CPCF: A Flexible Chunking and Proactive Insertion Cuckoo Filter

Wendi Hua[1,3], Ping Xie[1(✉)], Xiao Qin[2], Rui Mao[3], Yan Ji[3], Hao Chen[3], Hua Yan[3], and Jianbin Qin[3]

[1] College of Computer, Qinghai Normal University, Xining 810016, P.R. China
huawendi2023@email.szu.edu.cn, xieping@qhnu.edu.cn
[2] Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849-534, USA
xqin@auburn.edu
[3] College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, P.R. China
{mao,jy197541,haochen,yanhua,qinjianbin}@szu.edu.cn

Abstract. Cuckoo filter is a data structure for approximate membership queries widely used in various data science fields. However, inefficient space usage and element insertion prevent cuckoo filters from completely replacing Bloom filters. We present CPCF, a new and efficient version of cuckoo filter, which improves space utilization and insertion speed without any sacrifice. CPCF employs flexible chunking to optimize space efficiency. It automatically adjusts chunk sizes to the number of elements while minimizing granularity. A proactive insertion strategy accelerates insertion with reduced moving hash conflict elements. CPCF also astutely detects hashing failure, enhancing insertion stability. Experiments show that CPCF conserves more space than the state-of-the-art cuckoo filter variant in most cases. Additionally, CPCF augments insertion throughput by 21%∼101% under maximum load compared with other variants. The dynamic thresholds ensure accurate judgment of hashing failures at lower values. These optimizations render CPCF a versatile and high-performance approximate membership query filter.

Keywords: approximate membership query · cuckoo filters · space efficiency · proactive insertion · endless loop

1 Introduction

Approximate Membership Query(AMQ) probabilistically ascertains if an element X belongs to a set $\mathcal{S}$. AMQ-filters, a specialized class of probabilistic data structures, are designed to efficiently answer such queries. They substantially reduce storage overhead at the cost of a few false positives, where non-members are incorrectly deemed members. Their efficiency has led to adoption in big data scenarios like large-scale databases [10] and distributed systems [19]. AMQ-filters curtail network requests and I/O operations when searching for non-existent

M. Onizuka et al. (Eds.): DASFAA 2024, LNCS 14850, pp. 283–298, 2024.
https://doi.org/10.1007/978-981-97-5552-3_19

elements. Cuckoo filter [6], which stores element fingerprints by cuckoo hashing [20], is gaining traction. Its high load factor, the ratio of stored elements to all slots in the filter, either lessens space requirement or reduces query false positives. Coupled with supporting element deletion and fast operation, cuckoo filters have begun to cooperate with and gradually supersede other AMQ-filters in SlimDB [22], Redis [14] and others [2,11]. However, cuckoo filter's theoretical advantages usually misalign with real-world situations of filter size and element insertion.

Space Efficiency. Hash-based data structures require independence between hash values and address ranges. In practice, hash ranges are much bigger than address ranges, necessitating the modulo operation. This is computationally expensive with large numbers. So cuckoo filters use power-of-two slot counts, replacing modulo with fast bitwise AND. However, the footprint cannot be flexibly sized in proportion to the element count n and is restricted to powers of two. Despite the theoretical minimum of $(1+\varepsilon)\,n$ slots, where ε is a small parameter greater than 0. The actual space usage often greatly exceeds this, causing waste.

Insertion Latency. Cuckoo hashing utilizes dual mapping positions per element to mitigate collisions. On insertion, (i)the element tries to occupy one of its two mapping positions. Any free position immediately completes insertion. If both are occupied, a *relocation* process begins. (ii)The inserted element displaces and kicks out one of the elements in its intended positions. (iii)The kicked element becomes the new element to insert, and then attempts to save to its the other mapping that is different from the current position. If there are still placeholders, repeat (ii) and (iii) recursively until an empty slot is found. Although expected $O\,(1)$ insertion time, this amortizes all kick-outs across the entire set and only provides polylogarithmic [27] guarantees. Table 1 records the inserting information of four cuckoo filter schemes. Our CPCF significantly reduces the average kick-out per relocation and also has minimal slot checks with a lower cache miss rate.

Table 1. Inserting information of four cuckoo filter schemes.

$2^{12} \times 0.95$ Elements Inserted	Cuckoo Filter (CoNEXT14 [6])	Better Choice Cuckoo Filter (ICDCS19 [28])	Vacuum Filter (VLDB19 [29])	CPCF
Avg. Kick-out in One Relocation	8.53	7.00	3.31	2.05
# Slot Checking	29879	50146	41614	26223
Rt. Cache Miss	18.2%	20.5%	20.3%	15.4%

Endless Loops. In sometimes relocation, an evicted element displaces another element, starting a cycle where elements repeatedly kick out each other. Any new insertion is prevented in this *endless loop* [3], leading to the filter restructure.

Therefore, it is especially significant to judge the endless loop, as a delay in judgment will slow down the overall speed of inserting elements. Prior work on space efficiency uses manual chunking [15,29]. Combining several fine-grained chunks can better fit the space required. But they can not perform well in all situations, and even impact the load factor. To optimize insertion, various approaches have been explored by load-balanced insertion [28], better kick-out [24] and monitoring insertion [25]. However, these need extra statistical comparison or space, compromising the compactness motivating AMQ-filters. Endless loop detection currently relies on a large fixed timeout threshold for kick-out [20] repetitions in one relocation. When the number of kick-outs in a relocation reaches this threshold, it is assumed to have entered an endless loop. However, valid relocations may require a lot of kick-outs at high loads before finding a suitable position. The fixed threshold becomes useless, incorrectly halting insertions and impacting load factors.

