A Survey of Adaptive AMQs

Eric Knorr, Richard Xu, Zachary Yedidia

CS 223

November 4, 2020

Abstract

Bloom filters and other approximate membership query data structures, or AMQs, are often used to help avoid costly lookups to large dictionaries. They achieve this by keeping a compact representation of a set **S** of keys from a universe **U**. All AMQs support inserts and lookups while some also support deletes. When performing a lookup for an item x that has been inserted, i.e. $x \in \mathbf{S}$, the AMQ is guaranteed to return *present*. The downside of their compact representation of **S** is that, when a lookup is performed for an item $x \notin \mathbf{S}$, *present* may be returned with probability ϵ. This **false-positive probability** is a usually a tunable parameter with a lower ϵ typically requiring more space to be used by the AMQ. That said, their compact representation is also what makes them helpful since they can be kept in memory or on a local machine and queried locally to avoid having to access disk or another machine over a network. Two of the primary concerns when analyzing an AMQ are then, how often does it return false-positives and how much space does it require to do so?

The majority of current AMQs offer strong guarantees for independent queries; however, the false-positive probability rates for most can be pushed towards 1 given the right sequence of non-independent queries. Some recent AMQs have been designed to account for this shortcoming. We call these **adaptive** AMQs because they maintain a false-positive probability of ϵ for every lookup regardless of the answers to previous lookups (adapting their internal representation whenever a mistake is made so that it will be corrected in the future).

We provide a survey of techniques used for creating adaptive AMQs, comparing and contrasting existing implementations, and discussing the tradeoff between practicality and ease of analysis.

1 Introduction

Bloom filters and other approximate membership query data structures, or AMQs, are often used to help avoid costly lookups to large dictionaries. They achieve this by keeping a compact representation of a set S of keys from a universe U. All AMQs support inserts and lookups while some also support deletes. When performing a lookup for an item x that has been inserted, i.e. $x \in S$, the AMQ is guaranteed to return *present*. The downside of their compact representation of S is that, when a lookup is performed for an item $x \notin S$, *present* may be returned with probability ϵ. In the dictionary context, upon receiving a query for an element, an AMQ can be used to quickly determine if the element is in the dictionary at all, and if not the costly dictionary access can be avoided. With probability ϵ the AMQ will return a false-positive, causing a dictionary lookup when one was not necessary.

This **false-positive probability** is a usually a tunable parameter with a lower ϵ typically requiring more space to be used by the AMQ. That said, the compact representation of an AMQ is

also what makes it helpful since it can be kept in memory or on a local machine and queried locally to avoid having to access disk or another machine over a network. The two primary concerns when analyzing an AMQ are then, how often does it return false-positives and how much space does it require to do so?

The majority of current AMQs only offer strong guarantees for independent queries, but the false-positive probability rate for most of these can be pushed toward 1 given the right sequence of queries. In the adversarial case, this can be done by repeated lookups of elements that result in false-positives, but, even in regular workloads, the lack of strong guarantees for sequences of lookups can lead to poor performance depending on the circumstances. For example, Mitzenmacher et al. [8] discuss how repeated lookups can arise during packet processing because the lookups may correspond to flow identifiers. Their solution to this problem was to create a variant of the Cuckoo Filter [7] which adapts to false positives by removing them for future queries and show using simulations that this does indeed result in a better false positive rate under such circumstances. Bender et al. [2] take a similar approach in designing the Broom Filter which is a variant of the Quotient Filter [3] that also adapts to false positives, but rather than showing their advantages through simulations, they define an **adaptive** AMQ as one that maintains a false-positive probability of ϵ for each query regardless of the answers to previous queries and show that the Broom Filter is proveably adaptive.

We first discuss previous attempts at adaptivity, such as the bloomier filter and the retouched bloom filter. Then, we focus on the adaptive cuckoo filter and the broom filter. We study the theory for designing adaptive filters and compare the two filters' theoretical guarantees and practical performance. Through a long endeavor to implement the broom filter, we find concerns about the broom filter that makes it impractical. In other words, we find that adaptive cuckoo filters are currently the most practical solution for adaptive AMQs.

