Support Optimality and Adaptive Cuckoo Filters

Tsvi Kopelowitz[1], Samuel McCauley[2(✉)], and Ely Porat[1]

[1] Bar-Ilan University, Ramat Gan, Israel
porately@cs.biu.ac.il
[2] Williams College, Williamstown, MA 01267, USA
srm2@williams.edu

Abstract. Filters (such as Bloom Filters) are a fundamental data structure that speed up network routing and measurement operations by storing a compressed representation of a set. Filters are very space efficient, but can make bounded one-sided errors: with tunable probability ϵ, they may report that a query element is stored in the filter when it is not. This is called a ***false positive***. Recent research has focused on designing methods for ***dynamically adapting*** filters to false positives, thereby reducing the number of false positives when some elements are queried repeatedly.

Ideally, an adaptive filter would incur a false positive with bounded probability ϵ for each new query element, and would incur $o(\epsilon)$ total false positives over all repeated queries to that element. We call such a filter ***support optimal***.

In this paper we design a new Adaptive Cuckoo Filter, and show that it is support optimal (up to additive logarithmic terms) over any n queries when storing a set of size n.

We complement these bounds with experiments that show that our data structure is effective at fixing false positives on network trace datasets, outperforming previous Adaptive Cuckoo Filters.

Finally, we investigate adversarial adaptivity, a stronger notion of adaptivity in which an adaptive adversary repeatedly queries the filter, using the result of previous queries to drive the false positive rate as high as possible. We prove a lower bound showing that a broad family of filters, including all known Adaptive Cuckoo Filters, can be forced by such an adversary to incur a large number of false positives.

1 Introduction

A ***filter*** is a data structure that supports membership queries for a set of elements $S = x_1, \ldots x_n$ from a universe U. The answer to each filter query is `present` or `absent`. Typically, a filter has a ***correctness*** guarantee: if an element $q \in S$, the filter must return `present` to the query with probability 1.

This work was supported in part by ISF grants no. 1278/16 and 1926/19, by a BSF grant no. 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).

A. Lubiw et al. (Eds.): WADS 2021, LNCS 12808, pp. 556–570, 2021.
https://doi.org/10.1007/978-3-030-83508-8_40

There is also a ***performance*** guarantee: if an element $q \notin S$, the filter must return `present` with tunable probability at most ϵ. If a query on an element $q \notin S$ returns `present` then q is called a ***false positive***. Typically, filters use a small amount of space.

A filter's small size means that the filter can be stored in an efficiently accessible location. Meanwhile, the no-false-negative guarantee implies that if the filter returns $q \notin S$ for a query q, then there is no need for accessing the actual data, which is typically stored in a medium with expensive access cost. This ability to filter out queries to items not in S in a small-size structure has found a wide variety of applications both in theory [8,11] and in practice, e.g. [5,10].

There are several different kinds of filters. The Bloom filter [4] was the first filter data structure to be designed; it is still very popular due to its simplicity and efficiency. Later filters were designed to provide better worst-case lookup times and space guarantees [2,14,17], improved practical performance [9,18], and improved cache performance [3].

In this paper, we focus on filters that achieve space very close to the optimal $n \log 1/\epsilon$ bits [6,12], and that store elements from a large universe $|U| \gg n$.

Fixing False Positives. A well-known issue with many existing filters is that they cannot ***adapt*** to queries: if a query $q \notin S$ is a false positive, all subsequent queries $q' = q$ will be false positives. The focus of this paper is designing filters that do adapt to false positive queries, so that if a query q is a false positive, the filter undergoes structural changes so that a later query to q is unlikely to be a false positive. An element q is said to be ***fixed*** if q was previously a false positive, but is no longer a false positive. Similarly, q is ***broken*** if q was previously fixed, but is now a false positive.

Related Work. Bender et al. [2] analyzed how to fix false positives against an adversary. They give a data structure such that if queries are generated by an adversary trying to maximize the false positive rate, each query to a filter is a false positive with probability at most ϵ, even if the query element was queried before. This requirement essentially provides concentration bounds: over n queries, their filter incurs ϵn false positives, even if the queries are maliciously chosen based on previous false positives.

However, the benefit of adaptivity goes beyond resisting an adversary. As shown experimentally by Mitzenmacher et al. [13], adapting to queries can significantly decrease the number of false positives—in fact, if queries are repeated sufficiently frequently, the performance can be much better than $O(\epsilon n)$. In particular, network trace data consists of a structured sequence of queries—can we give a data structure that performs particularly well on this kind of data?

Most recently, Bender et al. [1] compared adaptivity to cache-based strategies, finding that adaptivity leads to significantly better practical performance.

Support Optimality. Ideally, an adaptive filter would incur a false positive with probability ϵ for each new query, and incur no further queries asymptotically. Thus, every new false positive is fixed, and this fixing is unlikely to break

previously-fixed false positives. In particular, let $q_1, \ldots q_n$ be a predetermined sequence of queries[1] to a filter $\mathcal{F}$, and let $Q = \bigcup_{i=1}^{n} \{q_i\}$ be the set of unique queries in the sequence. We say that $\mathcal{F}$ is ***support optimal*** if the expected number of false positives when querying $q_1, \ldots q_n$ is $\epsilon|Q|(1 + o(1))$. In this paper we give a support-optimal filter up to additive polylogarithmic terms, and show that it significantly improves practical performance.

1.1 Results

We discuss three data structures in this paper: two versions of the Adaptive Cuckoo Filter originally presented in [13] (which we call the ***Cyclic ACF*** and ***Swapping ACF***), and a Cuckoo Filter augmented with a new method of achieving adaptivity, which we call the ***Cuckooing ACF***.