In this paper, we present a brand-new **F**lexible chunking and **P**roactive insertion **C**uckoo **F**ilter(*CPCF*, for short). CPCF introduces flexible automated chunking for high space efficiency. While ensuring that the max load factors of the entire filter and each chunk are smaller than theoretical analysis results, CPCF minimizes the granularity of chunking. The timing of relocation about filter occupancy is first considered to optimize insertion. Reserving a proportion of empty slots at the end of the first mapping bucket makes a less kick-out insertion. With reversely finding free slots, CPCF further accelerates relocation. CPCF also dynamically adjusts kick-out thresholds, effectively identifying endless loops. And the good data locality brought by chunking enhances the speed of lookup and delete operations. Importantly, all improvements of CPCF do not trade off query false positives. In summary, the contributions made in this study are tabulated as follows.

1. Introduce the flexible chunking method, which automatically adjusts the chunk with the smallest granularity, which ensures high space efficiency.
2. Proposal of a proactive insertion based on the relationship between relocation and filter occupancy. With reverse detecting empty slots, kick-out and slot checking during relocation are greatly reduced.
3. Introduction of dynamic kick-out thresholds, which effectively reduces the delay in identifying endless loops.
4. We implement CPCF[1] and evaluate its performance improvement compared with several cuckoo filter variants including the state-of-the-art.

2 Preliminary and Related Work

2.1 Standard Cuckoo Hashing and Filter

Cuckoo hashing [20] places n elements into two hash tables T_0 and T_1, each with $m = (1 + \varepsilon)\, n$ buckets. Every bucket can accommodate one item (i.e. bucket is

[1] https://github.com/huawendi/CPCF.

equivalent to slot). Two hash functions $H_\sigma : \mathcal{S} \to T_\sigma, \sigma \in \{0, 1\}$ map elements to indexed i_0, i_1 in two tables. Elements are inserted into $T_0[i_0]$ or $T_1[i_1]$ either directly or via relocations. For query elements, return the logical OR of the values in two mapping buckets and the lookup element [23]. Deletions negate the bucket content after finding the elements. Two mappings ensure constant worst-case lookup and delete by eliminating overflow collisions.

Cuckoo filter(CF) [6] stores fingerprints instead of full elements. During operating, element X itself cannot be obtained from its fingerprint f_X. The *partial-key cuckoo hashing* from the current position $H_0(X)$ and fingerprint determines the alternate mapping $H_1(X)$:

$$\begin{aligned} H_0(X) &= hash(X) \\ H_1(X) &= H_0(X) \oplus hash(f_X). \end{aligned} \tag{1}$$

As an AMQ-filter, since the fingerprints are compression of elements, fingerprint collisions lead to false positives in CF queries. If the fingerprint length is fp bits, the probability of fingerprint collision arising in a slot is $\frac{1}{2^{fp}}$. The number of checking slots required to locate an element conforms to the discrete uniform distribution of $U(0, db\alpha)$, where there are d mapping positions for each element, b represents the number of slots in a bucket, and α means the load factor. So the asymptotic false positive rate(FPR) of CF is

$$1 - \left(1 - \frac{1}{2^{fp}}\right)^{\frac{0+db\alpha}{2}} \approx \frac{db\alpha}{2^{fp+1}}. \tag{2}$$

The traditional cuckoo hash with $d = 2$ and $b = 1$ has a very undesirable load factor. If $m \geq (1+\varepsilon)n$ the failure rate is of order $\frac{1}{m}$ when the load factor is below 0.5, but increases to approximately 18.4% for half-full table [5].

2.2 Choices of d and b

Increasing d and/or b can directly improve load. Sufficiently large d or b just provides expected $O(1)$ insertion [8]. D-ary cuckoo hashing [7] use d hash functions to correspond one-to-one in d hash tables each of $(1+\varepsilon)n$ buckets. Each element has $d > 2$ mapping positions, making relocation less likely to occur and increasing the maximum load. There has been existing work that applies d-ary cuckoo hashing to filter(i.e. d-ary cuckoo filter [30]). It can save 1 bit for each element, reducing the query FPR. But partial-key cuckoo hashing complicating in d-ary cuckoo filter, the actual operational performance is reduced exponentially.

Blocked cuckoo hashing [4] uses only one hash table consisting of $m = \frac{(1+\varepsilon)n}{b}$ buckets, each of b slots. The effect is the same as d-ary, maximum load factor increase to $\frac{1}{1+\varepsilon}$ [16]. But large b extends the lookup time to $O(2b)$, so b should be set as a small constant. Taking into consideration various performance trade-offs and mainstream implementation, cuckoo filter studied in this paper is based on the 2-choice blocked cuckoo hashing.

2.3 Relocation and Kick-Out

Standard relocation strategies are random walk depth-first search(RWDFS) and breadth-first search(BFS) for kicking out elements [9]. RWDFS kicks out elements by randomly selecting an element from $d \times b$ slots in all d-ary mapping buckets. BFS, on the other hand, goes through all the elements in the range simultaneously to find a possible empty slot, and then kicks out elements along the finding road when it is found. In practice, BFS performs poorly [13]. The recursive implementation stresses system stack, while non-recursive requires extra linear space [8].