2 Various Strategies for Adaptivity

In the past 15 years, several methods for addressing this particular shortcomings of AMQs have been explored. One method seen in the *Bloomier Filter* [5] makes use of a whitelist W with size w of predetermined potential false-positives. Whenever there is a lookup for an element $x \in W \setminus S$, the Bloomier Filter returns *is false-positive*. This works well when the filter can determine problematic queries ahead of time; however, the Bloomier Filter requires $O((n + w)\log(1/\epsilon))$ bits of space and the list of bad queries cannot be dynamically updated without a potential blow up in space. A more dynamic method is to targetedly remove false-positives at the cost of introducing random false-negatives. This can be seen in *Retouched Bloom Filters* [6]; however, not all applications are tolerant of false-negatives which makes this option infeasible in many situations. Yet another method that has been used is to assign different numbers of hash functions to different elements depending on their frequency as seen in *Weighted Bloom Filters* [4]; however, this technique is dependent on pre-determined statistics gathered about the workload and is not able to dynamically adapt. More recently, another method has been explored where the representations for elements within the AMQ are changed in response to false positives as seen in the *Adaptive Cuckoo Filter* [8] and *Broom Filter* [2]. Both of these use a technique where a second structure is accessed in tandem with the underlying set that is used to provide the information required to adapt to false-positives. We will go into greater detail on these last two, because they are the most rigorous solutions to adaptive AMQs that we could find, and explore both their theoretical guarantees and practical concerns.

3 Adaptive Cuckoo Filters

The *adaptive cuckoo filter* [8] is a practical solution to the AMQ problem with adaptivity, combining a cuckoo filter and cuckoo hash table. Both of these structures are based on the technique of *cuckoo hashing*, where each element can be stored in two different locations. We first describe the tools used in an adaptive cuckoo filter, then describe how ACF uses them to be adaptive [1].

3.1 Cuckoo Hashing

The adaptive cuckoo filter uses a cuckoo hash table [10] and the cuckoo filter [7]. As the name suggests, the cuckoo filter is an efficient data structure for AMQ's. Both objects maintain an array of length n. Each entry of the array is known as a *bucket*, and a bucket can have either one or multiple *slots*. The structures use two hash functions $h_1, h_2 : U \to \{0, \ldots, n-1\}$ to determine where each element of the set is stored.

In a cuckoo hash table, the entire key is stored in the bucket. In a cuckoo filter, only a fingerprint is stored. Insertions, lookup and deletes are performed as follows. For convenience, let x be the argument in all three procedures.

Insert: Compute both $h_1(x)$ and $h_2(x)$. If either of the two array entries (buckets) indexed by the $h_1(x)$ and $h_2(x)$ has remaining slots, we insert x into that slot. Otherwise, we pick one of the occupied slots at random. Suppose that slot currently stores the item y. Then, we evict y, insert x into the now-empty slot and perform `INSERT(y)`.

This process will succeed as long as we do not run into cycles, which occurs with low probability as long as the load is bounded. Another issue is that we need to compute $h_1(y), h_2(y)$ when we evict y and insert it back. This is not possible in a normal cuckoo filter since we only have access to the fingerprint, and the issue is resolved using a technique known as *partial-key cuckoo hashing*. However, this is not necessary when a cuckoo hash table and filter work in tandem, as we will explore in the next section.

Lookup: Compute $h_1(x)$ and $h_2(x)$ and check if either of the two buckets stores item x (or its fingerprint). If it does, return the object (or *present*, for a cuckoo filter). Otherwise, return `absent`.

Delete: After looking up the location of x, remove the item (or its fingerprint) from the data structure.

The cuckoo hash table provides worst case $O(1)$ lookup time and amortized $O(1)$ insertion time, as long as the load factor, the proportion of slots that are occupied, is less than some constant c. The value of c is around 50% when the bucket can only store 1 item, and increases to over 90% when the bucket can store 3 items. The worst case $O(1)$ lookup is a significant improvement over hash tables with other strategies of resolving collisions such as linear probing. The general phenomenon that having two choices increases a data structure's performance is named by Mitzenmacher as "the Power of Two Choices" [9].

In the cuckoo filter, a lookup causes a false positive only if two elements share the same hash and the same fingerprint. Then, we can tune the false positive probability ϵ using extra space by increasing the length of the fingerprint. The filter is more space efficient than the Bloom filter when the false-positive rate ϵ is relatively small (less than 3%), and continues to perform even when the load is above 90%. Fan et. al [7] generalized the filter to include multiple slots per bucket and used a technique known as *semi-sorting* to further improve the filter's space efficiency when buckets have more than 1 slots.

[1] Which we will keep brief since the reader invented this data structure.

Data Structure	Object in each Slot
Cuckoo Hash Table	Element $x \in S$
Classic Cuckoo Filter	Fingerprint hash(x)
Cuckoo Filter in ACF	Pair $(f_\alpha(x), \alpha)$

Table 1: Objects Stored in Each Cuckoo Data Structure

3.2 Adaptive Cuckoo Filter

Mitzenmacher et. al introduced in [8] the adaptive cuckoo filter, which can answer approximate membership queries while adapting to false positives. We will use a framework similar to the one used in the broom filter analysis [2] to facilitate comparisons and proofs in the future.