The first contribution of this paper is the Cuckooing ACF, a support-optimal filter which can be implemented using almost-trivial changes to current Cuckoo Filter implementations.

In Sect. 3, we analyze the Cuckooing ACF and prove that it is support optimal over any n queries, up to additive polylogarithmic terms. This gives a significant performance improvement over previous filters even for large Q, and the difference becomes more dramatic for small Q. For example, for the case of a repeated single query ($|Q| = 1$), static filters incur ϵn expected false positives, whereas the Cuckooing ACF incurs $O(\log^4 n)$ expected false positives.

We show that despite their strong practical performance, the Cyclic ACF and Swapping ACF are not support optimal–even if there are a constant number of queries ($|Q| = O(1)$), they may incur $\Omega(n)$ false positives, whereas the Cuckooing ACF incurs at most $O(\log^4 n)$. Thus, from the standpoint of support optimality, cuckooing is a better method for achieving adaptivity.

In Sect. 4, we provide experimental results that show that the theory bears out in practice: the Cuckooing ACF attains a low false positive rate on network trace datasets, which contain many repeated queries. The performance is not only stronger than a vanilla Cuckoo Filter, but also improves upon the performance of a Cyclic ACF and a Swapping ACF of the same size. This shows that the Cuckooing ACF is effective at fixing false positives in a practical sense. These results also emphasize the benefit of a simple adaptive filter: not only is the resulting data structure easier to implement, the simplicity entails less space usage compared to previous Adaptive Cuckoo Filters, leading to a significant performance improvement.

Finally, in Sect. 5, we prove lower bounds that demonstrate that a broad family of filters cannot be adaptive in the adversarial sense of Bender et al. [2]; this includes the Cyclic ACF, the Swapping ACF, and the Cuckooing ACF. This lower bound motivates the concept of support optimality: a support optimal filter achieves strong performance on real datasets without achieving adversarial

[1] Note that the filter does not have access to this sequence ahead of time; it must process the queries online.

adaptivity. Our proof also gives insight into the structure of adaptive filters—specifically, it shows that a space-efficient filter must have variable-sized fingerprints in order to be adversarially adaptive.

2 Three Adaptive Cuckoo Filters

In this section describe a new kind of filter, the Cuckooing ACF. We then discuss the Cyclic ACF and the Swapping ACF, both originally introduced in [13].

2.1 ACF Parameters and Internal State

We begin by defining a more general data structure which we call the ***adaptive cuckoo filter*** (ACF). As the name suggests, the Cyclic ACF, the Swapping ACF, and the Cuckooing ACF are adaptive cuckoo filters.

An ACF $\mathcal{F}$ has integer parameters $f, k, b > 0$, an additional parameter $\gamma > 1$, and supports storing n elements from a universe U with a false positive rate ϵ. The internal representation of a filter $\mathcal{F}$ consists of k hash tables, each of $N = \gamma n/bk$ bins,[2] where each bin consists of b slots of f bits; thus, the space usage of $\mathcal{F}$ is $N \cdot b \cdot f \cdot k$ bits. The parameter γ determines how densely elements are packed, trading off between insert time and space; often $\gamma \approx 1.05$ is used.

The hash tables are accessed using $k + 1$ hash functions: k ***location hash*** functions $h_1^\ell, \ldots, h_k^\ell : U \to \{0, \ldots, N-1\}$ that hash from U to a bit string of length $\log N$,[3] and a single ***fingerprint hash*** h^f mapping each $x \in U$ to an f-bit ***fingerprint***. The range and domain of h^f depend on which ACF is used and may depend on the internal state of $\mathcal{F}$; we provide details below. Following previous results on filters [2–4,7,9,13], this paper assumes free access to uniform random hash functions.[4]

When a set S is stored in $\mathcal{F}$, for each element $x_i \in S$, the fingerprint of x_i is stored in one of the slots of bin $B(x_i)$ in the β_ith hash table; this bin is defined using a location hash: $B(x_i) = h_{\beta_i}^\ell(x)$ for some integer $0 \le \beta_i < k$. We say a slot σ is ***occupied*** if the fingerprint of some $x_i \in S$ is stored in σ; otherwise σ is ***empty***. We call β_i the ***hash index*** of x_i.

Since an ACF stores each element using a hash index, we can keep track of the internal state of a filter using the hash index of each element. Thus, we use $C = (C[1], C[2], \ldots, C[n]) = (\beta_1, \ldots, \beta_n)$ to define the ***configuration*** of $\mathcal{F}$. This fully defines the internal representation of a Cuckooing ACF. The internal representation of a Cyclic ACF also depends on s metadata bits stored for each element, and the internal representation of a Swapping ACF also depends on which slot within the bin is used to store each element.

[2] We assume γn is an integer multiple of bk for simplicity.

[3] When treating the hash value as a bit string we assume that N is a power of two for simplicity; this assumption is not necessary for the implementation.

[4] While such strong hashes are not usable in practice, this analysis is generally predictive of experimental results (see i.e. [9,13,16]).

Suppose S is stored using hash indices $\beta_1, \ldots \beta_n$ under some configuration C. Then query $q \notin S$ ***collides*** with an element $x_i \in S$ under C when $h^\ell_{\beta_i}(x_i) = h^\ell_{\beta_i}(q)$ and q and x_i have the same fingerprint.