Reducing relocations and kick-outs is the most useful way to accelerate insertion. MinCounter [24] uses counters to record how often each slot is displaced, and selects the smallest to kick when relocating. SmartCuckoo [25] employs pseudo-forest theory to monitor insertion and turns the identification of endless loops into determining the "maximal directed subgraphs". However, all of these solutions take up additional space to an extent. Optimizations with added space defeat the motivation of AMQ-filter for compacting elements. In our solution, we do not want to compromise on any performance but aspire to make a versatile cuckoo filter.

2.4 Vacuum Filter

Vacuum filter(VF) [29] balances data locality and load factor by multiple alternate ranges(ARs). Chunking with ARs makes VF the best space efficiency improvement solution as far as we know. For relocation, VF combines both DFS and BFS to allow evicted elements to find an empty slot quickly. However, VF needs to manually set ARs, and chunk granularity is twice the maximum AR. CPCF automatically chunks according to the smallest possible granularity. And VF still has not changed the mass of element relocations.

2.5 Better Choice Cuckoo Filter

Better choice cuckoo filter(BCF) [28] inserts elements into the bucket that holds the fewest items among two mapping buckets, inspired by "the power of two choices" [17]. Although it has been proven that the BCF's strategy is better than the classic strategy, blindly counting elements in two mappings causes too many bucket accesses. CPCF innovatively considers the relationship between the timing of relocation and the filter load. Elements can be evenly inserted without bucket statistics.

2.6 Other Variants of Cuckoo Filter

The conditional cuckoo filter [26] enables cuckoo filters to handle insertion of duplicate elements using a novel chaining technique. It allows for set membership query given predicates on a pre-computed sketch. The logarithmic dynamic cuckoo filter [31] reduces the worst insertion and lookup time by using a multi-level tree structure. However, it only changes the linear structure of the dynamic cuckoo filter, but does not go into the operation strategy.

3 Design and Technology of CPCF

In most practical, 2-choice hashing gives a maximum load of 6 [18]. So examples for designs of CPCF are arranged as $b = 4$, which offers good results in terms of trading off. All notations used in this paper are summarized in Table. 2.

Table 2. Summary of Notations

Notation	Description
α	load factor ($0 \leq \alpha \leq 1$)
$\mathcal{A}$	theoretical maximum load factor (when $b = 4$, $\mathcal{A} = 0.96$)
a	formula coefficient in the "Balls in Bins" conclusion
b	number of fingerprints per bucket
B	number of buckets per sub-filter of chunking result
c	number of buckets per sub-filter while trying to chunk
C	number of sub-filter of chunking result
d	number of mapping buckets per element
f	fingerprint of the element
fp	fingerprint length in bits
$H_i(X)$	calculate index mapping of the element X
i_0, i_1	indexes of two mapping buckets for an element
k	number of hash functions
m	number of buckets in the whole filter
n	number of elements in the filter/sub-filter
p	number of element reserved in proactive insertion strategy
u	number of sub-filter while trying to chunk
$L \sim Z$	example elements to be manipulated

3.1 Flexible Chunking

The use of hash tables to store fingerprints of entries in conjunction with a well-performing hash strategy gives CFs a better FPR with the same space. However, the space wastage caused by the special overall bucket setting of integer powers of two affects the real-world application of CFs, keeping them more in theory. To shave waste, a mainstream and productive approach is to chunk table and filter with some small granularities. Chunking combines small power-of-two granularities, such as 256 [15] and 512 [1], to better fit the required buckets. But manual chunk sizes lack flexibility and can reduce the load factors. VF proposes multiple alternate ranges and proportionally maps elements to one of ARs. The distance of the two mapping positions of each element must be less than its AR. However, VF needs twice the maximum AR as the granularity size to guarantee a high load factor. And these chunking sizes remain artificially malleable.

In contrast, CPCF flexibly chunks to minimize granularity according to the number of elements and the target load factor. Every chunk size is the same power of two, which is treated as an independent sub-filter. When an operation is undertaken, an element is initially mapped to a sub-filter. But the mappings per chunk vary in practice. Once the load factor of a sub-filter exceeds the maximum causing an endless loop, the entire filter will be unable to insert anymore. So we must limit maximum mappings to be less than the number of elements at the maximum load factor. We leverage the analysis results [21] of "Balls in Bins" problem for maximum mappings. Compare elements to balls and sub-filters to bins, extracting conclusions that fit the situation to give:

$$SubFilterMaxLoad\,(n, C) = \frac{n}{C} + a\sqrt{\frac{2n}{C} log C}, \tag{3}$$

Here, n elements independently and uniformly at random mapping C sub-filters. $a > 1$ is stipulated in the original conclusion. But in actual chunking the filter, a sometimes equal to 1 can also ensure the safety of sub-filters. Table 3 provides the appropriate B and the corresponding value of a in Eq. 3 when n are less then 2^{30}. When the elements are less than 2^{23}, $a = 1$ suffices. And when it is less than 2^{15}, because there are too few elements, the filter is chunked into only one sub-filter. Algorithm 1 gives the chunking process. First, calculate the total number of buckets required for all elements. Then, the chunk size starts at 1 and doubles until it reaches the total. For each size, we check if the maximum partitioned elements meet setting load factor requirements. Once met, we set the chunk data and repeat chunking within sub-filters until no more fine-grained chunks can be split. Finally, return the number of chunks rounded up and the chunk size that is still a small power of two. Since the attempted chunking granularity increases exponentially, the chunking process takes polylogarithmic time. Figure 1 delineates sub-filters and buckets per sub-filter after chunking when n is the power of two from 2^{19} to 2^{26} with the 96% max load factor. Sub-filters increase preferentially with n, only expanding when the granularity is insufficient. This maximizes space efficiency with the finest possible chunks.

