2018 IEEE 59th Annual Symposium on Foundations of Computer Science

Bloom Filters, Adaptivity, and the Dictionary Problem

Michael A. Bender*, Martín Farach-Colton†, Mayank Goswami‡, Rob Johnson§, Samuel McCauley¶, and Shikha Singh¶

* Stony Brook University, Stony Brook, NY 11794-2424 USA. Email: bender@cs.stonybrook.edu.
† Rutgers University, Piscataway, NJ 08856 USA. Email: martin@farach-colton.com.
‡ Queens College, CUNY, NY 11367 USA. Email: mayank.goswami@qc.cuny.edu.
§ VMware Research, Creekside F, 3425 Hillview Ave, Palo Alto, CA 94304 USA. Email: robj@vmware.com.
¶ Wellesley College, Wellesley, MA 02481 USA. Email: {smccaule, shikha.singh}@wellesley.edu.

***Abstract*—An approximate membership query data structure (AMQ)—such as a Bloom, quotient, or cuckoo filter—maintains a compact, probabilistic representation of a set $\mathcal{S}$ of keys from a universe $\mathcal{U}$. It supports lookups and inserts. Some AMQs also support deletes. A query for $x \in \mathcal{S}$ returns PRESENT. A query for $x \notin \mathcal{S}$ returns PRESENT with a tunable *false-positive probability* ε, and otherwise returns ABSENT.**

AMQs are widely used to speed up dictionaries that are stored remotely (e.g., on disk or across a network). The AMQ is stored locally (e.g., in memory). The remote dictionary is only accessed when the AMQ returns PRESENT. Thus, the primary performance metric of an AMQ is how often it returns ABSENT for negative queries.

Existing AMQs offer weak guarantees on the number of false positives in a sequence of queries. The false-positive probability ε holds only for a single query. It is easy for an adversary to drive an AMQ's false-positive rate towards 1 by simply repeating false positives.

This paper shows what it takes to get strong guarantees on the number of false positives. We say that an AMQ is *adaptive* if it guarantees a false-positive probability of ε for every query, *regardless of answers to previous queries*.

We establish upper and lower bounds for adaptive AMQs. Our lower bound shows that it is impossible to build a small adaptive AMQ, even when the AMQ is immediately told whenever a query is a false positive. On the other hand, we show that it is possible to maintain an AMQ that uses the same amount of local space as a non-adaptive AMQ (up to lower order terms), performs all queries and updates in constant time, and guarantees that each negative query to the dictionary accesses remote storage with probability ε, independent of the results of past queries. Thus, we show that adaptivity can be achieved effectively for free.

***Keywords*-Bloom filters; approximate membership query data structures; adaptive data structures; dictionary data structures**

I. INTRODUCTION

An approximate membership query data structure (AMQ)—such as a Bloom [4, 5], quotient [3, 18], single hash [17], or cuckoo [12] filter—maintains a compact, probabilistic representation of a set $\mathcal{S}$ of keys from a universe $\mathcal{U}$. It supports lookups and inserts. Some AMQs also support deletes. A positive query for $x \in \mathcal{S}$ returns PRESENT. A negative query for $x \notin \mathcal{S}$ returns PRESENT with a tunable ***false-positive probability*** ε, and otherwise returns ABSENT.

AMQs are used because they are small. An optimal AMQ can encode a set $\mathcal{S} \subseteq \mathcal{U}$, where $|\mathcal{S}| = n$ and $|\mathcal{U}| = u$, with a false-positive probability ε using $\Theta(n \log(1/\varepsilon))$ bits [6]. In contrast, an error-free representation of $\mathcal{S}$ takes $\Omega(n \log u)$ bits.

One of the main uses of AMQs is to speed up dictionaries [5, 8, 10, 11, 13, 20, 21]. Often, there is not enough local storage (e.g., RAM) to store the dictionary's internal state, $\mathbf{D}$. Thus, $\mathbf{D}$ must be maintained remotely (e.g., on-disk or across a network), and accesses to $\mathbf{D}$ are expensive. By maintaining a local AMQ for the set $\mathcal{S}$ of keys occurring in $\mathbf{D}$, the dictionary can avoid accessing $\mathbf{D}$ on most negative queries: if the AMQ says that a key is not in $\mathcal{S}$, then no query to $\mathbf{D}$ is necessary.

Thus, the primary performance metric of an AMQ is how well it enables a dictionary to avoid these expensive accesses to $\mathbf{D}$. The fewer false positives an AMQ returns on a sequence of queries, the more effective it is.

AMQ guarantees. Existing AMQs offer weak guarantees on the number of false positives they will return for a sequence of queries. The false-positive probability of ε holds only for a single query. It does not extend to multiple queries, because queries can be correlated. It is easy for an adversary to drive an AMQ's false-positive rate towards 1 by simply repeating false-positives.

Even when the adversary is oblivious, i.e., it selects n queries without regard to the results of previous queries, existing AMQs have weak guarantees. With probability ε, a random query is a false positive, and repeating it n times results in a false-positive rate of 1. Thus, even when the adversary is oblivious, existing AMQs can have $O(\varepsilon n)$ false positives in expectation but not with high probability. This distinction has implications: Mitzenmacher et al. [15] show that on network traces,

2575-8454/18/$31.00 ©2018 IEEE
DOI 10.1109/FOCS.2018.00026

existing AMQs are suboptimal because they do not adapt to false positives.

Adaptive AMQs. We define an ***adaptive AMQ*** to be an AMQ that returns PRESENT with probability at most ε for every negative query, *regardless of answers to previous queries*. For a dictionary using an adaptive AMQ, *any* sequence of n negative queries will result in $O(\varepsilon n)$ false positives, with high probability. This gives a strong bound on the number of (expensive) negative accesses that the dictionary will need to make to $\mathbf{D}$. This is true even if the queries are selected by an adaptive adversary.

Several attempts have been made to move towards adaptivity (and beyond oblivious adversaries). Naor and Yogev [16] considered an adaptive adversary that tries to increase the false-positive rate by discovering collisions in the AMQ's hash functions, but they explicitly forbade the adversary from repeating queries. Chazelle et al. [7] introduced bloomier filters, which can be updated to specify a white list, which are elements in $\mathcal{U} - \mathcal{S}$ on which the AMQ may not answer PRESENT. However, bloomier filters are space efficient only when the white list is specified in advance, which makes them unsuitable for adaptivity. Mitzenmacher et al. [15] proposed an elegant variant of the cuckoo filter that stores part of the AMQ locally and part of it remotely in order to try to achieve adaptivity. They empirically show that their data structure helps maintain a low false-positive rate against queries that have temporal correlation.

However, no existing AMQ is provably adaptive.