The adaptive cuckoo filter uses a cuckoo hash table and cuckoo filter, as described above. The cuckoo hash table would be stored remotely since it is only used when adapting to false positives. Instead of storing on a fingerprint p in the cuckoo filter, we store a pair (p, α), where p is the fingerprint and α is a number with s bits that will help the filter adapt. The filter maintains a family of hash functions $f_0, \ldots, f_{2^s-1}$. The fingerprint of an element $x \in U$ and a number α is equal to $f_\alpha(x)$.

The filter and the table satisfy the following invariant: for each element x in the hash table, x's fingerprint is stored in the same location in the Cuckoo filter as it is in the table.

Insertions and Deletions: To insert or delete an element $x \in S$, perform the operation in both the cuckoo hash table and the cuckoo filter. Since the two structures use the same hash function, the element will be inserted into the same location in the array. Therefore, the invariant is maintained. The cuckoo hash table inserts element x and the cuckoo filter inserts the pair $(f_0(x), 0)$.

When we evict a finger p_y from the cuckoo filter, the corresponding element y is also evicted from the cuckoo hash table. Since the operation already accesses the remote hash table, retrieving the element y from the hash table has minimal overhead. As a result, partial-key cuckoo hashing is not necessary.

Lookup and Adapting: To check if an element x is in the set, we first calculate $h_1(x), h_2(x)$. If one of the bucket contains a pair (p, α) satisfying $f_\alpha(x) = p$, i.e. the fingerprint matches, return *present*.

A false positive occurs when an element $y \notin S$ shares the same fingerprint as $x \in S$. This is detected at the time when we retrieve the element from the Cuckoo hash table. To adapt for this false positive, we increment α and change the fingerprint for x accordingly. It is unlikely for x, y to share the same fingerprint again. Therefore, the filter has adapted to this false positive. The adaptive cuckoo filter shares a similar generalization as the cuckoo filter for cases where each bucket has more than one cell, with some changes to the adapting algorithm.

Table 1 illustrates the three different cuckoo structures and what they store in each slot.

Although their paper does not present many theoretical guarantees about their filter, Mitzenmacher et. al performed simulations with randomly generated queries and real-world data and found that in practice the adaptive cuckoo filter beats the cuckoo filter in most configurations where some queries are repeated more than others. Cuckoo filters already boast high efficiency in practice, which makes the adaptive cuckoo filters even more impressive.

The adaptive cuckoo filter is also easy to implement because each bucket uses a constant number of adaptivity bits that does not change over time. As a result, the memory for the adaptivity bits can be allocated at the start and it is easy to keep them compact. We will see that this is not the case for the broom filter and that this makes an efficient broom filter implementation almost impossible.

4 Broom Filters

Bender et al. [2] introduce a different adaptive filter, called the "broom filter" (because it cleans up after itself). They make two key contributions to the field of adaptive filters: describing the broom filter data structure itself, as well as formalizing the idea of adaptivity and providing proofs regarding the guarantees of the broom filter, and adaptive AMQs in general. The broom filter builds on a previous AMQ called the quotient filter [3], and, like the adaptive cuckoo filter, consists of both a local and remote representation. In fact, one of the main theoretical results they obtain shows that any adaptive AMQ must use some remote (larger) storage in order to maintain adaptivity. Unfortunately, from our experience the broom filter is quite impractical. Even a naive implementation would be cumbersome and time-consuming to write, and important details such as how adaptivity bits can be stored and maintained efficiently are left out.

4.1 Quotient Filters

The broom filter builds on a previous work called the quotient filter, which is a non-adaptive AMQ meant to be an optimal and practical alternative to a bloom filter. The quotient filter uses a single hash function and computes a *fingerprint* for each element inserted to the set. Given an element x, the fingerprint of x is a prefix of the hash $h(x)$ and is divided into two separate pieces, the *quotient* with size q bits and the *remainder*, with size r bits. The fingerprint is therefore the first $q + r$ bits of $h(x)$. The quotient filter is an array of 2^q buckets, where each bucket consists of r bits, plus three extra metadata bits. The sizes of the quotient and remainder are tunable parameters of the quotient filter, which affect the false-positive probability ϵ.

Insertions and lookups: To insert an element x into the quotient filter, we first compute the fingerprint of x: $p(x)$. Then we find the bucket corresponding to the quotient and insert the remainder bits at that bucket. Assuming the bucket is empty during insertion, a lookup for y would be simple: we go to the correct bucket as given by the quotient of y and check if the remainder bits are the same. However, if the bucket is full, linear probing is used to find an empty bucket, provided two invariants are maintained:

1. All remainders with the same quotient are stored contiguously in a *run*, wrapping around to the beginning if necessary.

2. If remainder a is stored before remainder b, then the quotient of a is less than or equal to the quotient of b modulo the wrapping.

This means that inserting into a run that is followed immediately by another run may require shifting everything in the following run over to make room.