2.2 Cuckoo Filter Operations

We begin by describing how inserts and queries work for an ACF. The Cuckooing ACF and Swapping ACF insert and query using these methods; the Cyclic ACF uses a generalization of these methods.

Insert. Suppose an element x_i is inserted into a set S of size $i-1$ currently stored with filter $\mathcal{F}$ in configuration C, where elements $S = x_1, \ldots x_{i-1}$ have hash indices $\beta_1, \ldots \beta_{i-1}$. Assume that $\mathcal{F}$ can store up to $n \geq i$ elements. The insertion algorithm finds a valid configuration C' of $\mathcal{F}$ on S such that there exists a hash index $\beta'_i \in \{0, \ldots k-1\}$ for which bin $h^\ell_{\beta'_i}$ has an empty slot. This may involve updating the hash indices of other elements; for $1 \leq j < i$ let β'_j be the hash index of x'_j under C'. We describe how to determine C' below.

If there is already an available empty slot, the filter stores the element immediately in that slot. Specifically, if there exists a $\beta \in \{0, \ldots, k-1\}$ where bin $h^\ell_\beta(x_i)$ in hash table β contains an empty slot, the filter sets $\beta'_i = \beta$, and stores the fingerprint of x_i in the empty slot. All other slots remain unchanged: $\beta'_j = \beta_j$ for all $1 \leq j < i$.

Now, consider the case where there is no available empty slot. Then the ACF makes room by shifting elements as one would in cuckoo hashing [15]. The filter selects a hash index β_i arbitrarily from $\{0, \ldots, k-1\}$. Since all slots in bin $h^\ell_{\beta'_i}(x_i)$ are occupied in C, the filter ***moves*** the fingerprint of some element x_j stored in a slot in $h^\ell_{\beta'_i}(x_j) = h^\ell_{\beta'_i}(x_i)$, leaving an empty slot in which x_i can be stored. If $h^\ell_\beta(x_j)$ contains an empty slot for some $\beta \in \{0, \ldots, k-1\}$ (i.e. if x_j can be stored in an empty slot), one such empty slot is arbitrarily selected to store x_j. Otherwise, the filter increments $\beta'_j = \beta_j + 1 \pmod k$ and recurses, moving an element stored in $h^\ell_{\beta'_j}(x_j)$ as necessary.

The move the elements as described above, the ACF must be able to access the set S during an insert in order to rehash each x_j. We follow all past work on adaptive filters [1,2,13] in assuming that an external dictionary can be accessed, enabling an element to be rehashed while inserting or fixing.

If this recursive process takes too many steps (more than $\Theta(\log n)$ elements are moved), the filter chooses new hashes and rebuilds from scratch. If $\mathcal{F}$ uses $N = \Omega(n)$ hash slots, then over n inserts, the probability of a rebuild is $O(1/n)$ [15].

Query. On a query q, a filter $\mathcal{F}$ in configuration C returns `present` if there exists a β and a slot index $\sigma \in \{1, \ldots b\}$ such that slot σ in bin $h^\ell_\beta(q)$ of table β is occupied and stores the fingerprint of q. This immediately guarantees correctness of the filter (queries to $x_i \in S$ always return `present`) and, via a union bound over the elements of S, a false positive rate of at most $n/(N2^f)$. The filter achieves a desired false positive rate ϵ by setting $f = \log(n/(N\epsilon)) = \log(bk/\epsilon\gamma)$.

Fixing False Positives. If an ACF returns `present` on a false positive query q (the filter knows that $q \notin S$ from the external dictionary storing S), the ACF modifies its configuration to attempt to fix q, so that subsequent queries to q return `absent`. Each type of ACF has its own method for fixing false positives, which we describe below. Notice that the process of modifying the configuration may cause some query $q' \notin S$ to become a false positive, even if q' was fixed some time in the past.

2.3 Cuckooing ACF

The primary data structure contribution of this paper is the ***Cuckooing ACF***. This data structure is a standard Cuckoo Filter [9] with an added operation to fix false positives; inserts and queries work exactly as described in Sect. 2.2.

Let q be a false positive under configuration C; we define how the Cuckooing ACF finds a new configuration C' with hash indices $\beta'_1, \ldots \beta'_n$ to attempt to fix q. For each $x_i \in S$ that collides with q under C, the filter moves x_i recursively as it would during an insert. Specifically, the filter sets the new hash index $\beta'_i = \beta_i + 1 \pmod k$; if bin $h^\ell_{\beta'_i}(x_i)$ in table β'_i does not contain an empty slot, an element x_j stored in $h^\ell_{\beta'_i}(x_i)$ under C is moved recursively. If $\Omega(\log n)$ steps are taken, the filter is rebuilt. Standard cuckoo hashing analysis shows that for any false positive on a Cuckooing ACF with $\gamma = 1 + \Omega(1)$ the probability of a rebuild is $O(1/n^2)$ [15].

2.4 Cyclic ACF

The ***cyclic ACF*** of Mitzenmacher et al. [13] is an ACF where each slot contains s additional ***hash selector*** bits. The cyclic ACF generally has $b = 1$; thus, the total space used by a Cyclic ACF is $kN(f + s)$. Usually, s is a small constant.

In the Cyclic ACF, the fingerprint hash maps $U \times \{0, \ldots, 2^s - 1\} \to \{0, \ldots, 2^f - 1\}$. In particular, the hash selector bits are used to determine the fingerprint of an element stored in a given slot.

When an element x_i is initially inserted, the insertion process continues as in Sect. 2.2, with fingerprint $h^f(x_i, 0)$. When an empty slot σ is found that can store x_i, the hash selector bits of σ are set to 0, and $h^f(x_i, 0)$ is stored in σ.

