if $f \in [T - \Delta t, T + \Delta t]$ then
3      $i_1$ = hash($x$);
4      $i_2$ = $i_1 \oplus$ hash($f$) $\wedge$ $bm_1$;
5      $i_3$ = $i_1 \oplus$ hash($f$) $\wedge$ $bm_2$;
6      $i_4$ = $i_1 \oplus$ hash($f$);
7      if $Bucket[i_1]$ or $Bucket[i_2]$ or $Bucket[i_3]$ or $Bucket[i_4]$
       has $f$ then
8          return True;
9  else
10     $i_1$ = hash($x$);
11     $i_2$ = $i_1 \oplus$ hash($f$);
12     if $Bucket[i_1]$ or $Bucket[i_2]$ has $f$ then
13         return True;
14 return False
```

Algorithm 6: Delete (x) in DVCF

```
1  $f$=fingerprint($x$);
2  if $f \in [T - \Delta t, T + \Delta t]$ then
3      $i_1$ = hash($x$);
4      $i_2$ = $i_1 \oplus$ hash($f$) $\wedge$ $bm_1$;
5      $i_3$ = $i_1 \oplus$ hash($f$) $\wedge$ $bm_2$;
6      $i_4$ = $i_1 \oplus$ hash($f$);
7      if $Bucket[i_1]$ or $Bucket[i_2]$ or $Bucket[i_3]$ or $Bucket[i_4]$
       has $f$ then
8          remove a copy of $f$ from that bucket;
9          return True;
10 else
11     $i_1$ = hash($x$);
12     $i_2$ = $i_1 \oplus$ hash($f$);
13     if $Bucket[i_1]$ or $Bucket[i_2]$ has $f$ then
14         remove a copy of $f$ from that bucket;
15         return True;
16 return False
```

Table II
THE NOTATIONS USED FOR OUR ANALYSIS.

Notation	Explanatory
f	the length of fingerprint in bits
α	the load factor $(0 \le \alpha \le 1)$
b	the number of slots per bucket
m	the number of buckets
n	the size of filter
d	the number of candidate buckets
ξ	the false positive rate
r	the probability of providing 4 candidate buckets
C	the average bits per item
β	the rate of increase in load factor

Indeed, the p in Equ. 9 is the same as the P in Equ. 8. The lookup and deletion operations of DVCF are also similar to VCF, but with one more step of judgment, as shown in Algorithm 5 and Algorithm 6 respectively. In conclusion, DVCF achieves the trade-off between load factor and false positive by adjusting the threshold, i.e., Δt, at the cost of one more step of judgment.

V. PERFORMANCE ANALYSIS

We present the analysis results on VCF in terms of the load factor, false positive rate, space and time cost in this section. The notations used for our analysis are shown in Table II.

A. Load Factor

We define the load factor as $\alpha = n/bm$, i.e., the ratio between the number of stored fingerprints and the filter's capacity. In reality, there is no theoretical result of the α that fits the four-slot table design [14]. Therefore, we use experiments to test the maximum load factor for the standard CF and VCF with the same parameter settings. The experiments show that a VCF with million items can realize around 98% loads when each bucket holds four fingerprints with 7 bits. VCF can achieve almost 100% of the load factor when $f = 18$, as shown in Fig. 4. If the hash table stores relatively short fingerprints, the chance of insertion failures will increase due to hash collisions, reducing the table occupancy ratio. Hence a larger f leads to a higher load factor.

B. False Positive Rate and Space Cost

In our designs, the false positive rate of VCF depends on three factors: 1) the length of fingerprint f; 2) the probability

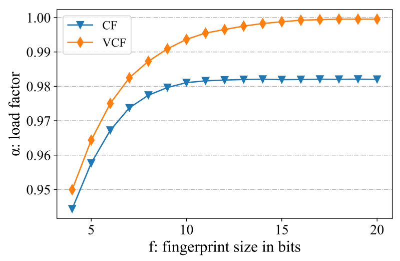


Figure 4. The load factor achieved with different fingerprint length in tables with 2^{20} slots.