We then have to apply the correct metadata bits to all buckets that are touched by the probe. These metadata bits will allow us to determine which remainders correspond to which quotients, since due to linear probing remainders will be shifted from their correct quotients. The three metadata bits provide the following information for each bucket:

1. `occupied` bit: is the bucket occupied?

2. `shifted` bit: is the remainder stored at this bucket shifted from its corresponding quotient?

3. `continuation` bit: is this bucket a part of the continuation of a run?

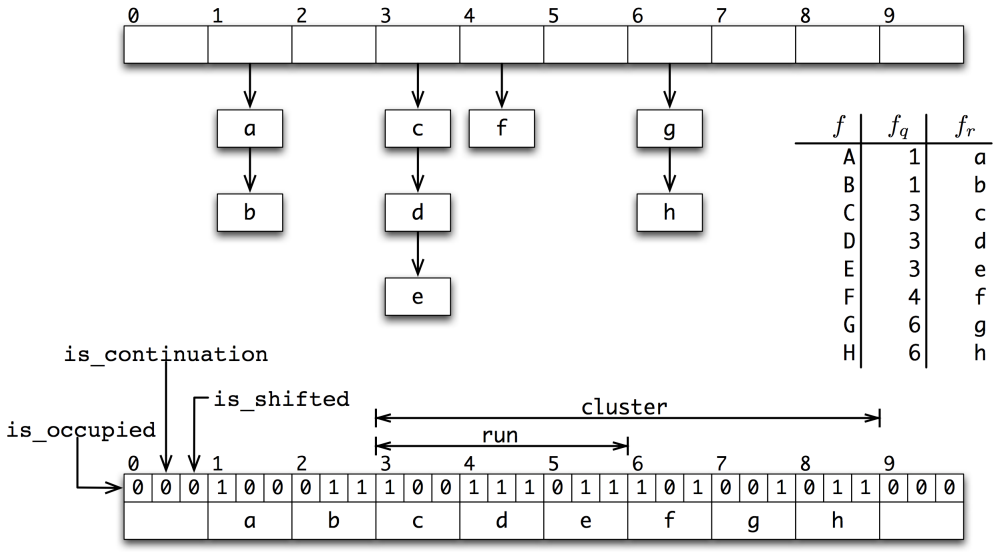

Figure 1: An example quotient filter as shown in [3]. The filter has 10 slots, and for a fingerprint f, the remainder f_r is stored in the bucket indexed by the quotient f_q. The remainder will be shifted from its intended bucket when there are quotient collisions.

Whenever a remainder is added, the `occupied` bit for its intended bucket is always set to 1. If a remainder is not stored in its intended bucket, then the `shifted` bit of the bucket it is stored in is set to 1. If the remainder has the same intended bucket as the remainder immediately before it, the `continuation` bit is set to 1. When we perform a lookup for element x, we check the bucket indexed by the first q bits of $h(x)$. If `occupied` is 0, we immediately return *absent*; otherwise, if `shifted` is 0, we scan the current run for a remainder that matches the remainder being looked up, and return *present* if a match is found, and *absent* otherwise. If the `shifted` bit is 1, then we must first find the beginning of the **cluster**, a group of runs with no empty buckets between them, and then scan forward keeping track of the number of runs and the number of occupied buckets to determine the location of our target run. If we reach the end of the cluster without finding it, then we return $false$, but if we do find the run, we compare to the remainders in that run and return $true$ or $false$ accordingly. Deletes can be performed by looking up the corresponding remainder (of which their may be multiple) and removing it and then shifting other remainders and updating metadata bits as needed.

Let m be the number of occupied buckets in the filter (the number of elements that have been inserted so far). A false positive thus occurs if $h(x) = h(y)$ when $x \in S$ and $y \notin S$. Assuming $h : \mathbf{U} \to \{0, ..., 2^{q+r} - 1\}$ generates outputs that are distributed uniformly and independently, then the probability of a false positive is given by

$$1 - \left(1 - \frac{1}{2^{q+r}}\right)^m \approx 1 - e^{-m/(q+r)} \leq \frac{m}{2^{(q+r)}} \leq \frac{2^q}{2^{(q+r)}} \leq 2^{-r}.$$

If we would like to choose r and q such that the filter has a false-positive probability of at most ϵ, and n is the maximum set size, we can choose $q = \log(n)$ and $r = \log(1/\epsilon)$. With these choices, the quotient filter achieves a false-positive probability of at most $2^{-r} = \epsilon$ while using $O(2^q r) = O(n \log(1/\epsilon))$ space.