Feedback, local AMQs, and remote representations. When an AMQ is used to speed up a dictionary, the dictionary always detects which are the AMQ's false positives and which are the true positives. Thus, the dictionary can provide this feedback to the AMQ. This feedback is free because it does not require any additional accesses to $\mathbf{D}$ beyond what was used to answer the query.

In this paper we show that, even with this feedback, it is impossible to construct an adaptive AMQ that uses less than $\Omega(\min\{n \log\log u, n \log n\})$ bits of space; see Theorem 5. That is, even if an AMQ is told which are the true and false positives, adaptivity requires large space.

This lower bound would appear to kill the whole idea of adaptive AMQs, since one of the key ideas of an AMQ is to be small enough to fit in local storage. Remarkably, efficient adaptivity is still achievable.

The way around this impasse is to partition an AMQ's state into a small local state $\mathbf{L}$ and a larger remote state $\mathbf{R}$. The AMQ can still have good performance, provided it access the remote state infrequently.

We show how to make an adaptive AMQ that consumes no more local space than the best non-adaptive AMQ (and much less than a Bloom filter). We call this data structure a ***broom filter*** (because it cleans up its mistakes). The broom filter accesses $\mathbf{R}$ only when the AMQ receives feedback that it returned a false positive.

When used to filter accesses to a remote dictionary $\mathbf{D}$, the AMQ's accesses to $\mathbf{R}$ are "free"—i.e. they do not asymptotically increase the number of accesses to remote storage—-because the AMQ access $\mathbf{R}$ only when the dictionary accesses $\mathbf{D}$.

Our lower bound shows that partitioning is essential to creating a space-efficient adaptive AMQ. Indeed, the adaptive cuckoo filter of Mitzenmacher et al. [15] also partitions its state into local and remote components, but it does not have the strong theoretical adaptivity guarantees of the broom filter.

The local component, $\mathbf{L}$, of the broom filter is itself a non-adaptive AMQ plus $O(n)$ bits for adaptivity. The purpose of $\mathbf{R}$ is to provide a little more information to help $\mathbf{L}$ adapt.

Thus, we have a dual view of adaptivity that helps us interpret the upper and lower bounds. The local representation $\mathbf{L}$ is an AMQ in its own right. The remote representation $\mathbf{R}$ is an "oracle" that gives extra feedback to $\mathbf{L}$ whenever there is a false positive. Because $\mathbf{R}$ is simply an oracle, all the heavy lifting is in the design of $\mathbf{L}$. In the broom filter, $\mathbf{R}$ enables $\mathbf{L}$ to identify an element $y \in \mathcal{S}$ that triggered the false positive.

Putting these results together, we pinpoint how much information is needed for an adaptive AMQ to update its local information. The lower bound shows that simply learning if the query is a false positive is not sufficient. But if this local information is augmented with asymptotically free remote lookups, then adaptivity is achievable.

A note on optimality. The broom filter dominates existing AMQs in all regards. Its local state by itself is an optimal conventional AMQ: it uses optimal space up to lower-order terms, and supports queries and updates in constant time with high probability. Thus the remote state is only for adaptivity. For comparison, a Bloom filter has a lookup time of $O(\log \frac{1}{\varepsilon})$, the space is suboptimal, and the filter does not support deletes. More recent AMQs [3, 12, 17, 18] also fail to match the broom filter on one or more of these criteria, even leaving aside adaptivity. Thus, we show that adaptivity has no cost.

II. PRELIMINARIES

We begin by defining the operations that our AMQ supports. These operations specify when it can access its local and remote states, when it gets to update its states, how it receives feedback, and basic correctness requirements (i.e., no false negatives). We define performance constraints (i.e., false-positive rates) later.

game ADAPTIVITY-GAME($\mathcal{A}, n, \varepsilon$)
  $\mathcal{O} \leftarrow$ SETUP(n, ε)
  $x' \leftarrow \mathcal{A}^{\mathcal{O}}(n, \varepsilon)$
  $b \leftarrow \mathcal{O}$.LOOKUP(x')
  return $(b =$ PRESENT$) \wedge (x' \notin \mathcal{O}.\mathcal{S})$

function SETUP(n,ε)
  $\mathcal{O}.\rho \xleftarrow{\$} \{0,1\}^{\mathbb{N}}$
  $\mathcal{O}.\mathcal{S} \leftarrow \emptyset$
  $(\mathcal{O}.\mathbf{L}, \mathcal{O}.\mathbf{R}) \leftarrow$ INIT($n, \varepsilon, \mathcal{O}.\rho$)
  return $\mathcal{O}$

method $\mathcal{O}$.LOOKUP(x)
  $(\mathbf{L}, b) \leftarrow$ LOOKUP($\mathbf{L}, x, \rho$)
  if $(b =$ PRESENT$) \wedge (x \notin \mathcal{S})$ **then**
    $(\mathbf{L}, \mathbf{R}) \leftarrow$ ADAPT($(\mathbf{L}, \mathbf{R}), x, \rho$)
  return b

method $\mathcal{O}$.INSERT(x)
  if $|\mathcal{S}| < n \wedge x \notin \mathcal{S}$ **then**
    $(\mathbf{L}, \mathbf{R}) \leftarrow$ INSERT($(\mathbf{L}, \mathbf{R}), x, \rho$)
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{x\}$

method $\mathcal{O}$.DELETE(x)
  if $x \in \mathcal{S}$ **then**
    $(\mathbf{L}, \mathbf{R}) \leftarrow$ DELETE($(\mathbf{L}, \mathbf{R}), x, \rho$)
    $\mathcal{S} \leftarrow \mathcal{S} \setminus \{x\}$

Fig. 1: Definition of the game between an adaptive AMQ and an adversary $\mathcal{A}$. The adversary gets n, ε, and oracular access to $\mathcal{O}$, which supports three operations: $\mathcal{O}$.LOOKUP, $\mathcal{O}$.INSERT, and $\mathcal{O}$.DELETE. The adversary wins if, after interacting with the oracle, it outputs an element x' that is a false positive of the AMQ. An AMQ is adaptive if there exists a constant $\varepsilon < 1$ such that no adversary wins with probability greater than ε.