To query an element q, for each location hash h^ℓ_β, with $\beta \in \{0, \ldots, k-1\}$, the filter looks at the slot h^ℓ_β of table β. The s hash selector bits stored in the slot contain a value $0 \le \alpha \le 2^s - 1$. The filter compares $h^f(q, \alpha)$ with the fingerprint stored in the slot; the filter returns `present` if they are equal. Otherwise the filter increments β and repeats. If no collisions are found for all $0 \le \beta \le k - 1$, the filter returns `absent`.

If a query q is a false positive, the Cyclic ACF fixes the query as follows. Let x_i be the element that collides with q, let σ be the slot storing x_i, and let α be the value of the s hash selector bits stored in σ. Then the filter sets the hash selector bits of σ to store value $\alpha + 1$, and stores $h^f(x_i, \alpha + 1)$ in σ. If multiple $x \in S$ collide with q, this procedure is repeated for each such x.

2.5 Swapping ACF

The idea of the Swapping ACF [13] is to have elements hash to bins with $b > 1$ slots, and to have the fingerprint of an item depend on its slot. In this way, false positives can be (potentially) fixed by moving elements to a different slot.

Inserts proceed as described in Sect. 2.2. However, in the Swapping ACF, the fingerprint hash maps $U \times \{0, \ldots, b-1\} \to \{0, \ldots, 2^f - 1\}$. During an insert, an element's slot must be determined before its fingerprint can be calculated.

If a query q is a false positive under configuration C, the filter can fix the query as follows. Let x_i be the element that collides with q and let $b(x_i) = h^\ell_{\beta_i}(x_i)$ be the bin currently storing x_i. Let $\sigma_i \in \{0, \ldots, b-1\}$ be the index of the slot in $b(x_i)$ currently storing x_i; thus x_i is stored in slot $h^\ell_{\beta_i}(x_i) \cdot b + \sigma_x$.

The filter picks a slot index $\sigma' \in \{0, \ldots, \sigma_i - 1, \sigma_i + 1, \ldots b-1\}$, selected at random from the slots in $b(x_i)$, excluding the slot currently storing x_i. Let x_j be the element currently stored in that slot if it exists. The filter then swaps the elements: it stores fingerprint $h^f(x_i, \sigma')$ in slot $h^\ell_{\beta_i}(x_i) \cdot b + \sigma'$, and fingerprint $h^f(x_j, \sigma_i)$ in slot $h^\ell_{\beta_i}(x_i) \cdot b + \sigma_i$ (if x_j does not exist, σ_i becomes unoccupied).

3 Bounding the False Positive Rate by the Number of Distinct Queries

In this section we show that the Cuckooing ACF is support optimal: it achieves strong performance against skewed datasets, where the queries are taken from a relatively small set of elements.

Our analysis focuses on a Cuckooing ACF with $k = 2$ hash tables, $b = 1$ slots per bin, and $N = n$ slots per hash table[5] (corresponding to the classic Cuckoo Hashing analysis). The experiments in Sect. 4 indicate that our analysis likely extends to broader parameter ranges. However, formally completing the analysis for all parameters would require significant new structural insights in our proofs (e.g. Lemma 1); we leave this to future work.

Theorem 1. *Consider a sequence of at most n queries $q_1, \ldots q_n$ to a Cuckooing ACF $\mathcal{F}$ with $k = 2$ hash tables, $N = n$ slots per table, and fingerprints of length $f = \log 1/\epsilon$ bits. Let $Q = \bigcup_{i=1}^{n} \{q_i\}$. Then the expected number of false positives incurred by $\mathcal{F}$ while querying $q_1, \ldots, q_n$ is $\epsilon|Q| + O(\epsilon^2|Q| + \log^4 n)$.*

Thus, for any sequence of n queries with a support of size $|Q| = \omega(\log^4 n/\epsilon)$, the Cuckooing ACF is support optimal.

In contrast, for a worst-case input sequence, the Cyclic ACF and the Swapping ACF do not perform much better than a Cuckoo Filter. Taking the Cyclic ACF as an example, consider a sequence of n queries, each chosen uniformly at random from a randomly-selected set of size $|Q| = 1/\epsilon^{2^s}$. Each of these queries collides with some $x \in S$ under *every* choice of hash selector bits with probability $\Omega(\epsilon^{2^s})$. Thus, over n queries, the Cyclic ACF incurs $\Omega(n)$ false positives for constant ϵ and s, compared to $O(\log^4 n)$ false positives for the Cuckooing ACF via Theorem 1. See the proof of Theorem 2 for a more detailed explanation.

[5] That is to say, $\gamma = 2$.

3.1 Proof Sketch of Theorem 1

We sketch the proof of Theorem 1, but do not include the details of the proof due to space. A full proof can be seen in the full version of the paper.

Without loss of generality we assume that each false positive query only collides in one of the hash tables. Since $k = 2$, fixing a query that collides in both hash tables can be simulated by fixing each hash table separately.

To simplify notation, we define $B(i, C) = h^{\ell}_{C[i]}(x_i)$ to be the slot storing x_i under configuration C, and $B'(i, C) = h^{\ell}_{1-C[i]}(x_i)$ to be the alternate slot for x_i.