Table 3. The appropriate a and B under different n.

n	B	a
$< 2^{15}$	/	1.0
$2^{15} \sim 2^{23}$	4096	1.0
$2^{23} \sim 2^{30}$	8192	1.5

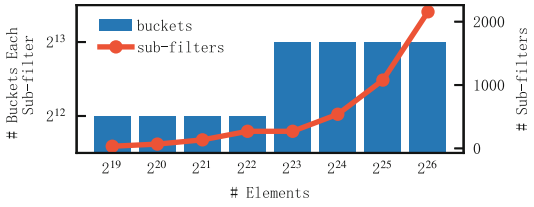


Fig. 1. Number of sub-filters and buckets in each sub-filter after chunking

With flexible chunking, the total buckets are no longer one large but multiple small powers of two. So the bitwise AND cannot map elements to sub-filters.

Algorithm 1: Flexible chunking

Input: total number of elements n, target load factor α
Output: the number of sub-filters C and buckets B in each sub-filter

$B \leftarrow \frac{n}{b \cdot \alpha}$, $C \leftarrow 1$;
do
  for $c \leftarrow 1$; $c \leq B$; $c \leftarrow c \ll 1$ **do**
    $u \leftarrow \frac{B}{c}$;
    if $SubFilterMaxLoad\,(n, u) \leq \mathcal{A} \cdot u \cdot b$ **then**
      $n \leftarrow \frac{n}{u}$, $C \leftarrow C \cdot u$, $B \leftarrow c$; `/* Save data of chunking */`
      break;
    end
  end
while $u \geq 2$;
$C \leftarrow \lceil C \rceil$, $B \leftarrow$ the smallest power of two greater than or equal to B;
return C and B;

However, fast random mapping is still possible using bitwise SHIFT(≫) [12]. If integer x is randomly in $\left[0, 2^l\right)$, then $(x \times s) \gg l$ maps x to $[0, s)$ randomly. When getting H_0 from Eq. 1 through hashing the *int* return value on a 64-bit machine, we can directly map an element to a bucket in the sub-filter using $(H_0 \times B \times C) \gg 32$. Since there are power-of-two buckets in a sub-filter, we could still depend on bitwise AND to realize the partial-key cuckoo hashing.

3.2 Proactive Insertion

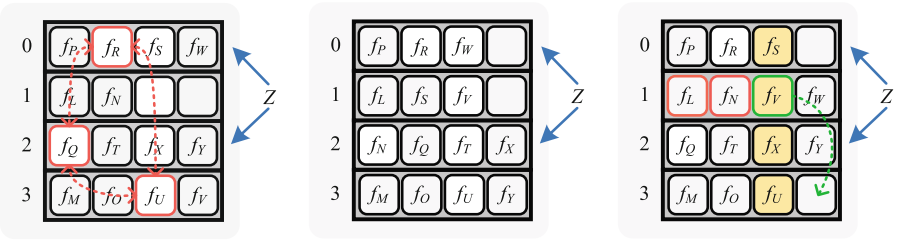
Standard CF's insertion is a greedy process of finding an empty slot and using the RWDFS to relocate elements. More often than not, this procedure is inefficient and unstable: the cause is rooted deep in the following three factors.

1. Greedy method quickly fills buckets in the early stage. Late-stage relocations have very limited options, with most buckets full.
2. During the relocation, RWDFS usually misses empty slots that are close at hand, accessing more memory to check buckets.
3. Classic insertion cannot quickly determine if elements can be inserted or encounter endless loops, waiting on the large threshold to decide.

BCF inserts each element by preferring emptier buckets of two mappings. Elements tend to be evenly inserted into the filter. Figure 2 compares standard, better choice, and proactive insertion on the same elements in alphabetical order. After inserting elements L to Y, CF leaves two empty slots in *bucket* [1]. BCF distributes emptiness across between *bucket* [0] and *bucket* [2]. For final element Z, CF triggers a relocation while BCF inserts Z into *bucket* [0] directly. Nevertheless, evenly inserting each element is redundant: an excessive number of comparisons incur hefty bucket-access costs. For example of N, counting both *bucket* [1] and *bucket* [2] is unnecessary since L is the only element in *bucket* [1]. No matter which bucket is inserted, it will have little impact on the final relocation. But for element V, the idea adopted by BCF when selecting *bucket* [1] for

insertion is very pivotal. Inserting into *bucket* [3] will fill cram the bucket. If an element is kicked out to *bucket* [3], it will have to continue relocation.

element	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
i_0	1	3	1	3	0	2	0	0	2	3	3	0	2	3	2
i_1	2	0	2	1	1	3	2	1	1	0	1	1	3	2	0



(a) Standard Insertion (b) Better Choice Insertion (c) Proactive Insertion

Fig. 2. Comparision in standard, better choice and proactive insertion strategy.