Definition 1 (***AMQs***)**.** *An approximate membership query data structure (AMQ) consists of the following deterministic functions. Here ρ denotes the AMQ's private infinite random string, $\mathbf{L}$ and $\mathbf{R}$ denote its private local and remote state, respectively, $\mathcal{S}$ represents the set of items that have been inserted into the AMQ more recently than they have been deleted, n denotes the maximum allowed set size, and ε denotes the false-positive probability.*

- INIT$(n, \varepsilon, \rho) \longrightarrow (\mathbf{L}, \mathbf{R})$. INIT *creates an initial state* $(\mathbf{L}, \mathbf{R})$.
- LOOKUP$(\mathbf{L}, x, \rho) \longrightarrow (\mathbf{L}', b)$. *For* $x \in \mathcal{U}$, LOOKUP *returns a new local state* $\mathbf{L}'$ *and* $b \in \{$PRESENT, ABSENT$\}$. *If* $x \in \mathcal{S}$, *then* $b =$ PRESENT *(i.e., AMQs do not have false negatives).* LOOKUP *does not get access to* $\mathbf{R}$.
- INSERT$((\mathbf{L}, \mathbf{R}), x, \rho) \longrightarrow (\mathbf{L}', \mathbf{R}')$. *For* $|\mathcal{S}| < n$ *and* $x \in \mathcal{U} \setminus \mathcal{S}$, INSERT *returns a new state* $(\mathbf{L}', \mathbf{R}')$. INSERT *is not defined for* $x \in \mathcal{S}$. DELETE *is defined analogously.*
- ADAPT$((\mathbf{L}, \mathbf{R}), x, \rho) \longrightarrow (\mathbf{L}', \mathbf{R}')$. *For* $x \notin \mathcal{S}$ *such that* LOOKUP$(\mathbf{L}, x, \rho) =$ PRESENT, ADAPT *returns a new state* $(\mathbf{L}', \mathbf{R}')$.

An AMQ is ***local*** if it never reads or writes $\mathbf{R}$; an AMQ is ***oblivious*** if ADAPT is the identity function on $(\mathbf{L}, \mathbf{R})$. Bloom filters, cuckoo filters, etc, are local oblivious AMQs.

False positives and adaptivity. We say that x is a ***false positive*** of AMQ state $(\mathbf{L}, \mathbf{R})$ if $x \notin \mathcal{S}$ but LOOKUP$(\mathbf{L}, x, \rho)$ returns PRESENT.

We define an AMQ's false-positive rate using the adversarial game in Figure 1. In this game, we give the adversary access to the AMQ via an oracle $\mathcal{O}$. The oracle keeps track of the set $\mathcal{S}$ being represented by the AMQ and ensures that the adversary respects the limits of the AMQ (i.e., never overloads the AMQ, inserts an item that is already in $\mathcal{S}$, or deletes an item that is not currently in $\mathcal{S}$). The adversary can submit queries and updates to the oracle, which applies them to the AMQ and calls ADAPT whenever LOOKUP returns a false positive. The adversary cannot inspect the internal state of the oracle. ADAPTIVITY-GAME outputs TRUE iff the adversary wins, i.e., if, after interacting with the oracle, $\mathcal{A}$ outputs a false positive x' of the final state of the AMQ.

The ***static false-positive rate*** is the probability, taken over the randomness of the AMQ, that a particular $x \in \mathcal{U} \setminus \mathcal{S}$ is a false positive of the AMQ. This is equivalent to the probability that an adversary that never gets to query the AMQ is able to output a false positive. We formalize this as follows. An adversary is a ***single-query adversary*** if it never invokes $\mathcal{O}$.LOOKUP. We call this "single-query" because there is still an invocation of $\mathcal{O}$.LOOKUP at the very end of the game, when ADAPTIVITY-GAME tests whether x' is a false positive.

Definition 2. *An AMQ supports* ***static false-positive rate*** $\boldsymbol{\varepsilon}$ *if for all n and all single-query adversaries $\mathcal{A}$,*

$$\Pr[\text{ADAPTIVITY-GAME}(\mathcal{A}, n, \varepsilon) = \text{TRUE}] \leq \varepsilon.$$

Definition 3. *An AMQ supports* ***sustained false-positive rate*** $\boldsymbol{\varepsilon}$ *if for all n and all adversaries $\mathcal{A}$,*

$$\Pr[\text{ADAPTIVITY-GAME}(\mathcal{A}, n, \varepsilon) = \text{TRUE}] \leq \varepsilon.$$

An AMQ is ***adaptive*** if there exists a constant $\varepsilon < 1$ such that the AMQ guarantees a sustained false-positive rate of at most ε.

The following lemma shows that, since an adaptive AMQ accesses its remote state rarely, it must use as much local space as a local AMQ.

Lemma 4. *Any adaptive AMQ must have a local representation $\mathbf{L}$ of size at least $n \log(1/\varepsilon)$.*

Proof: Consider an adaptive AMQ with a sustained false positive rate of ε. Consider the local state $\mathbf{L}'$ at the time when the adversary provides x'. By the definition of sustained-false positive rate, $\mathbf{L}'$ must have a static false positive rate of at most ε. Thus, by the Bloom-filter lower bound [6, 14], $\mathbf{L}'$ must have size at least $n \log(1/\varepsilon)$. ■

Cost model. We measure AMQ performance in terms of the RAM operations on $\mathbf{L}$ and in terms of the number of updates and queries to the remote representation $\mathbf{R}$. We measure these three quantities (RAM operations, remote updates, and remote queries) separately.

We follow the standard practice of analyzing AMQ performance in terms of the AMQ's maximum capacity, n.We assume a word size $w = \Omega(\log u)$ in most of the paper. For simplicity of presentation, we assume that $u = \text{poly}(n)$ but our results generalize.

Hash functions. We assume that the adversary cannot find a never-queried-before element that is a false positive of the AMQ with probability greater than ε. Ideal hash functions have this property for arbitrary adversaries. If the adversary is polynomially bounded, one-way functions are sufficient to prevent them from generating new false positives [16].

III. RESULTS

We prove the following lower bound on the space required by an AMQ to maintain adaptivity.

Theorem 5. *Any adaptive 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 whp to maintain any constant sustained false-positive rate $\varepsilon < 1$.*

Together, Definition 1, Theorem 5 and Lemma 4 suggest what an optimal adaptive AMQ should look like. Lemma 4 says that $\mathbf{L}$ must have at least $n \log(1/\varepsilon)$ bits. Theorem 5 implies that any adaptive AMQ with $\mathbf{L}$ near this lower bound must make remote accesses.

A consequence of Definition 1 is that AMQs access $\mathbf{R}$ only when the system is accessing $\mathbf{D}$, so, if an AMQ performs $O(1)$ updates of $\mathbf{R}$ for each update of $\mathbf{D}$ and $O(1)$ queries to $\mathbf{R}$ for each query to $\mathbf{D}$, then accesses to $\mathbf{R}$ are asymptotically free. Thus, our target is an AMQ that has approximately $n \log(1/\varepsilon)$ bits in $\mathbf{L}$ and performs $O(1)$ accesses to $\mathbf{R}$ per update and query.

Our upper bound result is such an adaptive AMQ:

Theorem 6. *There exists an adaptive AMQ—the broom filter—that, for any sustained false-positive rate ε and maximum capacity n, attains the following performance:*

- ***Constant local work:*** *$O(1)$ operations for inserts, deletes, and lookups w.h.p.*
- ***Near optimal local space:*** *$(1+o(1))n \log \frac{1}{\varepsilon} + O(n)$ local space w.h.p.*[1]
- ***Asymptotically optimal remote accesses:*** *$O(1)$ updates to $\mathbf{R}$ for each delete to $\mathbf{D}$; $O(1)$ updates to $\mathbf{R}$ with probability at most ε for each insertion to $\mathbf{D}$; $O(1)$ updates to $\mathbf{R}$ for each false positive.*

[1]All logarithms in this paper are base 2 unless specified otherwise.