Let C_0 be the configuration of $\mathcal{F}$ before the first query q_1, and for $1 \leq i \leq n$ let C_i be the configuration after query q_i. For each $1 \leq i \leq n$, if q_i is a false positive under C_{i-1}, let k_i be the number of elements moved when fixing query q_i; otherwise let $k_i = 0$. We denote the sequence of elements moved when fixing q_i as $x_{i_1}, x_{i_2}, \ldots, x_{i_{k_i}}$. Thus, q_i collides with x_{i_1} under C_{i-1}. We call the sequence of slots affected by these movements $B(i_1, C_{i-1}), B(i_2, C_{i-1}), \ldots B(i_{k_i}, C_{i-1}), B'(i_{k_i}, C_{i-1})$ the ***path*** on C_{i-1} of q_i.

We say that q_i ***loops*** if one of the moved elements repeats; i.e. there exist $1 \leq \ell_1 < \ell_2 \leq k_i$ such that $i_{\ell_1} = i_{\ell_2}$. Interestingly, classic cuckoo hashing analysis generally only needs to bound the number of queries that loop twice, as only twice-looping queries force a rebuild. However, even a query that loops once cannot be fixed in a Cuckooing ACF, so we must bound how frequently this happens in our analysis.

Let the ***initial false positives*** be the queries in Q that are false positives for $\mathcal{F}$ in configuration C_0.

We start with a structural lemma: the elements moved when fixing any query consist of a (possibly empty) sequence of elements stored in the slot they occupied in C_0, followed by a (possibly empty) sequence of elements not stored in the slot they occupied in C_0.

Lemma 1. *If a query q_i on a configuration C_{i-1} moves an element x_{i_ℓ} satisfying $C_{i-1}[i_\ell] \neq C_0[i_\ell]$, and q_i does not loop, then all j with $\ell \leq j \leq k_i$ satisfy $C_{i-1}[i_j] \neq C_0[i_j]$.*

Proof. This proof is by induction on j; the base case $j = \ell$ is immediate.

Assume by induction that $C_{i-1}[i_{j-1}] \neq C_0[i_{j-1}]$ for some $j > \ell$. Since q_i does not loop, when $x_{i_{j-1}}$ is moved, it cannot have been moved previously while fixing q_i, and thus must be stored in slot $B(i_{j-1}, C_{i-1})$. Then after $x_{i_{j-1}}$ is moved it must be stored in slot $B(i_{j-1}, C_0)$; this must be equal to the slot storing x_{i_j}. Because q_i does not loop, x_{i_j} must be stored where it was when the fixing began; i.e. in $B(i_j, C_{i-1})$. Thus $B(i_j, C_{i-1}) = B(i_{j-1}, C_0)$, so $C_0[i_j] \neq C_{i-1}[i_j]$, as otherwise x_{i_j} and $x_{i_{j-1}}$ would be stored in the same slot in C_0.

Lemma 1 immediately gives structure to the problem in two key ways. First, it limits how queries can break one another: if q is a false positive, but is not an initial false positive, then there must be some initial false positive q_i that caused q to become a false positive. We do not need to worry about non-initial false positives causing other, new false positives. Second, it ties the behavior of all

elements to how they behave on the initial configuration C_0. This means that we can make statements about how queries interact using C_0; we do not need to reset our analysis every time the filter configuration changes.

Let us summarize how to obtain the $\epsilon|Q| + O(\epsilon^2|Q| + \log^4 n)$ bound in Theorem 1. The initial false positives immediately give us a cost of $\epsilon|Q|$ false positives; we must bound the cost of all other queries (including repeated queries to the initial false positives) by $O(\epsilon^2|Q| + \log^4 n)$.

We give a set of four criteria that constitute ***costly queries***. For example, a query q_i is costly if it hashes to the path of some initial false positive q_j on C_0—this means that q_i can break q_j. Another example is if q_i loops—in that case, the Cuckooing ACF cannot fix it.

We begin by showing that all false positives are either costly queries, or are initial false positives. (Lemma 1 is the basic building block of this proof.)

The remainder of the proof uses a potential function analysis, where the potential of a configuration of the filter is the number of pairs (q, x_i), where q is a query, and x_i is an element of S stored in its original position (i.e. its position in C_0). One important property of this potential function is that if a query is not costly, it has no amortized cost (if it is a false positive, the potential function decreases by at least 1, offsetting the false positive cost incurred by the query)—again, Lemma 1 is crucial in showing this step. This is where we bound the cost of repeated queries to initial false positives—each such false positive can be charged to the (costly) query that broke it.

Then, we begin analyzing the impact of the costly queries on the potential function. We show that the expected amortized cost of a costly query q_i is $O(1 + \epsilon|Q|k_i/n)$—that is to say, it depends on the length of the path of q_i.

We complete our analysis by first bounding k_i for each q_i (conditioned on q_i being costly), and finally bounding the total cost of all costly queries. The number of elements moved by each query are not independent—for example, it is possible (though extremely unlikely) that all $x \in S$ are stored in $n + 1$ slots, where for each j the second hash of x_j is equal to the first hash of x_{j+1}. In this case, we would have $\mathbb{E}[k_i] = \Omega(n)$ for all false positive queries. To avoid these cases, we must show that all of the paths of costly queries are fairly small and do not intersect with high probability, allowing us to treat them independently. These bounds are the source of the $O(\log^4 n)$ term in the final bound.

4 Experiments

In this section, we examine how the Cuckooing ACF performs on network trace datasets. There are two main takeaways from this section. First, the design of the Cuckooing ACF results in better practical performance than previous adaptive filters on network trace datasets. Second, the analysis of Sect. 3 extends to practice: an adaptive cuckoo filter with practical parameter settings (including very high load factor) still achieves strong performance.