of successfully providing four candidate buckets, denoted as r ($r = P$ or p, given in Equ. 8 and Equ. 9, respectively); 3) the number of slots b in each bucket (usually set to 4).

When looking up a non-existent item x in a slot, the probability of mismatch (i.e., false positive error), is at most $1/2^f$. The lookup algorithm must probe all slots in its candidate buckets, and the number of these slots is related to both α and d. Hence, the expected number of fingerprint comparisons is $(2r+2)b\alpha$. Therefore, the upper bound of the total probability of false positive rate can be calculated as:

$$\xi = 1 - (1 - \frac{1}{2^f})^{(2r+2)b\alpha} \approx \frac{2(r+1)b\alpha}{2^f} \tag{10}$$

In Equ. 10, ξ is proportional to the average bucket size $2(r+1)b$. We use $\beta = (\alpha - \alpha')/\alpha'$ to represent the growth rate of load factor, where α' is the load factor of CF. We conclude that the false positive rate of VCF increases $(\beta+1)(r+1)$-fold compared to CF (where $r = 0$). Furthermore, the minimal fingerprint size for a given target false positive ξ is:

$$f \geq \lceil \log_2(2(r+1)b\alpha/\xi) \rceil \tag{11}$$

Similar to ξ, f depends on the average bucket size $2(r+1)b$. For a given set of items, the average space cost C is:

$$C = \lceil \log_2(2(r+1)b\alpha/\xi) \rceil / \alpha \tag{12}$$

Unlike the standard CF, VCF can not only improve α by increasing b but also adjust α and ξ by dynamically changing the value of r. Because VCF improves the load factor such that it can store more items with the same filter size compared to standard CF, VCF is more space-saving than CF. For example, when setting $b = 4$ and $f = 10$, the α of CF is 0.95. So the bits per item for CF is $3.08 + 1.05\log_2(1/\xi_0)$, where ξ_0 is the false positive rate of CF. Given $r = 1/2$, the load factor for VCF is $\alpha = 98\%$. Therefore, the bits per item for VCF is $2.98 + 1.02\log_2(1/\xi_0)$, which is smaller than CF.

C. Time Cost

The time cost for each lookup or deletion is constant in both CF and VCF, as either of them only needs about $2(r+1)$ memory accesses. However, the value of r in CF is 0, which means VCF needs more time to look up or delete with additional $2r$ memory accesses.

Wang et al. [14] use the average traversed number of buckets to reflect the time cost for each insertion of CF indirectly. Notice that the searching process follows the Bernoulli distribution with a successful rate $1 - \alpha^{(2r+1)b}$ in the insertion algorithm. Therefore, the probability that all the $(2r+1)b$ slots in the candidate buckets for an evicted fingerprint are full is $\alpha^{(2r+1)b}$. The reason is that all fingerprints are uniformly distributed among the buckets. Let π be the number of evicted fingerprints, and $E(\pi_\alpha)$ be the expectation of π under the load factor α, then $E(\pi_\alpha)$ can be estimated by:

$$E(\pi_\alpha) = (1 - \alpha^{(2r+1)b}) + \alpha^{(2r+1)b}(E(\pi_\alpha)+1) \tag{13}$$

We have $E(\pi_\alpha) = 1/(1 - \alpha^{(2r+1)b})$, then the average insertion cost E for serial insertions from load factor 0 to α can be derived from the following formula:

$$E = \int_0^\alpha E(\pi_x)dx = \int_0^\alpha 1/(1 - x^{(2r+1)b})dx \tag{14}$$

In order to better analyze the experimental results, we further improve Equ. 14 as Equ. 15, where λ means the number of slots in the filter, while λ_0 demotes the number of inserted items when the filter reaches the theoretical maximum space utilization. For example, let $r = 0$, $b = 4$, $\alpha = 0.95$ and $\lambda_0/\lambda = 0.98$, then we have $E_0 = 11.3$, which means the standard CF needs to kick out almost 11.3 fingerprints for each inserted item in our experiments. While with $r \approx 1$, $b = 4$, $\alpha = 0.995$ and $\lambda_0/\lambda \approx 1$, we have $E_0 = 1.22$ for VCF. The experimental results show that the VCF can significantly reduce the number of eviction operations to effectively decrease the time cost for insertion compared to the standard CF.