The local component of the broom filter is, itself, an AMQ with performance that strictly dominates the Bloom Filter, which requires $(\log e) n \log(1/\varepsilon)$ space and $O(\log(1/\varepsilon))$ update time [4], and matches (up to lower-order terms) or improves upon the performance of more efficient AMQs [3, 12, 17, 19].

Since $\mathbf{L}$ contains an AMQ, one way to interpret our results is that a small local AMQ cannot be adaptive if it is only informed of true positives versus false positives, but it can adapt if it is given a little more information. In the case of the broom filter, it is given the element of S causing a false positive, that is, the element in S that has a hash function collision with the query, as we see next.

IV. BROOM FILTERS: DEFINING FINGERPRINTS

The broom filter is a single-hash-function AMQ [3, 12, 17], which means that it stores fingerprints for each element in $\mathcal{S}$. In this section, we begin our proof of Theorem 6 by describing what fingerprints we store and how they establish the sustained false-positive rate of broom filters. In Section V, we show how to maintain the fingerprints space-efficiently and in $O(1)$ time.

A. *Fingerprints*

The broom filter has a hash function $h : \mathcal{U} \to \{0, \ldots n^c\}$ for some constant $c \geq 4$. Storing an entire hash takes $c \log n$ bits, which is too much space—we can only afford approximately $\log(1/\varepsilon)$ bits per element. Instead, for set $\mathcal{S} = \{y_1, y_2, \ldots, y_n\}$, the broom filter stores a set of ***fingerprints*** $\mathcal{P} = \{p(y_1), p(y_2), \ldots, p(y_n)\}$, where each $p(y_i)$ is a ***prefix*** of $h(y_i)$, denoted $p(y_i) \sqsubseteq h(y_i)$.

Queries. A query for x returns PRESENT iff there exists a $y \in \mathcal{S}$ such that $p(y) \sqsubseteq h(x)$. The first $\log n + \log(1/\varepsilon)$ bits of a fingerprint comprise the ***baseline fingerprint***, which is subdivided as in a quotient filter [3, 18]. In particular, the first $q = \log n$ bits comprise the ***quotient***, and the next $r = \log(1/\varepsilon)$ bits the ***remainder***. The remaining bits (if any) comprise the ***adaptivity bits***.

Using the parts of the fingerprint. The baseline fingerprint is long enough to guarantee that the false-positive rate is at most ε. We add adaptivity bits to fix false positives, in order to achieve a sustained false-positive rate of ε. Adaptivity bits are also added during insertions. We maintain the following invariant:

Invariant 7. *No fingerprint is a prefix of another.*

By this invariant, a query for x can match at most one $p(y) \in \mathcal{P}$. As we will see, we can fix a false positive by adding adaptivity bits to the single $p(y)$, for which $p(y) \sqsubseteq h(x)$. Thus, adding adaptivity bits during insertions reduces the number of adaptivity bits added during false positives, which will allow us to achieve $O(1)$ work and remote accesses for each operation.

Shortly we will give a somewhat subtler reason why adaptivity bits are added during insertions—in order to defeat deletion-based timing attacks on the sustained false-positive rate.

Maintaining the fingerprints. Here we describe what the broom filter does on a call to ADAPT. In this section we drop $(\mathbf{L}, \mathbf{R})$ and ρ from the notation for simplicity.

We define a subroutine of ADAPT which we call EXTEND$(x, \mathcal{P})$. This function is used to maintain Invariant 7 and to fix false positives.

Observe that on a query x there exists at most one y for which $p(y) \sqsubseteq h(x)$, by Invariant 7. If such a y exists, the EXTEND$(x, \mathcal{P})$ operation modifies the local representation by appending adaptivity bits to $p(y)$ until $p(y) \not\sqsubseteq h(x)$. (Otherwise, EXTEND$(x, \mathcal{P})$ does nothing.) Thus, EXTEND performs remote accesses to REVLOOKUP$_{\mathcal{P}}$, where REVLOOKUP$_{\mathcal{P}}(x)$ returns the (unique) $y \in \mathcal{S}$ such that $p(y) \sqsubseteq h(x)$. REVLOOKUP$_{\mathcal{P}}$ is a part of $\mathbf{R}$, and can be implemented using a dictionary.

We can define ADAPT(x) as follows:

- **Queries.** If a query x is a false positive, we call EXTEND$(x, \mathcal{P})$, after which x is no longer a false positive.
- **Insertions.** When inserting an element x into $\mathcal{S}$, we first check if Invariant 7 is violated, that is, if there exists a $y \in \mathcal{S}$ such that $p(y) \sqsubseteq h(x)$.[2] If so, we call EXTEND$(x, \mathcal{P})$, after which $p(y) \not\sqsubseteq h(x)$. Then we add the shortest prefix of $h(x)$ needed to maintain Invariant 7.
- **Deletions.** Deletions do not make calls to ADAPT. We defer the details of the deletion operation until after we discuss how to reclaim bits introduced by ADAPT. For now we note the naïve approach of deleting an element's fingerprint is insufficient to guarantee a sustained false-positive rate.

B. Reclaiming Bits

Each call to ADAPT adds bits, and so we need a mechanism to remove bits. An amortized way to reclaim bits is to rebuild the broom filter with a new hash function every $\Theta(n)$ calls to ADAPT.

This change from old to new hash function can be deamortized without losing a factor of 2 on the space. We keep two hash functions, h_a and h_b; any element y greater than frontier z is hashed according to h_a, otherwise, it is hashed according to h_b. At the beginning of a ***phase***, frontier $z = -\infty$ and all elements are hashed according to h_a. Each time we call ADAPT, we delete the smallest constant $c > 1$ elements in $\mathcal{S}$ greater than z and reinsert them according to h_b. (Finding these elements requires access to $\mathbf{R}$; again this can be efficiently implemented using standard data structures.) We then set z to be the value of the largest reinserted element. When z reaches the maximum element in $\mathcal{S}$, we begin a new phase by setting $h_a = h_b$, picking a new h_b, and resetting $z = -\infty$. We use this frontier method for deamortization so that we know which hash function to use for queries: lookups on $x \leq z$ use h_b and those on $x > z$ use h_a.

Observation 8. *A hash function times out after $O(n)$ calls to* ADAPT.

Because every call to ADAPT introduces an expected constant number of adaptivity bits, we obtain:

Lemma 9. *In any phase,* ADAPT *introduces $O(n)$ adaptivity bits into the broom filter with high probability.*

The proof of Lemma 9 follows from Chernoff bounds and is included in the full version [2].