Our experiments use three network traces from the CAIDA 2014 dataset, as in the experiments of Mitzenmacher et al. [13]: equinix-chicago.dirA.20140619

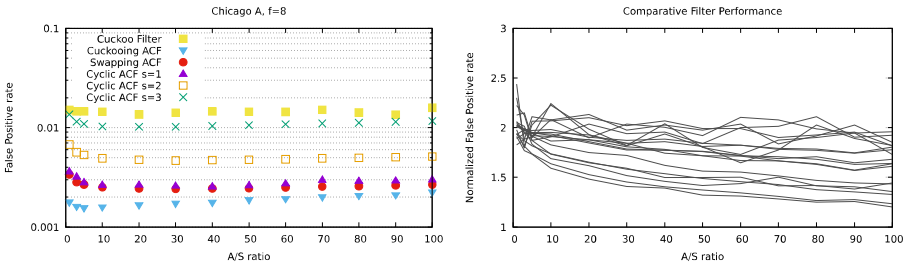

Fig. 1. We examine the false positive rate of each adaptive filter, varying the ratio of the number of queries to the number of stored elements. The right hand figure normalizes the false positive rate by the number of false positives incurred by the Cuckooing ACF. It summarizes the results for the Swapping ACF and the Cyclic ACF with $s = 1$, for all three datasets, for $f = 8, 12, 16$.

(which we call "Chicago A") equinix-chicago.dirB.20140619-432600 ("Chicago B"), and equinix-sanjose.dirA.20140320-130400 ("San Jose"). Let A be the set of query elements. We perform tests for different $|A|/|S|$ ratios; specifically $|A|/|S| = \{1, 3, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$.

We begin by setting $n = |S|$ using the prescribed $|A|/|S|$ ratio and the total number of unique flows in the trace. The first n unique flows seen in the trace are inserted into each filter. The remaining flows in the trace (those not in S) are used as queries. We consider six data structures in our experiments:

- a classic Cuckoo Filter, with $k = 4$ hash tables and $b = 1$ slots per bucket;
- the Cuckooing ACF, with $k = 4$ hash tables and $b = 1$ slots per bucket;
- three implementations of the Cyclic ACF described in [13], with $s = 1$, $s = 2$, and $s = 3$ hash selector bits. To ensure a fair comparison in space usage, each hash selector bit used is accounted for with a corresponding decrease in the number of fingerprint bits; and
- a Swapping ACF with $b = 4$ slots per bin and $k = 2$ hash tables.

All filters are at 95% occupancy in all of our experiments (i.e. $\gamma = 1/.95$). We give results for fingerprints of length $f = 8$ bits on Chicago A, and summarize key results for Chicago B and San Jose, as well as results with $f = 12$ and $f = 16$ bits on all datasets. All results given are average performance over 10 trials.

4.1 Experimental Results

The left hand plot in Fig. 1 shows that the Cuckooing ACF has the strongest performance of all adaptive filters on the Chicago A dataset with $f = 8$. Its performance is particularly strong for low values of $|A|/|S|$—that is to say, its performance is strong when the number of unique queries is small relative to n (as one may expect given Theorem 1).

We ran further experiments, using fingerprints of size $f = 8$, $f = 12$, and $f = 16$ on Chicago A, Chicago B, and San Jose datasets, achieving similar (in

fact slightly better) results. These experiments are summarized in the right hand plot of Fig. 1, and provided in full in the full version of the paper. The y-axis in this figure indicates the false positive rate of the given filter divided by the false positive rate of the Cuckooing ACF. This plot only includes the two best filters: the Swapping ACF, and the Cyclic ACF with $s = 1$. We run the experiments for three fingerprint sizes $\{8, 12, 16\}$ on all three datasets, giving 18 total lines in the plot. Note that there is some overlap with the left hand plot—one of the bottommost two lines in the plot corresponds to the Swapping ACF with $f = 8$ on Chicago A. Specifically, the Cuckooing ACF does even better with larger fingerprints like $f = 12$ and $f = 16$ compared to $f = 8$.

Overall, the Cuckooing ACF always performs at least as well as every other cuckoo filter on these datasets, often outperforming them by nearly a factor of 2.

5 Adversarial Adaptivity

Previous work leaves a dichotomy: the Adaptive Cuckoo Filters of Mitzenmacher et al. [13] work well in practice, whereas the "Broom Filter" of Bender et al. [2] is effective even against an adversary that tries to "learn" a filter's internal state. In Sects. 3 and 4 we showed that the Cuckooing ACF is practical while retaining theoretical bounds. But our theoretical bounds are not adversarial; they are based on the number of unique queries made to the filter. Can our results be taken further—is there an ACF that adapts effectively even against an adversary?

In this section we give a general lower bound showing that an adversary can obtain a false positive rate of $\Omega(1)$ against any space-efficient ACF. This result is closely tied to a key structural distinction: the Broom Filter is difficult to implement because the length of the stored fingerprint may be different for each element. Our lower bound shows that this flexibility is, in fact, necessary in order to achieve adaptivity.

5.1 Definition

Bender et al. [2] defined a notion of adaptivity that captures a worst-case adversary attempting to maximize the filter's false positive rate. We summarize this model in this subsection, and refer readers to [2] for a more thorough discussion.

In the ***adaptivity game***, an adversary generates a sequence of queries. After each query q, the adversary and filter both learn if q was a false positive. The filter may change its internal representation in response. The adversary will use whether or not q was a false positive to determine the further queries.