4.2 Adding Adaptivity

The broom filter is a provably adaptive AMQ which makes use of quotient filters while still providing the guarantees about the false-positive probability for a sequence of possibly non-independent queries. The broom filter consists of both a local and remote structure. The local partition is a fully functioning AMQ on its own, but the remote partition is necessary to provide additional information for adapting to false positives. Specifically, whenever there is a false-positive caused by some element $x \notin S$, the remote structure should provide the element $y \in S$ that caused the false-positive so that we can modify its representation in the local AMQ so that future lookups of x will no longer result in false positives. The specific implementation of the remote structure is not detailed, but we provide some ideas for how it could be implemented in a later section. The need to access a remote portion of the filter may seem counter-intuitive given that AMQs are designed to avoid remote lookups, but the AMQ will only ever access the remote structure when it would have been accessed anyway (to look up an item in the dictionary or to insert an item to the dictionary).

4.2.1 Local representation

The details of the local representation are presented here but are tuned to improve lookup and insert performance, rather than for enabling adaptivity, so we only give a brief overview here. Let n be the maximum set size that will be stored by the broom filter, and ϵ be the false-positive probability of a single query. The local representation of the broom filter consists of two quotient filters, both with remainders of size $r = \log(1/\epsilon)$. The designs of these quotient filters differs based on the size of the remainders. In the **small remainder** case, where $r \leq 2 \log \log n$, both levels are quotient filters with independent hash functions. The first layer contains $O(n)$ buckets and the second layer contains $O(\log n/n)$ buckets. In the **large remainder** case, where $r > 2 \log \log n$, only a single hash function is used and backyard hashing [1] is used to differentiate them.

Insertion and deletion in the quotient filters used by the broom filter are slightly modified to ensure constant-time accesses. In particular, a probe limit is imposed in both cases on the first layer's quotient filter. If a valid bucket is not found in $O((\log n)/r)$ probes (in the small remainder case) or $O(\log n/ \log \log n)$ probes (in the large remainder case), then the second layer is used to execute the corresponding operation.

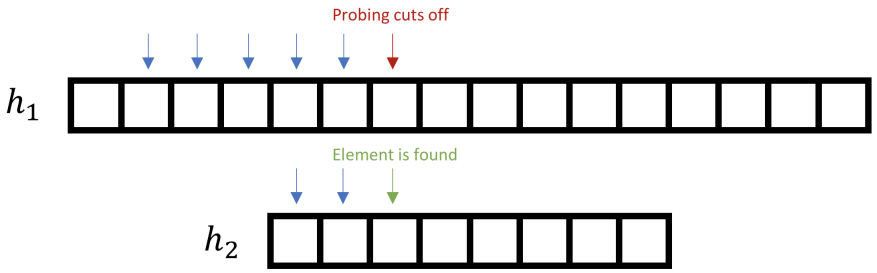

Figure 2: In the small remainder case, two hash functions are used and the second layer is smaller than the first layer. Probing in the first layer is cut off after a threshold to ensure constant-time accesses.

4.2.2 Hash function

The broom filter uses hash functions $h : \mathbf{U} \to \{0, ..., n^c\}$, for some constant $c \geq 4$. The constant c must ensure that the target set of the hash function is large enough that the hash function is

perfect and cannot cause elements to collide. However since only a subset of the bits from the hash are used in the local AMQ this doesn't cause space problems.

4.2.3 Inserts, lookups, and adaptivity bits

In the broom filter, the **fingerprint** of an element x is comprised of the first $q + r + a$ bits of the hash of x, where q is the number of quotient bits, r is the number of remainder bits, and a is the number of *adaptivity* bits. The number of adaptivity bits can vary dynamically and can be different for each element in the set. We also maintain an invariant regarding fingerprints to make extending adaptivity bits possible.

> **Invariant:** No fingerprint is a prefix of another fingerprint.

The adaptivity bits of a fingerprint may be extended in two cases:

1. When a false-positive occurs from a lookup.
2. When a fingerprint is added to the broom filter.

Note that in both cases, we must access the remote structure regardless of adaptivity. A false-positive caused by the lookup causes a lookup in the remote dictionary, and insertion requires inserting to the remote dictionary to maintain its state correctly. Therefore in these cases we can acquire some additional information from the remote structure for "free" in the sense that the remote access is the expensive operation and must be done anyway.

To check if an item is in the set, we check if the fingerprint at each bucket is a prefix of the item's hash, and if so, we return *present*. If a false positive is caused by some element $x \notin S$, this means that the fingerprint of some element y in the set is a prefix of $h(x)$. If we simply extend the fingerprint of y to include more adaptivity bits until it is no longer a prefix of $h(x)$ then x will no longer cause a false positive. Extending the fingerprint of y requires accessing the full hash of y, which can only be provided by the remote dictionary. Luckily, our invariant ensures that there is only one element in the dictionary with y's fingerprint, so we can find the unique y.