If we did not have deletions, then Observation 8 and Lemma 9 would be enough to prove a bound on total size of all fingerprints—because adaptivity bits are removed as their hash function times out. To support deletions we introduce adaptivity bits via a second mechanism. We will show that this second mechanism also introduces a total of $O(n)$ adaptivity bits per phase.

C. Deletions and Adaptivity Bits

It is tempting to support deletions simply by removing fingerprints from $\mathcal{P}$, but this does not work. To see why, observe that false positives are eliminated by adding adaptivity bits. Removing fingerprints destroys history and reintroduces false positives. This opens up the data structure to timing attacks by the adversary.

We describe one such timing attack to motivate our solution. The adversary finds a false positive x and an element $y \in \mathcal{S}$ that collides with x. (It finds y by deleting and reinserting random elements until x is once again a false positive.) The attack then consists of repeatedly looking up x, deleting y, then inserting y. This results in a false positive on every lookup until x or y's hash function changes.

Thus, the broom filter needs to remember the history for deleted elements, since they might be reinserted. Only once y's hash function has changed can y's history be forgotten. A profligate approach is to keep the fingerprints of deleted elements as "ghosts" until the hash function changes. Then, if the element is reinserted, the adaptivity bits are already there. Unfortunately, remembering deleted elements can blow up the space by a constant factor, which we cannot afford.

[2]This step and the following assume x does not already belong to $\mathcal{S}$. If it does, we don't need to do anything during insertions.

Instead, we remember the adaptivity bits and quotient from each deleted element's fingerprint—but we forget the remainder. Only once the hash function has changed do we forget everything. This can be accomplished by including deleted elements in the strategy described in Section IV-B. (with deletions, we increase the requirement on adaptivity bits reclaimed at once to $c > 2$).

Now when a new element x gets inserted, we check whether there exists a ghost that matches $h(x)$. If so, then we give x at least the adaptivity bits of the ghost, even if this is more than needed to satisfy Invariant 7. This scheme guarantees the following:

Property 10. *If x is a false positive because it collides with y, then it cannot collide with y again until x or y's hash function times out (even if y is deleted and reinserted).*

D. *Sustained False-Positive Rate*

We now establish the sustained false-positive rate of broom filters. We begin by introducing notation:

Definition 11. *Hashes $h(x)$ and $h(y)$ have a* ***soft collision*** *when they have the same quotient. They have a* ***hard collision*** *when they have the same quotient and remainder. Hash $h(x)$ and fingerprint $p(y)$ have a* ***full collision*** *if $p(y) \sqsubseteq h(x)$.*

The hash function is fixed in this section, so we refer to x and y themselves as having (say) a soft collision, with the understanding that it is their hashes that collide.

Lemma 12. *The probability that any query has a hard collision with any of n fingerprints is at most ε.*

Proof: The probability that any query collides with a single fingerprint is $2^{-(\log n+\log(1/\varepsilon))} = \varepsilon/n$. Applying the union bound, we obtain the lemma. ■

Lemma 13. *The sustained false-positive rate of a broom filter is ε.*

Proof: We prove that on any query $x \notin S$, $\Pr[\exists y \in \mathcal{S} \mid x \text{ has a full collision with } y] \leq \varepsilon$, regardless of the previous history. Any previous query that is a negative or a true positive has no effect on the data structure. Furthermore, deletions do not increase the chance of any full collision, so we need only consider false positives and insertions, both of which induce rehashing.

We say that $x \in \mathcal{U}$ and $y \in \mathcal{S}$ are ***related at time t*** if (1) there exists $t' < t$ such that x was queried at time t' and y was in $\mathcal{S}$ at t', and (2) between t' and t, the hash functions for x and y did not change. Suppose x is queried at time t. Then, by Property 10, if x and y are related at time t, then $\Pr[x \text{ is a false positive at } t] = 0$. If x and y are not related at time t, then $\Pr[x \text{ has a full collision with } y] \leq \Pr[h(x) \text{ has a hard collision with } h(y)]$. Finally, by Lemma 12, $\Pr[x \text{ is a false positive at } t] \leq \varepsilon$. ■

E. *Space Bounds for Adaptivity Bits*

We first prove that at any time there are $O(n)$ adaptivity bits. Then we bootstrap this claim to show a stronger property: there are $\Theta(\log n)$ fingerprints associated with $\Theta(\log n)$ contiguous quotients, and these fingerprints have a total of $O(\log n)$ adaptivity bits w.h.p. (thus they can be stored in $O(1)$ machine words).

For the purposes of our proofs, we partition adaptivity bits into two classes: ***extend bits***, which are added by calls to EXTEND, and ***copy bits***, which are added on insertion due to partial matches with formerly deleted items. As some bits may be both extend and copy bits, we partition adaptivity bits by defining all the adaptivity bits in a fingerprint to be of the same type as the last bit, breaking ties in favor of extend. If an item is deleted and then reinserted, its bits are of the same type as when it first got them. (So if an item that gets extend bits is deleted and reinserted with the same adaptivity bits, then it still has extend bits.)

See [2] for the proofs of the following lemmas.

Lemma 14. *At any time, there are $O(n)$ adaptivity bits in the broom filter with high probability.*

Lemma 15. *There are $\Theta(\log n)$ fingerprints associated with a range of $\Theta(\log n)$ contiguous quotients, and these fingerprints have $O(\log n)$ total extend bits w.h.p.*

Lemma 16. *There are $\Theta(\log n)$ fingerprints associated with a range of $\Theta(\log n)$ contiguous quotients, and these fingerprints have $O(\log n)$ total adaptivity bits w.h.p.*

V. BROOM FILTERS: IMPLEMENTING FINGERPRINTS

In Section IV, we showed how to use fingerprints to achieve a sustained false-positive rate of ε. In this section we give space- and time-efficient implementations for the fingerprint operations that are specified in Section IV. We explain how we store and manipulate adaptivity bits (Section V-A), quotients (Section V-B), and remainders. We describe two variants of our data structure, because there are two ways to manage remainders, depending on whether $\log(1/\varepsilon) \leq 2 \log\log n$, the ***small-remainder case*** (Section V-C), or $\log(1/\varepsilon) > 2 \log\log n$, the ***large-remainder case*** (Section V-D).

Bit Manipulation within Machine Words. In the full version [2], we show how to implement a variety of primitives on machine words in $O(1)$ time using word-level parallelism. The upshot is that from now on, we may assume that the asymptotic complexity for any operation on the broom filter is simply the number of machine words that are touched during the operation.

A. ***Encoding Adaptivity Bits and Deletion Bits***

We store adaptivity bits separately from the rest of the fingerprint. By Lemma 16, all of the adaptivity bits in any range of $\Theta(\log n)$ quotients fit in a constant number of words. Thus, all of the searches and updates to adaptivity bits take $O(1)$ time.