At any time, the adversary may name a special element $\hat{q}$—the adversary is asserting that this query is likely to be a false positive. The adversary "wins" if $\hat{q}$ is a false positive, and the filter "wins" if $\hat{q}$ is not a false positive.

The ***sustained false positive*** rate of a filter is the maximum probability ϵ with which the adversary can win the adaptivity game. We call a filter $\mathcal{F}$ ***adaptive*** if $\mathcal{F}$ can achieve a sustained false positive rate of ϵ for *any* constant ϵ.

5.2 Lower Bounds

To begin, we note that the Cyclic ACF is not adaptive. A nearly-identical proof shows that the Swapping ACF is not adaptive.

Theorem 2. *Let $\mathcal{F}$ be a Cyclic ACF with $k = O(1)$ hash tables, each with $N = \Theta(n)$ slots. Then there exists an adversarial strategy, making $\Theta(2^s/\epsilon^{2^s})$ queries, which wins the adaptivity game against $\mathcal{F}$ with probability $\Omega(1)$. Thus the sustained false positive rate of $\mathcal{F}$ is $\Omega(1)$.*

Proof. The adversary begins by selecting a query element q_1 at random. The adversary queries q_1 2^s times. If q_1 is a false positive every time it is queried, the adversary sets $\hat{q} \leftarrow q_1$; otherwise the adversary chooses a new query element q_2 and repeats. This process is repeated until $O(1/\epsilon^{2^s})$ query elements have been chosen, requiring $O(2^s/\epsilon^{2^s})$ queries overall.

We show that the adversary finds a $\hat{q}$ with probability $\Omega(1)$, and that $\hat{q}$ will be a false positive with probability 1.

Each time q collides with an element $x_i \in S$, the hash selector bits associated with x_i are incremented; thus, if q does not collide with x_i on the jth query, it will not collide on the j'th query for $j' > j$. Then if q is a false positive on all 2^s collisions, there is an $x_j \in S$ such that $h^f(q, \alpha) = h^f(x_j, \alpha)$ for all $\alpha \in \{0, \ldots, 2^s - 1\}$. We immediately obtain that any $\hat{q}$ found by the adversary is a false positive with probability 1.

For a given query q_i and a given $x_j \in S$, the probability that $h^\ell_{\beta_j}(q_i) = h^\ell_{\beta_j}(x_j)$ is $1/n$. The probability that, for all α, $h^f(q_i, \alpha) = h^f(x_j, \alpha)$ is $1/\epsilon^{2^s}$.

After the algorithm has made $1/\epsilon^{2^s}$ queries, the probability that there exists a query q' and an $x^* \in S$ such that $h^f(q_i, \alpha) = h^f(x_j, \alpha)$ for all α is $1 - (1 - \epsilon^{2^s}/n)^{n/\epsilon^{2^s}} = \Omega(1)$. Thus, the adversary finds a $\hat{q}$ with constant probability.

One might think that hashing elements to another bucket (as in the Cuckooing ACF) is sufficient to make a filter adaptive. The reason Theorem 2 gives such a strong lower bound for the Cyclic ACF is that when we move an element to the next fingerprint, it is still a false positive with probability ϵ. A constant number of these movements still leaves a significant probability that a false positive is not yet fixed. In contrast, when a colliding element is moved in the Cuckooing ACF, it still collides with the query with probability only ϵ/n—this seems low enough that almost all queries are successfully fixed after only a single movement.

Nonetheless, the adversary can use a birthday attack to find a small set of elements that cannot all be simultaneously fixed.

We obtain lower bounds for a fairly broad class of filters, where the total total information stored (i.e. hash index plus location plus fingerprint) for each

element is at most $\log(n/\epsilon) + O(1)$ bits. This stands in contrast to the Broom Filter of Bender et al. [2], which is adaptive and which stores an *average* of $\log(n/\epsilon) + O(1)$ bits—in short, this proof shows that the nonuniformity of hash lengths in [2] is crucial to achieving adaptivity.

Definition 1. *A* deterministic k-adaptive filter $\mathcal{F}$ *on* n *elements with false positive rate* ϵ *is a filter satisfying the following:*

- $\mathcal{F}$ *has access to* k *uniform random hash functions* $h_0, \ldots, h_{k-1}$*. Each hash has length at most* $\log(N/\epsilon)$*, for some* $N = O(n/k)$ *with* $N \geq n/k$*.*
- *For every configuration* C *of* $\mathcal{F}$*, each* $x_i \in S$ *is stored using at least one hash* $h_{C[i]}(x)$*,* $0 \leq C[i] \leq k-1$*.*
- *The filter answers* `present` *to a query* q *on configuration* C *if there exists an* x_i *such that* $h_{C[i]}(q) = h_{C[i]}(x_i)$*. Otherwise, it answers* `absent`*.*
- *On a false positive* q*,* $\mathcal{F}$ *updates* C *to a new configuration* C' *in round-robin order. In particular, if a query* q *collides with an element* $x_i \in S$ *stored using* h_β*, then* x_i *is stored in* C' *using* $h_{\beta'}$ *satisfying* $\beta' = \beta + 1 \pmod{k}$*.*

By setting each hash h_i in Definition 1 so that for any $i \in \{0, \ldots, k-1\}$ and $x \in U$, $h_i(x)$ is the concatenation of $h_i^\ell(x)$ and $h^f(x, i)$, the Cuckooing ACF is a deterministic k-adaptive filter. By setting $h_{(i,\alpha)}(x)$ to be the concatenation of $h_i^\ell(x)$ and $h^f(x, \alpha)$, the Cyclic ACF is a deterministic $k2^s$-adaptive filter.[6]

The round-robin ordering requirement stands out as being a bit artificial, but our proof can fairly easily be generalized to handle other deterministic methods to update the configuration.