$$E_0 = \frac{\lambda_0}{\lambda}E + 500(1 - \frac{\lambda_0}{\lambda}) \tag{15}$$

As mentioned in Section III-C, the endless loops during item insertion will drastically impact space utilization and waste time. Besides, due to the potential endless loops in insertion, the filter suffers from its performance reliability problems. However, VCF dramatically reduces the probability of endless loops as the number of candidate buckets increases. In this way, VCF can significantly improve space utilization, save time cost, and make the insertion performance more robust.

VI. Evaluation

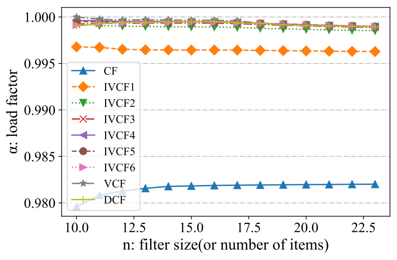
This section empirically compares the performance of IVCF, DVCF with the standard CF and DCF. To demonstrate the effectiveness and efficiency of VCF, we test all filters on a real-world dataset. We describe our experimental settings and present the results of extensive simulations from several orthogonal metrics: load factor, time consumption, false positives, and the impact of parameters, i.e., the number of candidate buckets k and the type of hash functions. The code for Vertical Cuckoo Filter is opensource[1].

A. Experimental Settings

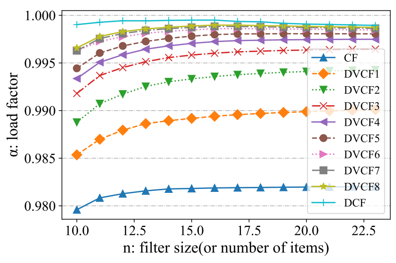
Datasets. We use the real-world dataset Higgs[2] as the input of the filters. There are in total 28 kinematic features obtained through particle detectors. We merge the third and fourth features and then perform the data pre-processing through deduplication to build the dataset used in our experiments.

Metrics. For item insertion, we use *load factor* and *time-consumption* to test the insertion throughput. We also indirectly test the insertion time consumption through the average number of fingerprint evictions. For item query, we compare the lookup throughput via the average response time. After querying all items that have never been stored, the ratio

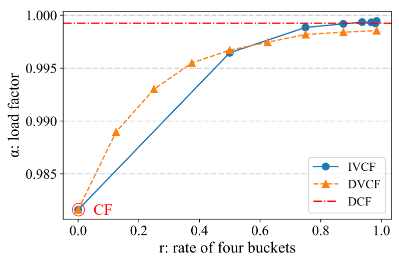
[1]https://github.com/fptjy/vertical-cuckoo-filter
[2]https://archive.ics.uci.edu/ml/datasets/HIGGS



(a) α of IVCF.



(b) α of DVCF.



(c) Comparison of IVCF and DVCF in α.

Figure 5. The load factor for item insertion of *IVCFs* and *DVCFs*, with respect to the filter size.

between the number of TRUE returned by the filters and this dataset's size is recorded as their *false positive rate*. Besides, we design several comprehensive experiments to explore the impact of the parameter k in $k-$VCFs. Furthermore, we run the filters with diverse hash functions to quantify their impacts.

Experiment setup. For all experiments, let $MAX = 500$ and $f = 14$. We vary the value of r from a small value to nearly 1 in IVCFs and DVCFs. The hash function used in our experiments is FNV hash[3]. All experiments are conducted in a machine with an Intel Core i7 processor and 16GB DRAM. We run each experiment one thousand times and then report the average value. To understand how the IVCF and DVCF achieve the trade-off between different performances, we utilize seven IVCFs and eight DVCFs with different values of r. To be specific, let IVCF$_i$ represent the IVCF that contains i 1s in its first bitmask, and DVCF$_j$ denote the DVCF that satisfies $2\Delta t = j \times 0.125 \times 2^{14}$, where i and j are positive integers. Note that r can be derived from Equ. 8 and Equ. 9.