B. ***Encoding Quotients***

Quotients and remainders are stored succinctly in a scheme similar to quotient filters [3, 18]; we call this high-level scheme ***quotienting***.

Quotienting stores the baseline fingerprints succinctly in an array of $\Theta(n)$ slots, each consisting of r bits. Given a fingerprint with quotient a and remainder b, we would like to store b in position a of the array. This allows us to reconstruct the fingerprint based on b's location. So long as the number of slots is not much more than the number of stored quotients, this is an efficient representation. (In particular, we will have a sublinear number of extra slots in our data structure.)

The challenge is that multiple fingerprints may have the same quotient and thus contend for the same location. Linear probing is a standard technique for resolving collisions: slide an element forward in the array until it finds an empty slot. Linear probing does not immediately work, however, since the quotient is supposed to be reconstructed based on the location of a remainder. The quotient filter implements linear probing by maintaining a small number (between 2 and 3) of metadata bits per array slot which encode the target slot for a remainder even when it is shifted to a different slot.

The standard quotient filter does not achieve constant time operations, independent of ε. This is because when the remainder length $r = \log(1/\varepsilon) = \omega(1)$, and the fingerprint is stored in a set of $\Omega(\log n)$ contiguous slots, there can be $\omega(1)$ locations (words) where the target fingerprint could be. (This limitation holds even when the quotient filter is half empty, in which case it is not even space efficient enough for Theorem 6.)

Nonetheless, the quotient filter is a good starting point for the broom filter because it allows us to maintain a multiset of baseline fingerprints subject to insertions, deletions, and queries. In particular, some queries will have a hard collision with multiple elements.[3] We need to compare the adaptivity bits of the query to the adaptivity bits of each colliding element. The quotienting approach guarantees that these adaptivity bits are contiguous, allowing us to perform multiple comparisons simultaneously using word-level parallelism. In particular, Lemma 15 ensures that the adaptivity bits for $O(\log n)$ quotients fit into $O(1)$ machine words.

[3] This is the main challenge in achieving optimality with the single-hash function bloom filters of Pagh et al. [17] or the backyard hashing construction of Arbitman et al. [1]. Instead we used techniques that permit the same element to be explicitly duplicated multiple times.

C. ***Broom Filter Design for the Small-Remainder Case***

In this section we present a data structure for the case that $r = O(\log\log n)$.

High Level Setup. Our data structure consists of a primary and a secondary level. Each level is essentially a quotient filter; however, we slightly change the insert and delete operations for the primary level in order to ensure constant-time accesses.

As in a quotient filter, the primary level consists of $n(1+\alpha)$ slots, where each slot has a remainder of size $r = \log(1/\varepsilon) = O(\log\log n)$. Parameter α denotes the subconstant extra space we leave in our data structure; thus the primary level is a quotient filter as described in Section V-B, with space parameterized by α (and with slightly modified inserts, queries, and deletes). We require $\alpha \geq \sqrt{(9r\log\log n)/\log n}$.

The secondary level consists of a quotient filter with $\Theta(n/\log n)$ slots with a different hash function h_2. Thus, an element x has two fingerprints $p_1(x)$ and $p_2(x)$. The internals of the two levels are maintained entirely independently: Invariant 7 is maintained separately for each level, and adaptivity bits do not carry over from the primary level to the secondary level.

How to Perform Inserts, Queries and Deletes. To insert $y \in \mathcal{S}$, we first try to store the fingerprint $p_1(y)$ in the primary level. This uses the technique described in Section V-B: we want to store the remainder in the slot determined by the quotient. If the slot is empty, we store the remainder of $p_1(y)$ in that slot. Otherwise, we begin using linear probing to look for an empty slot, updating the metadata bits accordingly; see [3, 18].

However, unlike in previous quotienting-based data structures, we stop our probing for an empty slot early: the data structure only continues the linear probing over $O((\log n)/r)$ slots (and thus $O(1)$ words). If all of these slots are full, the item gets stored in the secondary level. In Lemma 18 we show that it finds an empty slot in $O(1)$ words in the secondary level w.h.p.

We always attempt to insert into the primary level first. In particular, even if x is deleted from the secondary level while reclaiming bits (Section IV-B), we still attempt to insert x into the primary level first.

Queries are similar to inserts—to query for y, we calculate $p_1(y)$ and search for it in the primary level for at most $O((\log n)/r)$ slots; if this fails we calculate $p_2(y)$ and search for it in the secondary level.

Lemma 17. *With high probability, $O(n/\log^2 n)$ elements are inserted into the secondary level.*

Proof: Partition the primary level into ***primary bins*** of $(1+\alpha)(\log n)/r$ consecutive slots. An element is inserted into the secondary level only if it is inserted into a sequence of $\Omega((\log n)/r)$ full slots; for this to happen either the primary bin containing the element is full or the bin adjacent to it is full. We bound the number of full primary bins.

In expectation, each bin is (initially) hashed to by $(\log n)/r$ elements. Thus, by Chernoff bounds, the probability that a given primary bin is hashed to by at least $(1+\alpha)(\log n)/r$ elements is at most $\exp(-(\alpha^2 \log n)/(3r)) \leq 1/\log^3 n$.

Thus, in expectation, $n/\log^3 n$ primary bins are full. Since these events are negatively correlated, we can use Chernoff bounds, and state that $O(n/\log^3 n)$ primary bins are full with high probability.

Each primary bin is hashed to by $O(\log n)$ elements in expectation (even fewer, in fact). Using Chernoff, each primary bin is hashed to by $O(\log n)$ elements w.h.p.

Putting the above together, even if all $O(\log n)$ elements hashed into any of the $O(n/\log^3 n)$ overflowing primary bins (or either adjacent bin) are inserted into the secondary level, we obtain the lemma. ■

Lemma 18. *With high probability, all items in the secondary level are stored at most $O(\log n/r)$ slots away from their intended slot.*

Proof: Partition the secondary level into ***secondary bins*** of $\Theta(\log n/r)$ consecutive slots. Thus, there are $\Theta(nr/\log^2 n)$ secondary bins. The lemma can only be violated if one of these bins is full.

By Lemma 17, we are inserting $O(n/\log^2 n)$ elements into these bins. By classical balls and bins analysis, because there are more bins than balls, the secondary bin with the most balls has $O((\log n)/\log\log n) = O((\log n)/r)$ elements with high probability. Thus, no secondary bin ever fills up with high probability. ■

Performance. The $O(1)$ lookup time follows by definition in the primary level, and by Lemma 18 in the secondary level. The total space of the primary level is $O((1+\alpha)n\log(1/\varepsilon)) + O(n)$, and the total space of the second level is $O((n\log(1/\varepsilon))/\log n)$. We guarantee adaptivity using the ADAPT function defined in Section IV, which makes $O(1)$ remote memory accesses per insert and false positive query.