Theorem 3. *There exists an adversarial strategy making* $O(n)$ *queries such that, for any deterministic* k*-adaptive filter* $\mathcal{F}$ *with* $k < \log n/(6 \log\log n)$ *and* $\epsilon > 1/(n^{1/k})$*, the adversary wins the adaptivity game with probability* $\Omega(1)$*.*

Querying to Find a Mutually Unfixable Set. The proof of Theorem 3 begins with the adversary searching for a structure that "blocks" the filter, preventing it from fixing a false positive.

Consider a stored element $x_i \in S$, and fix a filter $\mathcal{F}$ with k hash functions $h_0, \ldots h_{k-1}$. A set of queries K is called ***mutually unfixable for*** x_i if,

- for all $\beta \in \{0, \ldots, k-1\}$, there exists a $q' \in K$ with $h_\beta(x_i) = h_\beta(q')$, and
- for all $q' \in K$ there exists a $\beta, \in \{0, \ldots, k-1\}$ such that $h_\beta(x_i) = h_\beta(q')$.

The goal of our adversary is to find a mutually unfixable set of the queries, since for any configuration, at least one element in such a set is a false positive.

We briefly sketch the remaining details of our adversary to prove Theorem 3. The full proof can be found in the full version of the paper.

The adversary begins by choosing a set Q of size $(1+1/k)N/(\epsilon n^{1/k})$ uniformly at random from U. We show that if Q is this size, then with constant probability Q will contain $\Theta(1)$ mutually unfixable sets, each of size $O(k)$.

[6] The Cyclic ACF does not quite satisfy Definition 1 since its hashes are not independent. However, this only makes it easier for an adversary to find false positives.

The adversary then queries members of Q for $2k$ rounds; any query that is a false positive during the second set of k rounds is stored in a set Q_d. We show that Q_d will be the union of some mutually unfixable subsets of Q.

Finally, the adversary repeatedly selects k elements from Q_d and randomly selects a permutation P on these elements. The adversary queries these elements in order, twice. We show that, over $O(n)$ total queries, the adversary will (with constant probability) find k elements corresponding to a mutually unfixable set, and query them in an order such that each is a false positive every time it is queried. Then, the adversary can find a false positive $\hat{q}$ with constant probability.

References

1. Bender, M.A., Das, R., Farach-Colton, M., Mo, T., Tench, D., Ping Wang, Y.: Mitigating false positives in filters: to adapt or to cache? In: Symposium on Algorithmic Principles of Computer Systems (APOCS), pp. 16–24. ACM-SIAM (2021)
2. Bender, M.A., Farach-Colton, M., Goswami, M., Johnson, R., McCauley, S., Singh, S.: Bloom filters, adaptivity, and the dictionary problem. In: Foundations of Computer Science (FOCS), pp. 182–193. IEEE (2018)
3. Bender, M.A., et al.: Don't thrash: how to cache your hash on flash. Proc. VLDB Endow. **5**(11), 1627–1637 (2012)
4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
5. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: a survey. Internet Math. **1**(4), 485–509 (2004)
6. Carter, L., Floyd, R., Gill, J., Markowsky, G., Wegman, M.: Exact and approximate membership testers. In: Symposium on Theory of Computing (STOC), pp. 59–65 (1978)
7. Eppstein, D.: Cuckoo filter: simplification and analysis. In: Scandinavian Symposium and Workshops on Algorithm Theory (SWAT), vol. 53, pp. 8:1–8:12 (2016)
8. Eppstein, D., Goodrich, M.T., Mitzenmacher, M., Torres, M.R.: 2–3 cuckoo filters for faster triangle listing and set intersection. In: Principles of Database Systems (PODS), pp. 247–260. ACM (2017)
9. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.D.: Cuckoo filter: practically better than Bloom. In: International Conference on Emerging Networking Experiments and Technologies (CoNEXT), pp. 75–88. ACM (2014)
10. Geravand, S., Ahmadi, M.: Bloom filter applications in network security: a state-of-the-art survey. Comput. Netw. **57**(18), 4047–4064 (2013)
11. Jiang, S., Larsen, K.G.: A faster external memory priority queue with decrease keys. In: Symposium on Discrete Algorithms (SODA), pp. 1331–1343. ACM-SIAM (2019)
12. Lovett, S., Porat, E.: A lower bound for dynamic approximate membership data structures. In: Foundations of Computer Science (FOCS), pp. 797–804. IEEE (2010)
13. Mitzenmacher, M., Pontarelli, S., Reviriego, P.: Adaptive cuckoo filters. In: Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 36–47 (2018)
14. Pagh, A., Pagh, R., Rao, S.S.: An optimal bloom filter replacement. In: Symposium on Discrete Algorithms (SODA), pp. 823–829. ACM-SIAM (2005)
15. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)

16. Pandey, P., Bender, M.A., Johnson, R., Patro, R.: A general-purpose counting filter: making every bit count. In: International Conference on Management of Data (SIGMOD), pp. 775–787. ACM (2017)
17. Porat, E.: An optimal bloom filter replacement based on matrix solving. In: Frid, A., Morozov, A., Rybalchenko, A., Wagner, K.W. (eds.) CSR 2009. LNCS, vol. 5675, pp. 263–273. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03351-3_25
18. Wang, M., Zhou, M., Shi, S., Qian, C.: Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. Proc. VLDB Endow. **13**(2), 197–210 (2019)