Baselines. To better illustrate the advancements of vertical hashing, we conducted experiments on CF, DCF ,and the variants of VCF simultaneously with the same experimental settings. As baselines, CF can be regarded as a kind of IVCFs or DVCFs when $r = 0$. Besides, we fix d in DCF as 4.

B. Numerical Results

In this subsection, we present numerical results of filters, for item insertion (load factor and time consumption), lookup (average response time and false positive rate), and more (different k in k-VCFs and hash functions) in Table III. These results verify that all performance metrics of VCF are related to r, yielding a trade-off among space, speed and accuracy.

Table III
THE RESULTS OF LF (LOAD FACTOR), IT (AVERAGE INSERT TIME), QT (AVERAGE MIXED QUERY TIME) AND FPR (FALSE POSITIVE RATE).

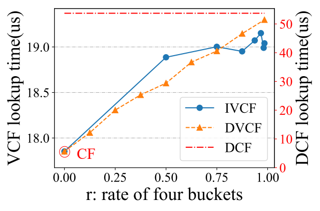
	LF(%)	IT(us)	QT(us)	FPR($\times 10^{-3}$)
CF	98.16	15.86	18.10	0.485
DCF	99.94	32.80	53.77	0.967
VCF1-VCF	99.64-99.95	13.40-8.95	19.42-19.62	0.739-0.974
DVCF1-8	98.90-99.85	14.72-10.21	18.20-19.88	0.547-0.974

[3]http://www.isthe.com/chongo/tech/comp/fnv/index.html

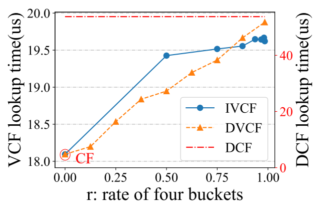
1) Load Factor: We evaluate the load factor α for IVCF and DVCF with the same filter size n (assume that $n = 2^\theta$, and the parameter θ is varied from 10 to 23). In the experiments, we first select n items randomly from the real-world dataset and feed them to an empty filter with n slots. A small portion of items fail to be stored as the number of eviction fingerprints reached the predefined threshold. We also calculate the total average value of α under different filter sizes to compare the performance of VCFs using two different trade-off methods. Fig. 5 shows the load factor achieved as we vary two parameters: 1) the probability of providing 4 candidate buckets, i.e., r; 2) the filter size n.

Fig. 5 records the improvement of load factor α with increment of r. When $r = 0.9844$, the table occupancy ratio of CF is only 98.2%, while the load factors of IVCF and DVCF can reach near 100% and 99.85%, respectively. Compared with CF, IVCF$_i$ (i=1,2,. . . ,6) in Fig. 5(a) and DVCF$_i$ (i=1,2,. . . ,8) in Fig. 5(b), can improve space utilization significantly. Furthermore, the load factor of DVCF degrades when the filter size decreases, while IVCF has no such problem. We also calculate all load factor values of a filter with different r. As shown in Fig. 5(c), the α of IVCF and DVCF, increase consistently when r grows. Besides, with the same r, the load factor of IVCF is slightly higher than that of DVCF. The experiment result of IVCF is a set of discrete points because IVCF cannot continuously adjust two bitmasks. Since r has no impact on DCF, DCF has a constant load factor as VCF.

2) Time Consumption: **Lookup time consumption.** In the lookup experiments, we fix the number of both slots in filters and dataset size as 2^{20}. Fig. 6 shows the average time consumption for each lookup in the following two cases: 1) 100% of existing items; 2) 50% - 50% mix of existing and

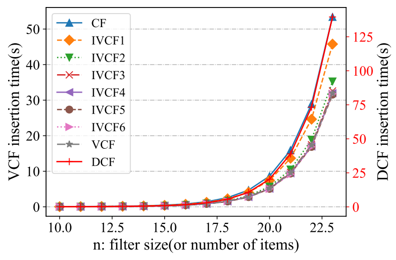


(a) Lookups for existing items.