During insertion to the broom filter, we must make sure that the invariant is maintained. We do so by checking the newly added fingerprint against all other fingerprints with the same quotient. If any of those fingerprints share the same remainder, adaptivity bits are added to both fingerprints until neither is a prefix of the other (so the invariant is maintained). Extending the fingerprints that already exist in the AMQ requires using the same query in the remote dictionary as during the lookup described above. Note that additional adaptivity bits may be added in this step to handle deletions as well, but this will be explained in a later section.

It is worth mentioning that the maintenance of the invariant and the adaptivity of the filter is dependent on the hash function providing no collisions. Consider if $x \notin S$, $y \in S$ and $h(x) = h(y)$. In such a case, adding adaptivity bits to the fingerprint of y will never make it not a prefix of $h(x)$. This could potentially be resolved by allowing fingerprints to grow beyond the size of their hash, but this would likely have problematic implications for the size of the local representation. This does not appear to be addressed in the design of the Broom Filter.

4.2.4 Deletions and cleanup

What has been described so far is sufficient to achieve adaptivity for an AMQ that only supports insertions and lookups, since no lookup for any given element will ever result in a false positive more

than once. However, there are two additional features that are important to support: deletions, and space cleanup. As lookups and insertions are performed on the broom filter, the adaptivity bits of the elements will continue to grow, until eventually too much space is used (for example the full hash for every item is used).

Deletes in the broom filter are carried out like those in a quotient filter except that the quotient and the adaptivity bits are not removed but are left as a **ghost** in the filter. When a new fingerprint is added, if there is a matching ghost in the filter, then the new fingerprint takes on all the adaptivity bits of the ghost even if this is more than required for the standard insert process. This is important because it prevents an adversary from generating false-positives by repeatedly adding and removing an element form the filter that collides with some element not in the filter. Since a re-inserted element retains its adaptivity bits from before it was removed, it will not collide with any elements that collided with it in the past.

Finally, there is the issue of continually growing adaptivity bits. This is addressed done with a deamortized equivalent of rebuilding the the filter every $\Theta(n)$ times adaptivity bits are extended. The broom filter keeps two hash functions, h_a and h_b, and has phases that gradually switch between hash functions. At the beginning of a phase, frontier, we keep a counter $z = -\infty$ and only h_a is used. Once adaptivity bits are added, the smallest constant $k > 1$ elements of S that are greater than z are deleted from the filter (including their ghosts) and re-inserted using h_b, after which z is set to be the largest element that was re-inserted. This requires an access to the remote representation, but again this will only happen when the underlying dictionary is already being accessed (adding adaptivity bits already requires a call to the remote). When an element is x looked up or inserted, h_a is used if $x > z$ and h_b is used otherwise. Once z reaches the largest element in S, then a new phase begins. This is then enough to guarantee that, with high probability, there are $O(n)$ adaptivity bits in the filter at any given time. The broom filter thus achieves adaptivity while still only using $O(n \log(1/\epsilon))$ space locally.

4.3 Implementation details

Bender et al. [2] provide a thorough description of the theory of the broom filter, but some key implementation details are left out. In this section, we provide some additional details regarding how one might implement a broom filter and the practical issues that arise.

4.3.1 Adaptivity bits

Most notably, the storage of the adaptivity bits is a real problem because they are variable-sized chunks of memory that do not fit nicely into machine words. Storing the adaptivity bits directly in the quotient filter for each element is a bad idea because this means extending a fingerprint would require resizing the entire quotient filter. Instead a better approach would be to use a separate memory for the adaptivity bits and use some metadata at the start to keep track of the sizes and locations of the bits for each element. Retrieving the adaptivity bits for a certain element would require decoding the bits from the block of memory using bitwise operations to decode across word boundaries. Extending bits also causes the adaptivity bits array to need to be re-allocated each time (at least this is better than re-allocating the quotient filter), and perhaps some sort of ahead-of-time space allocation could help here too (allocating more space than necessary so resizing isn't necessary on every extension).

In general the concept of adaptivity bits is impractical because computers perform well when working with constant, repetitive data, which is the opposite of what is needed to implement adaptivity bits. Small, often single-bit extensions to variable-sized pieces of memory is an inefficient

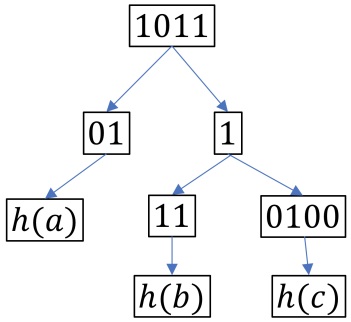

Figure 3: Using a radix tree (compressed binary tree in this case) to map fingerprints to hashes. Extending adaptivity bits is much less costly with this structure than with a hashtable.

paradigm in practice.