We observe that the vast majority of relocations occur when bucket occupancy is greater than 60%. So the proactive insertion is proposed in CPCF. Proactive insertion provides even insertion like BCF and faster subsequent relocation by reserving later slots. The whole process does not require needless per-element bucket statistics. The details of proactive insertion are in Algorithm 2. On insertion, the first $b-p$ slots in the primary mapping bucket are checked for availability with p slot(s) reserved. Otherwise, the alternate bucket is checked fully by no more reserving. Figure 2c shows the proactive insertion. When $b=4$ and $p=1$, each element checks the first 3 slots and serves the last slot in its first mapping bucket. Elements $L \sim U$ are the same as classic insertion. V cannot be inserted in *bucket* [3] that already accommodates three elements, and takes the penultimate slot in *bucket* [1] just like BCF. Element W stores in the second mapping *bucket* [1]. Finally, proactive insertion avoids relocation of Z with only 19 bucket accesses, while BCF does 30.

For it failed to nail down an appropriate slot, CPCF also improves relocation itself. VF alternates RWDFS and BFS. When relocating, VF uses one step BFS to check if the other mapping bucket can be per element in two full buckets can be inserted. If such an element exists, this element will be immediately evicted. And relocation is to end after the kick-out. Only when all the other mapping buckets have no empty slot, one of the elements is kicked by RWDFS. CPCF enhances this by searching empty slots back-to-front in buckets. Because proactive insertions concentrate empty slots at the front of buckets. Reverse checking finds empty slots faster with fewer checks and fingerprint decodings. Also given as an example in Fig. 2c. If two mapping buckets of element Z become *bucket* [2] and *bucket* [1] in sequence, a relocation is required. The other mapping buckets of all elements in *bucket* [1] will be checked from back to front to search

for an empty slot. The other mapping of L and N are both *bucket* [2] that is already full. When checking the other mapping *bucket* [3] of V from back to front, it can find the empty slot reserved when inserting V. Therefore, V is directly kicked out to the 4th slot of *bucket* [3], and Z is saved in the original slot of V.

Algorithm 2: Proactive insertion

Input: element's first mapping bucket index i_0 and fingerprint f
Output: whether the insertion is successful

```
if first b − p slot(s) of bucket[i0] have an empty slot then
    write f into the empty slot;
    return successful insertion;
end
calculate the other candidate bucket index i1;
if all slots of bucket[i1] have an empty slot then
    write f into the empty slot;
    return successful insertion;
end
return failure insertion;
```

3.3 Dynamic Kick-Out Threshold

As more and more elements are inserted, positional relationships among elements increasingly complex. There is already a potential endless loop as shown in Fig. 2a. When inserting element Z into *bucket* [2] and *bucket* [0], both buckets are full. In such cases, element Q is selected to make space for Z. Q, in turn, selects element U in *bucket* [3], triggering a chain of displacements. U further kick out element R and R kick out Z that is already in *bucket* [2]. This sequence lead to Z further kicks U from *bucket* [0] back to *bucket* [3]. continuing to execute, the state will return to that when Z was not inserted. It goes on endlessly and never be able to find an empty slot for the kick-out element. Presently, fixed thresholds halt relocations after sufficient kick-outs to identify endless loops. In the practical implementation of CF, the threshold is set to a fixed value of 500. When an endless loop occurs, the excessively large threshold will seriously slow down the insertion throughput and become invalid as elements increase. CPCF dynamizes kick-out thresholds according to the number of elements. We derive dynamic thresholds by analyzing BFS relocation, where checked buckets grow exponentially with kicks. Let $\mathcal{T}$ denote the maximum kicks before declaring a failure insertion. Then $\sum_{i=0}^{\mathcal{T}} b^i = 1 + b + b^2 + \cdots + b^{\mathcal{T}} \geq B$, so $\mathcal{T} \geq log_b (B \cdot b - B + 1)$. Since BFS we wield is one step, the threshold is configured to $\mu \cdot \mathcal{T}$ where μ varies with the number of elements. To amortize the randomness, we let $\mu = d \cdot b \cdot logC$. Therefore, the dynamic kick-out thresholds are

$$MaxKick = \lceil d \cdot b \cdot logC \cdot log_b (B \cdot b - B + 1) \rceil . \tag{4}$$

As with fixed thresholds, $MaxKick$ limits the number of kick-outs in one relocation nesting a *for*-loop. It maintains effectiveness even with more elements and higher loads.

4 Experimental Evaluation

We implement the proposed CPCF with $p = 1$ in C++ and evaluate it against:

- Cuckoo Filter(CF): We test the traditional cuckoo filter as the baseline solution[2].
- Better Choice Cuckoo Filter(BCF): We evaluate Feiyue's implementation of the BCF scheme[3], which adopts the function already in CF to count the elements of each bucket.
- Vacuum Filter(VF): We select VF - an extended and fast version of CF[4] in the experiments. For the ARs in VF, we choose the default settings(32, 64 and more) of the implementation.

All experiments are run on Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz with 24MB L3 cache and 128GB RAM. The Linux kernel version is 5.4.0-155-generic. Compilation uses GUN G++ 9.4.0 with *-O3 -march=native*. We consistently set the parameters to $d = 2$, $b = 4$ and $fp = 12$ bits. Each operating element employs CityHash[5] to generate a 64-bit hash value. The high 32 bits are divided for the element's first mapping index, whereas the low 32 bits for calculating fingerprint. Every point in the resulting figure is an average of 10 runs.