D. Broom Filter for Large Remainders

In this section we present a data structure for the the large-remainder case, $\log(1/\varepsilon) > 2\log\log n$. Large remainders are harder to store efficiently since only a small number can fit in a machine word. E.g., we are no longer guaranteed to be able to store the remainders from all hard collisions in $O(1)$ words w.h.p.

However, large remainders also have advantages. We are very likely to be able to search using only a small portion of the remainder—a portion small enough that many can be packed into $O(1)$ words. In particular, we can "peel off" the first $2\log\log n$ bits of the remainder, filter out collisions just based on those bits, and we are left with few remaining potential collisions. We call these ***partial collisions***.

So we have an initial check for uniqueness, then a remaining check for the rest of the fingerprint. This allows us to adapt the small-remainder case to handle larger remainders without a slowdown in time.

Data structure description. As before, our data structure consists of two parts. We refer to them as the ***primary level*** and the ***backyard***. This notation emphasizes the structural difference between the two levels and the relationship with backyard hashing [1]. Unlike the small-remainder case, we use only a single hash function.

The primary level consists of two sets of slots: ***signature slots*** of size $2\log\log n$, and ***remainder slots*** of size $r - 2\log\log n$. As in Section V-C, the number of remainder slots is $(1+\alpha)n$ and the number of signature slots is $(1+\alpha)n$, where $\alpha \geq \sqrt{18\log^2\log n/\log n}$. Because the appropriate slot is found while traversing the signature slots, we only need to store metadata bits for the signature slots; they can be omitted for the remainder slots. The signature slots are stored contiguously; thus $O(\log n/\log\log n)$ slots can be probed in $O(1)$ time.

Each item is stored in the same remainder slot as in the normal quotient filter (see Subsection V-B). The signature slots mirror the remainder slots; however, only the first $2\log\log n$ bits of the remainder are stored, the rest are stored in the corresponding remainder slot.

The primary level. To insert an element y, we first try to insert $p(y)$ in the primary level. We find the signature slot corresponding to the quotient of $p(y)$. We then search through at most $O(\log n/\log\log n)$ signatures to find a partial collision (a matching signature) or an empty slot. We use metadata bits as usual—the metadata bits guarantee that we only search through signatures that have a soft collision with $p(y)$.

If there is a partial collision—a signature that matches the first $2\log\log n$ bits of the remainder of $p(y)$—we insert $p(y)$ into the backyard. If there is no empty slot, we insert $p(y)$ into the backyard. If we find an empty slot but do not find a partial collision, we insert $p(y)$ into the empty slot; this means that we insert the signature into the empty signature slot, and insert the full remainder of $p(y)$ into the corresponding remainder slot. We update the metadata bits of the signature slots as in [3, 18].

Querying for an element x proceeds similarly. In the primary level, we find the signature slot corresponding to the quotient of $p(x)$. We search through $O(\log n/\log\log n)$ slots for a matching signature. If we find a matching signature, we look in the corresponding remainder slot to see if we have a hard collision; if so we return PRESENT. If we do not find a matching signature, or if the corresponding remainder slot does not have a hard collision, we search for $p(x)$ in the back yard.

The back yard. The back yard is a compact hash table that can store $O(n/\log n)$ elements with $O(1)$ worst-case insert and delete time [1, 9]. When we store an element y in the back yard, we store its entire hash $h(y)$. Thus, w.h.p. there are no collisions in the back yard. Since the back yard has a capacity for $\Theta(n/\log n)$ elements, and each hash has size $\Theta(\log n)$, the back yard takes up $\Theta(n)$ bits, which is a lower-order term.

Lemma 19. *The number of elements stored in the back yard is $O(n/\log^2 n)$ with high probability.*

VI. A Lower Bound on Adaptive AMQs

In this section, we show that an AMQ cannot maintain adaptivity along with space efficiency. More formally, we show that any adaptive AMQ must use $\Omega(\min\{n \log n, n \log\log u\})$ bits. This means that if an AMQ is adaptive and the size of $\mathbf{L}$ is $o(\min\{n \log n, n \log\log u\})$ bits, then it must access $\mathbf{R}$. The proof itself does not distinguish between bits stored in $\mathbf{L}$ or $\mathbf{R}$. For convenience, we show that the lower bound holds when all bits are stored in $\mathbf{L}$; this is equivalent to lower bounding the bits stored in $\mathbf{L}$ and $\mathbf{R}$.

Interestingly, a similar lower bound was studied in the context of Bloomier filters [7]. In this section, we generalize this bound to all adaptive AMQ strategies.

A. Notation and Adversary Model

We begin by further formalizing our notation and defining the adversary used in the lower bound. We fix n and ε and drop them from most notation. We use BUILD$(\mathcal{S}, \rho)$ to denote the state that results from calling INIT(n, ε, ρ) followed by INSERT(x, ρ) for each $x \in \mathcal{S}$ (in lexicographic order).

Adversary Model. The adversary does not have access to the AMQ's internal randomness ρ, or any internal state $\mathbf{L}$ of the AMQ. The adversary can only issue a query x to the AMQ and only learns the AMQ's output—PRESENT or ABSENT—to query x.

The goal of the adversary is to adaptively generate a sequence of $O(n)$ queries and force the AMQ to either use too much space or to fail to satisfy a sustained false-positive rate of ε.

Let $\varepsilon_0 = \max\{1/n^{1/4}, (\log^2 \log u)/\log u\}$. Our lower bound is $m = |\mathbf{L}| = \Omega(n \log 1/\varepsilon_0)$. Note that $\varepsilon_0 \leq \varepsilon$; otherwise the classic AMQ lower bound of $m \geq n \log 1/\varepsilon$ [6, 14] is sufficient to prove Theorem 5. One can think of ε_0 as a maximum bound on the effective false positive rate—how often the AMQ encounters elements that need fixing.

Attack Description. First, the adversary chooses a set $\mathcal{S}$ of size n uniformly at random from $\mathcal{U}$. Then, the attack proceeds in rounds. The adversary selects a set Q of size n uniformly at random from $\mathcal{U} - \mathcal{S}$. Starting from Q, in each round, he queries the elements that were false positives in the previous round. To simplify analysis, we assume that the adversary orders his queries in lexicographic order. Let FP$_i$ be the set of queries that are false positives in round $i \geq 1$. The attack:

1) In the first round, the adversary queries each element of Q.
2) In round $i > 1$, if $|\text{FP}_{i-1}| > 0$, the adversary queries each element in FP$_{i-1}$; otherwise the attack ends.