4.3.2 Remote layout

Another key detail that is not given attention by Bender et al. is the layout of the remote data structure, and its implementation of a `RevLookup` function, which will find the full hash of an item x given by its fingerprint $p(x)$. A naive implementation of the remote data structure would be a hashset, and the `RevLookup` operation would need to search every item in the set, which is very expensive. Perhaps a better solution would involve maintaining a mapping of fingerprints to full hashes in the remote data structure. This way, looking up the fingerprint is immediate. However this method is not easy either, since every time a fingerprint is extended, the key in the mapping must change. Implementing this mapping as a hash table (the obvious first thought) would prove ineffective as extending a fingerprint would require first extending the bits of the key and then re-hashing and re-inserting the element. Maintaining this mapping as a radix tree seems better because extending keys is not as difficult and we are already make sure that no key is a prefix of any other.

4.3.3 Hashing

Other difficulties with implementation arise with the hash function. Bender et al. mention that the hash function must map $U \to \{0, \ldots, n^c\}$ where n is the maximum set size and c is a constant ≥ 4. If the universe is larger than n^c then it is possible to have a complete collision between $h(x)$ and $h(y)$. This has a low probability of occurring but if it does occur and the colliding element is repeatedly queried, then the false positive rate will be close to 1, causing the adaptivity of the filter to fall apart. Therefore our hash function must hash from the universe to something larger than the universe and be a perfect hash function to avoid these collisions. This immediately rules out using the broom filter for strings because in that case the universe is too large. Even with a universe of 64-bit integers (the standard case), keeping a set of size larger than 2^{16} requires using a hash function that produces $>$ 64-bit integers. Maintaining integers that are larger than the word size (and that may not fit in a multiple of the word size) is often impractical and difficult.

4.4 A Lower Bound for Adaptivity

While the implementation of the broom filter may be impractical, the paper's theory section offers an important insight in the design of adaptive AMQs. As part of the broom filter's design, Bender

et al. also prove that any AMQ storing a set of size n from a universe of size $u > n^4$ requires $\Omega(\min\{n \log n, n \log \log u\})$ bits of space with high probability in order to maintain a sustained false-positive rate $\epsilon < 1$. This is why the remote representation is required if the local representation is to be near the optimal size for a traditional AMQ and why it is not surprising that the Adaptive Cuckoo Filter also shares this two part structure. A similar proof was explored with the *Bloomier Filter* [5] in assessing the space required when dynamically updating its whitelist. The proof makes use of adversarial model where the goal of the adversary is to adaptively generate a sequence of $O(n)$ lookups that force the AMQ to either use too much space or to fail to sustain its false-positive rate of ϵ.

The Adversary's Attack: The adversary starts with a set S of size n chosen uniformly at random form U. The attack then proceeds in rounds. For each round, a set Q of size n is chosen uniformly at random from $U \setminus S$. All members of Q are then looked up. On each subsequent round, any false-positives from the previous round are queried again.

The idea of the proof is that we expect the adversary to find a certain fraction of false-positives which is shown to have a high concentration using Chernoff bounds. Therefore, with high probability, the AMQ will have to fix an ϵ fraction of false-positives in each round or fail to sustain its false-positive rate. Each time the AMQ corrects a false-positive it must change its configuration. Each configuration of the AMQ for a given set S can be defined using the list of false-positives that have been corrected. This results in a one-to-one mapping between the set of fixed false-positives and the configurations of the AMQ. The number of configurations can then be lower bounded by the number of sets of false-positives that the adversary can force the AMQ to fix. The space required for the AMQ is thus lower bounded by the amount of space required to enumerate its configurations. Finally, the bound is shown to be tight by providing a construction on an AMQ, the broom filter, that uses $O(\min\{n \log n, n \log \log u\})$ bits of space with high probability.

5 Future Work

After the strenuous hardship that is the broom filter implementation attempt, we present some future directions for this project. We believe that the directions we pose are actual directions for potential research, not cliché generalizations, and hope to continue exploring these topics.

5.1 Benefits of Rigorous Adaptivity

While the fully optimized design of the broom filter proved difficult to implement (especially given time constraints), a naive implementation that uses as much space as it wants would likely be more feasible. This in turn could be used to see the benefits of the broom filter's strong guarantees in actual work loads. This could also be used to test whether strategies that are simpler to implement but may have weaker theoretical guarantees, like the Adaptive Cuckoo Filter, actually fair any worse in practice.

5.2 A Generalized Framework for Adaptivity

Bender et al. [2] made significant strides in showing what is required for a generalized AMQ to be adaptive, in particular with their lower bound on the required space, but this can likely be taken even further. For example, are the discrete representations of elements that we see in both the Adaptive Cuckoo and Broom Filters also a requirement for adaptivity or could a traditional Bloom Filter be made adaptive using the right strategy?