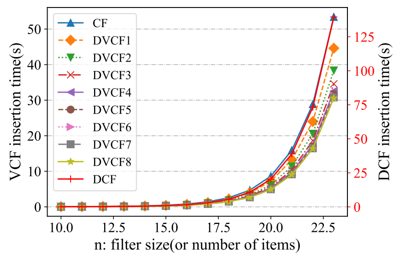


(b) Lookups for mixed items.

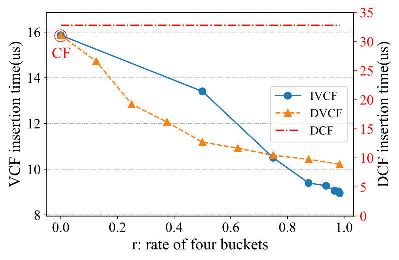
Figure 6. The time consumption of item lookup with different r.



(a) Insert time of IVCF.



(b) Insert time of DVCF.



(c) Comparison of IVCF and DVCF.

Figure 7. The time consumption of item insertion for *IVCFs* and *DVCFs*, with respect to the filter size.

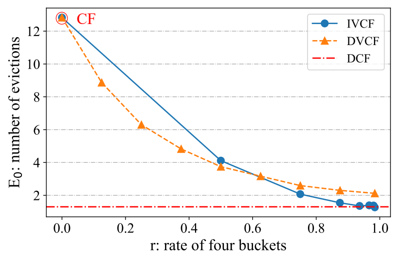


Figure 8. The average number of eviction for item insertion of *IVCFs* and *DVCFs* when the filter size is fixed as 2^{20}.

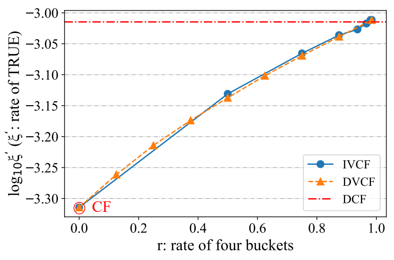


Figure 9. The false positive rate with respect to the r, when the filter size is set as 2^{20}.

alien items. Results show that the lookup time overhead of IVCFs is constant $0.06\times$ and $0.08\times$ more than the standard CF for positive and mixed lookups, respectively. The reason is that, for any given item, no matter whether the number of its candidate buckets increases successfully or fails, its lookup needs to operate on all four buckets (which may be duplicates).

The lookup time of DVCFs has a significant positive relationship with the value of r, which ranges from 0.125 to 0.9844. In general, the lookup performance of both IVCF and DVCF is slightly inferior to CF, and a major inefficiency is resulted from the number of candidate buckets. IVCF outperforms DVCF when $r > 0.8$ because DVCF needs an additional judgment before each lookup. Furthermore, comparing Fig. 6(a) and Fig. 6(b), those negative lookups consume more time as they need to access more candidate buckets. All VCFs perform better than DCF in lookup since indexing the candidate buckets in DCF is more complicated.

Insertion time consumption. To better analyze the results in Fig. 7, we use E_0 to represent the average times of evicted operations. Fig. 8 shows that E_0 of VCFs drops sharply as r increases. This phenomenon means that the VCF algorithm can significantly reduce the kick-out and reinsertion operations when r grows. For example, the E_0 of VCF is about 1.27 while this value of CF achieves 12.8, and these numbers are close to analysis results in Section V-C. The E_0 of DVCF is slightly larger than that of IVCF, reaching almost 2.0 when r is larger than 0.7. This result shows that the cuckoo hash collision caused by IVCF is less than DVCF.

Fig. 7 further illustrates the insertion performance by quantifying the insertion time. VCF can nearly cut the insertion time by half, compared to CF. The reason is that VCF reduces unnecessary fingerprint reallocation and hash computation by indexing more candidate buckets for each item. In Fig. 7(c), for each item insertion, IVCF takes about 10% less time than DVCF when $r > 0.8$. Although DCF also reduces the evictions, it consumes twice as much time as VCF due to its complex indexing algorithm.