4.1 Space Efficiency

Figure 3 reveals the total buckets when $n = 2^{19} \sim 2^{20}$, in powers of two. VF and CPCF can greatly slash wasted space versus CF. However, the coexistence of ARs in VF cannot achieve exactly 95% load factor. Doubling the max range also produces larger chunk sizes than CPCF. With minimum granularity, CPCF gets closest to the ideal 95% load factor curve in most cases. In the few remaining cases, it is also the same as the state-of-the-art VF.

4.2 Throughput with Filter Occupancy

In this group of experiments, we test the throughput varies with filter occupancy. All filters are created by $2^{27} \times 0.95$ elements. The space usage is 192MB which is much larger than L3 cache.

Insert. Figure 4a presents the experimental results of the insert operation. With low occupancy, there are very few elements requiring relocation. The filters only need to check one or two mapping buckets for each element. VF has to calculate the AR of each element, so its throughput is slightly inferior to CF. BCF performs poorly due to redundant count comparisons. As the occupancy increases, there

[2] https://github.com/efficient/cuckoofilter.
[3] https://github.com/CGCL-codes/BCF.
[4] https://github.com/wuwuz/Vacuum-Filter.
[5] https://github.com/google/cityhash.

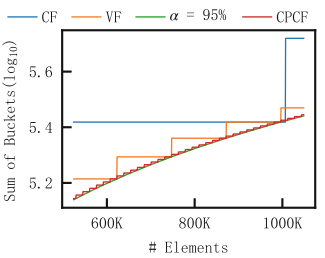


Fig. 3. Total buckets required at different numbers of elements.

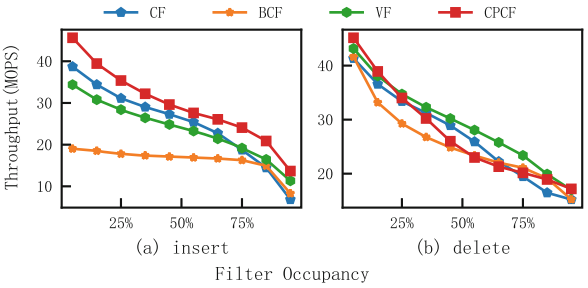


Fig. 4. Insertion and delete throughput at different occupancy.

are an increasing number of relocation cases. The best data locality of ARs and the optimized relocation make the performance of VF surpass that of CF. Too many kick-outs sharply drop the throughput of CF. BCF reflects the effect of better choice when occupancy exceeds 85%. Proactive insertion evenly inserts elements into the filter without excessive calculations and bucket statistics in the early stage. In the later relocation, free slots can be quickly found for the kicked elements. So CPCF to maintain the leading throughput across all occupancy.

Lookup. Figure 5 shows the lookup throughput of the evaluated filters under all positive, all negative and mixed (50% positive and 50% negative) elements. The throughput is greatly enhanced by the Intel Xeon micro-architecture whose multiple memory-accessing units coalesce two accesses in parallel. To avert unnecessary mapping index calculation, VF actively separates two memory accesses. So VF's throughput during positive and mixed lookups will sharply drop with the growth in occupancy. And negative search is inferior to all comparison filters. Except for VF, table occupancy has little effect on the performance of lookup operations. The throughput of CPCF is larger than those of the other filters in all situations of 4.5 MOPS or more due to the good data locality in the fine-grained sub-filter.

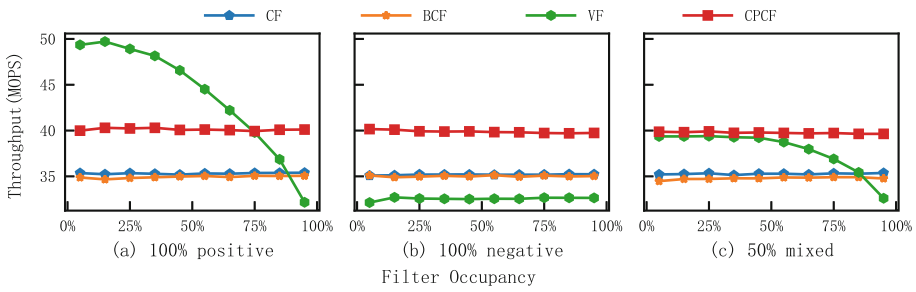


Fig. 5. Lookup throughput at different occupancy with three situations.

4.3 Other Scales for Throughput

Number of Elements. The optimization effect of each solution is best reflected under the max load factor. The load factor is set at a fixed 95% and gauges the throughput of insertion and deletion under a diverse number of elements. The number of items is 0.95 times the power of two from 2^{20} to 2^{27}. The experimental results are plotted in Fig. 6. The increase in the number of elements only reduces the throughput proportionally. The advantage of CPCF is more obvious, in which the inserting throughput of CPCF is 101%, 65%, and 21% higher than those of CF, BCF and VF, respectively. Note that the default AR setting of VF may affect the load factor. We need to run it a few more times to get 10 experimental data. For the deletion, the designs of these filters are identical. So there is little difference among the throughput of both occupancy and element count as shown in Fig. 4b and Fig. 6b.

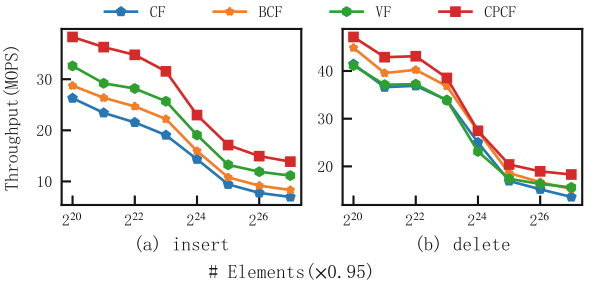


Fig. 6. Insertion and delete throughput at different numbers of elements.