5.3 Making More AMQs Adaptive

After analyzing the adaptive cuckoo filter [8] and Bender et al.'s work on making a generalized AMQ adaptive, we can apply our newfound knowledge of adaptivity to other currently non-adaptive AMQs, especially with real world traffic data. For example, perhaps the Bloomier filter [5], an early attempt at adapting to false positive using a whitelist, could be expanded to be adaptive with real world network data by changing the whitelist to a cache of previously seen false positives.

The broom filter and the ACF take different approaches towards adaptivity. The broom filter extends stored fingerprints using the concept of adaptivity bits, which we have seen is less practical, while the ACF achieves adaptivity by changing the hash functions used dynamically. Perhaps this idea of switching hash functions can be applied more generally to many different kinds of AMQs via a method of partitioning and re-hashing. We can try to maintain an array of AMQs, and rehash/redistribute load among the sub-AMQs when there is a false-positive. More work is needed to work out the details of this idea and determine if it is viable, since rehashing/redistributing is often expensive and can be difficult without the original value of the element. Nonetheless, this type of idea could prove interesting for a future data structure.

5.4 Adaptive Bloom Filters to Speed Up Databases

Many existing databases, for example PostgreSQL, use bloom filters to speed up disk lookups. In these applications, many queries are run repeatedly to gather real-time data and it is possible that some of the queries can repeatedly trigger the same group of false-positives. Is it possible to use a practical adaptive AMQ, such as the adaptive cuckoo filter, to speed up query lookup times? We can benchmark this using databases from Harvard courses or on a fork of the PostgreSQL database.

6 Conclusion

After having discussed the designs of both the Adaptive Cuckoo Filter and the Broom Filter, there seems to be a general strategy that can be applied to achieve a fairly strong amount of adaptivity in an AMQ without a prohibitive increase in the amount of local space used. We start by taking a traditional AMQ that uses discrete representations of its members, like the fingerprints used in the Quotient and Cuckoo filters, so that we have the ability to change the representations of individual elements in the filter and use that as our local representation. We then add a second structure which is updated and accessed in tandem with underlying set. This second structure then needs to be able to supply sufficient information for the local representation to change the representations of individual elements in order to correct for false-positives. As shown by Bender et al.'s proof [2], this second structure is critical to keeping the size of the local representation near optimal.

As shown by the Broom Filter, this strategy can be taken to the extreme where the method of adaptation goes as far as to guarantee that two elements will never collide more than once within a given phase. Alternatively, we can use a slightly more relaxed method of adaptation as seen in the Adaptive Cuckoo Filter where, once two elements have collided, they are very unlikely to collide again. After having attempted to implement the rigorous method of the Broom Filter, we have an increased appreciation for the simplicity of Adaptive Cuckoo Filter's design, even if it does not lend itself as nicely to rigorous analysis.

It is also worth noting that this strategy for adding adaptivity does not lend itself well to all traditional AMQs. For example, consider a traditional Bloom Filter where each element in the filter is represented by k bits in a table that are assigned by k hash functions. Because the representations of multiple elements can overlap, updating a single one without changing the others

becomes problematic and potentially prohibitive to adaptivity. That said, filters that make use of discrete representations for each element tend to make better use of space in the first place through methods like quotienting in the Quotient and Broom Filters.

References

[1] Arbitman, Y., Naor, M., and Segev, G. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. *CoRR abs/0912.5424* (2009).

[2] Bender, M. A., Farach-Colton, M., Goswami, M., Johnson, R., McCauley, S., and Singh, S. Bloom filters, adaptivity, and the dictionary problem. *CoRR abs/1711.01616* (2017).

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

[4] Bruck, J., Gao, J., and Jiang, A. Weighted bloom filter. In *IEEE International Symposium on Information Theory* (08 2006), pp. 2304 – 2308.

[5] Chazelle, B., Kilian, J., Rubinfeld, R., and Tal, A. The bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2004), SODA '04, Society for Industrial and Applied Mathematics, pp. 30–39.

[6] Donnet, B., Baynat, B., and Friedman, T. Retouched bloom filters: Allowing networked applications to trade off selected false positives against false negatives. In *Proceedings of the 2006 ACM CoNEXT Conference* (New York, NY, USA, 2006), CoNEXT '06, ACM, pp. 13:1–13:12.

[7] Fan, B., Andersen, D. G., Kaminsky, M., and Mitzenmacher, M. D. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2014), CoNEXT '14, ACM, pp. 75–88.

[8] Mitzenmacher, M., Pontarelli, S., and Reviriego, P. Adaptive cuckoo filters. *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)* (2018), 36–47.

[9] Mitzenmacher, M., and Upfal, E. *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*, 2nd ed. Cambridge University Press, New York, NY, USA, 2017.

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