3) False Positives: We construct a new dataset D, which is composed of 2^{20} items that are also randomly chosen from our experimental dataset. Note that the items in D are not inserted into the filters. So the ratio of the positive query results, i.e., ξ' represents the false positive rate of the filter.

Fig. 9 shows the linear relationship between ξ' and r. The false positive rate of IVCF and DVCF are similar. Both of them increase with the growth of r. The reason is that VCFs need to check more candidate buckets than CF, which increases the probability of fingerprint collisions.

4) Impact of Parameters in VCF: We conduct extensive experiments to quantify the impact of parameters in VCFs. We first compare the total insertion time consumed by these filters with different hash functions: FNV hash, MurmurHash[4], and DJBHash[5]. Setting r of IVCF and DVCF to the maximum, the experimental results are summarized in Table IV.

Table IV shows that our VCF methods outperform the standard CF in terms of insertion time even when different hash functions are employed. Compared with CF, the insertion time is reduced by half when using the FNV hash and DJBHash. However, this advantage is somehow degraded when using

[4]http://blog.teamleadnet.com/2012/08/murmurhash3-ultra-fast-hash-algorithm.html

[5]http://www.cse.yorku.ca/ oz/hash.html

Table IV
TIME OVERHEAD OF VCFS WITH DIFFERENT HASH FUNCTIONS.

	CF	IVCF	DVCF
FNV hash	44.6s	26.7s	26.1s
MurmurHash	67.7s	48.2s	48.5s
DJBHash	38.2s	20.1s	19.6s

Table V
THE COMPARISON OF k-VCFS.

k	2	4	5	6	7	8	9	10
α (%)	88.7	94.0	95.1	95.7	96.3	96.7	96.9	96.8
time (s)	17.7	18.7	20.4	23.0	20.4	22.4	24.0	22.0

MurmurHash. This phenomenon implies that different hash functions lead to diverse performance for VCFs.

We also compare the load factor and the total insertion time consumed by k-VCFs with different k. Specifically, we first fix the fingerprint length as 16 bits, set the reallocations threshold as 0, and vary k from 2 to 10. Table V shows the experimental results of space utilization and total insertion time consumption of $k-$VCFs. Even without reallocations, the load factor of $k-$VCF achieves nearly 97% when $k \geq 9$. The result also shows that a larger k can help $k-$VCF to achieve a higher load factor at the cost of longer time consumption.

VII. CONCLUSION

This paper reports VCF, a novel variant of Cuckoo Filter, targeting insertion-intensive scenarios such as online applications. VCF provides more candidate buckets to each item by utilizing vertical hashing. The Vertical Cuckoo Filter improves the standard Cuckoo filter in three ways: 1) higher load factors and less space overhead; 2) better insertion performance; 3) more flexible trade-off strategies to fit the dynamic requirement. Besides, VCF provides a methodology using one hash function for many sketches rather than independent hash functions. With the accurate representation and constant-time lookup/insertion features inherited from CF, VCF improves insertion throughput for frequent insertion and deletion.

VIII. ACKNOWLEDGEMENT

The authors thank all the anonymous reviewers for their insightful feedback. Besides, this work was supported in part by the National Key Research and Development Program of China under grant 2018YFB1800203, in part by the National Natural Science Foundation of China under Grants 62002378 and 61772544, and in part by the Research Funding of NUDT under Grant ZK20-30.

REFERENCES

[1] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Commun. Surv. Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[2] X. Wang, Y. Liu, Z. Yang, K. Lu, and J. Luo, "Robust component-based localizationin sparse networks," *IEEE Trans. Parallel Distributed Syst.*, vol. 25, no. 5, pp. 1317–1327, 2014.

[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[4] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," in *Proc. of ACM SIGCOMM, August 31 - September 4, 1998, Vancouver, B.C., Canada*, G. Neufeld, G. S. Delp, J. Smith, and M. Steenstrup, Eds., pp. 254–265.

[5] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *Proc. of ESA, September 11-13, 2006, Zurich, Switzerland*, vol. 4168, pp. 684–695.