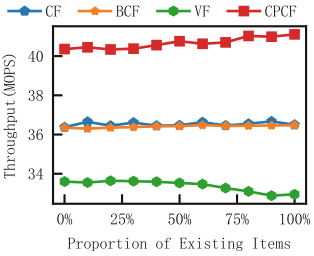


Fig. 7. Lookup throughput at different proportions of existing items.

Lookup for Different Proportions of Existing and Non-existing Elements. Figure 7 also measures lookup throughput by percent of positive elements when the number of elements is $2^{27} \times 0.95$. CPCF can still sustain 4.5 MOPS higher in all cases. And the FPR of CPCF overlaps with that of CF as Eq. 2. Suppose h elements with the same fingerprint as the lookup element in different buckets. The probability of finding these h elements by chance in CF is $\frac{dh}{m}$. For CPCF, the operations of any element are in a sub-filter. We define the random variable $\mathcal{X}$ to indicate that the number of fingerprint conflicts in the sub-filter where the lookup is located. Each fingerprint is independent of any other, so $\mathcal{X}$ follows the binomial distribution of $B\left(h, \frac{1}{C}\right)$. Hence, we have

$$P\left(\mathcal{X}=i\right)=\binom{h}{i}\left(\frac{1}{C}\right)^{i}\left(1-\frac{1}{C}\right)^{h-i}, i \in [0,h]. \tag{5}$$

Similarly, we define $\mathcal{F}$ to signify whether a false positive occurs: $\mathcal{F}=1$ means a false positive shows up. Thus, $P\left(\mathcal{F}=1|\mathcal{X}=i\right)=\frac{d \cdot i}{B}$. Given $E\left(\mathcal{X}\right)=\frac{h}{C}$, we

derive the probability of a false positive taking place as $P(\mathcal{F}=1)=\frac{d\cdot i}{B}\cdot\frac{h}{C}$. We investigate at least one fingerprint collision and let $i=1$. Under the same space size, the relationship between CF and CPCF is $m=B\times C$. The probability of finding a colliding element if CPCF is identical to the probability in CF.

4.4 Dynamic Thresholds

Figure 8 plots max kick-outs for 100 successful relocations across element counts. Until $n \leq 2^{27}\times 0.95$, dynamic thresholds in CPCF are lower than CF's fixed 500, while still achieving 98%+ accuracy. Only when the elements are 2^{18} and $2^{19}\times 0.95$, the two abnormal points are incorrectly determined to be endless loops. As the number of elements increases, the threshold will eventually exceed 500, but it still be available in use. But CF will then incorrectly detect an endless loop even though one does not exist, severely affecting the load factor.

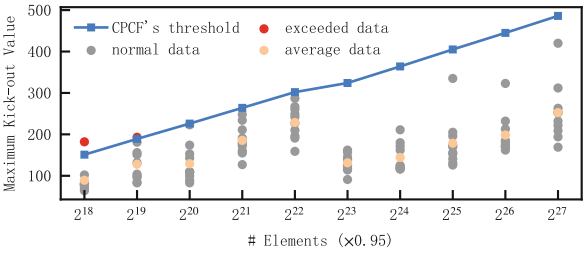


Fig. 8. Comparison of maximum kick-out in one relocation and dynamic thresholds in CPCF.

5 Conclusion

We introduce CPCF, an efficient cuckoo filter using flexible automated chunking, proactive insertion and dynamic kick-out thresholds. Space efficiency, element insertion speed and endless loop judgment have been optimized correspondingly. Experiments show CPCF saves more space versus state-of-the-art in most cases. Insertion throughput improves 21%∼101% under maximum load factor. Dynamic thresholds effectively identify endless loops with fewer kick-outs. All optimizations without sacrificing any other performance make CPCF a versatile approximate membership query filter.

Acknowledgements. This paper is supported by the National Nature Science Foundation of China(NSFC) under Grant No. 62362057 and 61762075, CCF-Tencent Rhino-Bird Open Research Fund CCF-Tencent RAGR20230126, Shenzhen University Research Instrument Development and Cultivation 2023YQ017, the Guangdong "Pearl River Talent Recruitment Program" under Grant 2019ZT08X603 and 2019JC01X235, the Foundation of Shenzhen Grant number 20220810142731001. Xiao Qin's work is supported by the National Aeronautics and Space Administration (Grant 80NSSC20M0044), the National Highway Traffic Safety Administration (Grant 451861-19158).