[6] 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," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1627–1637, 2012.

[7] G. Einziger and R. Friedman, "Tinyset - an access efficient self adjusting bloom filter construction," *IEEE/ACM Trans. Netw.*, vol. 25, no. 4, pp. 2295–2307, 2017.

[8] 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 ICNP, Orlando, Florida, USA, October 19-22, 2008*, pp. 73–82.

[9] B. Fan, D. G. Andersen, M. Kaminsky, and M. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proc. of ACM, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*, A. Seneviratne, C. Diot, J. Kurose, A. Chaintreau, and L. Rizzo, Eds., pp. 75–88.

[10] M. Mitzenmacher, S. Pontarelli, and P. Reviriego, "Adaptive cuckoo filters," in *Proc. of ALENEX, New Orleans, LA, USA, January 7-8, 2018*, R. Pagh and S. Venkatasubramanian, Eds., pp. 36–47.

[11] L. Luo, D. Guo, O. Rottenstreich, R. T. B. Ma, X. Luo, and B. Ren, "The consistent cuckoo filter," in *Proc. of IEEE INFOCOM, Paris, France, April 29 - May 2, 2019*, pp. 712–720.

[12] H. Chen, L. Liao, H. Jin, and J. Wu, "The dynamic cuckoo filter," in *Proc. of IEEE, ICNP, Toronto, ON, Canada, October 10-13, 2017*, pp. 1–10.

[13] D. Eppstein, "Cuckoo filter: Simplification and analysis," in *Proc. of SWAT, June 22-24, 2016, Reykjavik, Iceland*, ser. LIPIcs, R. Pagh, Ed., vol. 53, pp. 8:1–8:12.

[14] M. Wang, M. Zhou, S. Shi, and C. Qian, "Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters," *VLDB Endow.*, vol. 13, no. 2, pp. 197–210, 2019.

[15] A. D. Breslow and N. Jayasena, "Morton filters: fast, compressed sparse cuckoo filters," *VLDB J.*, vol. 29, no. 2-3, pp. 731–754, 2020.

[16] F. Wang, H. Chen, L. Liao, F. Zhang, and H. Jin, "The power of better choice: Reducing relocations in cuckoo filter," in *Proc. of IEEE ICDCS, Dallas, TX, USA, July 7-10, 2019*, pp. 358–367.

[17] 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, Shenzhen, China, December 15-17, 2017*, pp. 190–197.

[18] B. K. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H. Du, "Bloomflash: Bloom filter on flash-based storage," in *Proc. of ICDCS , Minneapolis, Minnesota, USA, June 20-24, 2011*, pp. 635–644.

[19] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing bloom filter: Challenges, solutions, and comparisons," *IEEE Commun. Surv. Tutorials*, vol. 21, no. 2, pp. 1912–1949, 2019.

[20] A. Z. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *Proc. of IEEE INFOCOM, Alaska, USA, April 22-26, 2001*, pp. 1454–1463.

[21] Y. Lu, B. Prabhakar, and F. Bonomi, "Perfect hashing for network applications," in *Proc. of IEEE ISIT, The Westin Seattle, Seattle, Washington, USA, July 9-14, 2006*, pp. 2774–2778.

[22] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *Proc. of USENIX NSDI, Lombard, IL, USA, April 2-5, 2013*, N. Feamster and J. C. Mogul, Eds., pp. 371–384.

[23] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[24] A. Shrivastava, A. C. König, and M. Bilenko, "Time adaptive sketches (ada-sketches) for summarizing data streams," in *Proc. of SIGMOD, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds., pp. 1417–1432.

[25] P. Zuo, Y. Hua, and J. Wu, "Level hashing: A high-performance and flexible-resizing persistent hashing index structure," *ACM Trans. Storage*, vol. 15, no. 2, pp. 13:1–13:30, 2019.

[26] Y. Sun, Y. Hua, Z. Chen, and Y. Guo, "Mitigating asymmetric read and write costs in cuckoo hashing for storage systems," in *Proc. of USENIX ATC, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds., pp. 329–344.