References

1. Alcantara, D.A., et al.: Real-time parallel hashing on the GPU. ACM Trans. Graph. **28**(5), 1–9 (2009)
2. Dayan, N., Twitto, M.: Chucky: a succinct cuckoo filter for LSM-tree. In: International Conference on Management of Data, pp. 365–378. Association for Computing Machinery, New York (2021)
3. Devroye, L., Morin, P.: Cuckoo hashing: further analysis. Inf. Process. Lett. **86**(4), 215–219 (2003)
4. Dietzfelbinger, M., Weidling, C.: Balanced allocation and dictionaries with tightly packed constant size bins. Theoret. Comput. Sci. **380**(1–2), 47–68 (2007)
5. Drmota, M., Kutzelnigg, R.: A precise analysis of cuckoo hashing. ACM Trans. Algorithms **8**(2), 1–36 (2012)
6. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: practically better than bloom. In: 10th ACM International on Conference on emerging Networking Experiments and Technologies, pp. 75–88. Association for Computing Machinery, New York (2014)
7. Fotakis, D., Pagh, R., Sanders, P., Spirakis, P.: Space efficient hash tables with worst case constant access time. Theor. Comput. Syst. **38**(2), 229–248 (2005)
8. Frieze, A., Johansson, T.: On the insertion time of random walk cuckoo hashing. Random Struct. Algorithms **54**(4), 721–729 (2019)
9. Frieze, A.M., Melsted, P., Mitzenmacher, M.: An analysis of random-walk cuckoo hashing. SIAM J. Comput. **40**(2), 291–308 (2011)
10. Hua, W., Gao, Y., Lyu, M., Xie, P.: Research on bloom filter: a survey. J. Comput. Appl. **42**(6), 1729–1747 (2022)
11. Krishna, R.S., Tekur, C., Bhashyam, R., Nannaka, V., Mukkamala, R.: Using cuckoo filters to improve performance in object store-based very large databases. In: 13th Annual Computing and Communication Workshop and Conference, pp. 0795–0800. IEEE, Las Vegas (2023)
12. Lemire, D.: Fast random integer generation in an interval. ACM Trans. Model. Comput. Simul. **29**(1), 1–12 (2019)
13. Li, D., Du, R., Liu, Z., Yang, T., Cui, B.: Multi-copy cuckoo hashing. In: 35th International Conference on Data Engineering, pp. 1226–1237. IEEE, Macao (2019)
14. Li, P., Luo, B., Zhu, W., Xu, H.: Cluster-based distributed dynamic cuckoo filter system for Redis. Int. J. Parallel Emergent Distrib. Syst. **35**(3), 340–353 (2020)
15. Maier, T., Sanders, P., Walzer, S.: Dynamic space efficient hashing. Algorithmica **81**(8), 3162–3185 (2019)
16. Minaud, B., Papamanthou, C.: Note on generalized cuckoo hashing with a stash. arXiv preprint arXiv:2010.01890 (2020)
17. Mitzenmacher, M.: The power of two choices in randomized load balancing. IEEE Trans. Parallel Distrib. Syst. **12**(10), 1094–1104 (2001)
18. Mitzenmacher, M.: Some open questions related to cuckoo hashing. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 1–10. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04128-0_1
19. Moreira, M.D.D., Laufer, R.P., Velloso, P.B., Duarte, O.C.M.: Capacity and robustness tradeoffs in bloom filters for distributed applications. IEEE Trans. Parallel Distrib. Syst. **23**(12), 2219–2230 (2012)
20. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: auf der Heide, F.M. (ed.) ESA 2001. LNCS, vol. 2161, pp. 121–133. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44676-1_10

21. Raab, M., Steger, A.: "balls" into bins— a simple and tight analysis. In: Luby, M., Rolim, J.D.P., Serna, M. (eds.) RANDOM 1998. LNCS, vol. 1518, pp. 159–170. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49543-6_13
22. Ren, K., Zheng, Q., Arulraj, J., Gibson, G.: SlimDB: a space-efficient key-value storage engine for semi-sorted data. Proc. VLDB Endow. **10**(13), 2037–2048 (2017)
23. Reviriego, P., Sánchez-Macián, A., Walzer, S., Dillinger, P.C.: Approximate membership query filters with a false positive free set. arXiv preprint arXiv:2111.06856 (2021)
24. Sun, Y., Hua, Y., Feng, D., Yang, L., Zuo, P., Cao, S.: MinCounter: an efficient cuckoo hashing scheme for cloud storage systems. In: 31st Symposium on Mass Storage Systems and Technologies, pp. 1–7. IEEE, Santa Clara (2015)
25. Sun, Y., Hua, Y., Jiang, S., Li, Q., Cao, S., Zuo, P.: SmartCuckoo: a fast and cost-efficient hashing index scheme for cloud storage systems. In: USENIX Annual Technical Conference, pp. 553–565. USENIX Association, Santa Clara (2017)
26. Ting, D., Cole, R.: Conditional cuckoo filters. In: Proceedings of the International Conference on Management of Data, pp. 1838–1850. Association for Computing Machinery, New York (2021)
27. Walzer, S.: Insertion time of random walk cuckoo hashing below the peeling threshold. In: Chechik, S., Navarro, G., Eotenberg, E., Herman, G. (eds.) ESA 2022, LIPIcs, vol. 244. Springer, Potsdam (2022). https://doi.org/10.4230/LIPIcs.ESA.2022.87
28. Wang, F., Chen, H., Liao, L., Zhang, F., Jin, H.: The power of better choice: reducing relocations in cuckoo filter. In: 39th International Conference on Distributed Computing Systems, pp. 358–367. IEEE, Dallas (2019)
29. Wang, M., Zhou, M.: Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. Proc. VLDB Endow. **13**(2), 197–210 (2019)
30. Xie, Z., Ding, W., Wang, H., Xiao, Y., Liu, Z.: D-Ary cuckoo filter: a space efficient data structure for set membership lookup. In: 23rd International Conference on Parallel and Distributed Systems, pp. 190–197. IEEE, Shenzhen (2017)
31. Zhang, F., Chen, H., Jin, H., Reviriego, P.: The logarithmic dynamic cuckoo filter. In: 37th International Conference on Data Engineering, pp. 948–959. IEEE, Chania, Greece (2021)