Classifying False Positives. The crux of our proof is that some false positives are difficult to fix—in particular, these are the queries where an AMQ is unable to distinguish whether or not $x \in \mathcal{S}$ by looking at its state $\mathbf{L}$.[4] We call $y \in \mathcal{U} \setminus \mathcal{S}$ an ***absolute false positive*** of a state $\mathbf{L}$ and randomness ρ if there exists a set $\mathcal{S}'$ of size n and a sequence of queries $(x_1, \ldots, x_t)$ such that $y \in \mathcal{S}'$ and $\mathbf{L}$ is the state of the AMQ when queries $x_1, \ldots, x_t$ are performed on BUILD$(\mathcal{S}', \rho)$. We use AFP$(\mathbf{L}, \mathcal{S}, \rho)$ to denote the set of absolute false positives of state $\mathbf{L}$, randomness ρ, and true-positive set $\mathcal{S}$. We call $(\mathcal{S}', (x_1, \ldots, x_t))$ a ***witness*** to y.

We call $y \in \mathcal{U} \setminus \mathcal{S}$ an ***original absolute false positive*** of $\mathcal{S}$ and ρ if and only if $y \in$ AFP(BUILD$(\mathcal{S}, \rho), \mathcal{S}, \rho)$. We denote the set of original absolute false positives OFP$(\mathcal{S}, \rho) =$ AFP(BUILD$(\mathcal{S}, \rho), \mathcal{S}, \rho)$.

As the AMQ handles queries, it will need to fix some previous false positives. To fix a false positive, the AMQ must change its state so that it can safely answer ABSENT to it. For a state $\mathbf{L}$, we define the set of elements that are no longer false positives by the set FIX$(\mathbf{L}, \mathcal{S}, \rho) =$ OFP$(\mathcal{S}, \rho) \setminus$ AFP$(\mathbf{L}, \mathcal{S}, \rho)$. Note that all fixed false positives are original absolute false positives.

As an AMQ cannot have false negatives, it cannot fix an original absolute false positive y unless it learns that $y \notin \mathcal{S}$. This is formalized in the next two observations.

Observation 20. *For any randomness ρ, set $\mathcal{S}$, and state $\mathbf{L}$ of the AMQ, if a query $x \in$ AFP$(\mathbf{L}, \mathcal{S}, \rho)$, then* LOOKUP $(\mathbf{L}, x, \rho)$ *must return* PRESENT.

Observation 21. *Let $\mathbf{L}_1$ be a state of the AMQ before a query x and $\mathbf{L}_2$ be the updated state after x (that is, after invoking* LOOKUP *and possibly* ADAPT*). Let y be an absolute false positive of $\mathbf{L}_1$ with witness S_y. Then if y is not an absolute false positive of $\mathbf{L}_2$, then $x \in S_y$.*

[4]This is as opposed to easy-to-fix queries where, e.g., the AMQ answers PRESENT randomly to confuse an adversary. For all previous AMQs we are aware of, all false positives are absolute false positives.

B. Analysis

We start with an overview of the lower bound.

First, we recall a known result (Claim 23) that a space-efficient AMQ must start with a large number of original absolute false positives for almost all $\mathcal{S}$. Given that an AMQ has a large number of original absolute false positives, an adversary can discover a fraction of them through randomly chosen queries Q (Lemma 24).

Next, we show that through adaptive queries, the adversary forces the AMQ to fix almost all of these discovered original absolute false positives, for most sets Q (Lemma 25 and Lemma 26).

The crux of the proof relies on Lemma 27, which says that the AMQ cannot fix too many *extra* original absolute false positives during the attack—thus, it needs a large number of distinct "fixed" sets to cover all the different sets of original absolute false positives that the adversary forces the AMQ to fix. This is where we use that the AMQ only receives a limited amount of feedback on each false positive—it cannot fix more false positives without risking some false negatives.

Finally, we bound lower bound the space used by the AMQ by observing that there is a 1-to-1 mapping from "fixed" sets of original absolute false positives to AMQ states. Thus, we can lower bound the number of AMQ states (and hence the space needed to represent them) by lower-bounding the number of sets of original absolute false positives the adversary can force the AMQ to fix.

Observation 22. *For a given randomness ρ and set $\mathcal{S}$ of size n, consider two fixed false positive sets* $\text{FIX}(\mathbf{L}_1, \mathcal{S}, \rho)$ *and* $\text{FIX}(\mathbf{L}_2, \mathcal{S}, \rho)$. *Then if* $\text{FIX}(\mathbf{L}_1, \mathcal{S}, \rho) \neq \text{FIX}(\mathbf{L}_2, \mathcal{S}, \rho)$, *then* $\mathbf{L}_1 \neq \mathbf{L}_2$.

Discovering original absolute false positives through random queries. While for some special sets $\mathcal{S}$ given in advance, an AMQ may be able to store $\mathcal{S}$ very accurately (with very few false positives), this is not true for most random sets $\mathcal{S}$ chosen from the universe by the adversary. We note the following claim from Naor and Yogev [16].

Claim 23 ([16, Claim 5.3]). *Given any randomness ρ of AMQ using space $m \leq n \log 1/\varepsilon_0 + 4n$ bits, for any set $\mathcal{S}$ of size n chosen uniformly at random from $\mathcal{U}$, we have:* $\Pr_{\mathcal{S}}\left[|\text{OFP}(\mathcal{S}, \rho)| \leq u\varepsilon_0\right] \leq 2^{-n}$.

For the remainder of this section, we fix a set $\mathcal{S}^* \subseteq \mathcal{U}$ of size n such that $|\text{OFP}(\mathcal{S}^*, \rho)| > u\varepsilon_0$.[5] Let $\mathcal{Q}$ be the set of all possible query sets Q the adversary can choose, that is, $\mathcal{Q} = \{Q \subseteq \mathcal{U} \backslash \mathcal{S}^* \mid |Q| = n\}$. (We do not include S^* in the notation of $\mathcal{Q}$ for simplicity.) The following lemma follows immediately from Chernoff bounds.

[5]With probability $1/2^n$, the adversary gets unlucky and chooses a set $\mathcal{S}^*$ that does not satisfy this property, in which case he fails. This is okay, because we only need to show *existence* of a troublesome set $\mathcal{S}^*$—and we in fact show the stronger claim that most $\mathcal{S}^*$ suffice.

Lemma 24. *For a fixed randomness ρ of an AMQ of size $m \leq n \log 1/\varepsilon_0 + 4n$ and fixed set $\mathcal{S}^*$ such that $|\text{OFP}(\mathcal{S}^*, \rho)| > u\varepsilon_0$, we have* $\Pr_{Q \in \mathcal{Q}}\left[|Q \cap \text{OFP}(\mathcal{S}^*, \rho)| = \Omega(n\varepsilon_0)\right] \geq 1 - 1/\text{poly}(n)$.

Forcing the adaptive AMQ to fix large number of original absolute false positives. From the definition of sustained false-positive rate, the AMQ must fix an ε fraction of false positives in expectation in each round. If the expected number of false positives that the AMQ has to fix in each round is high, classic concentration bounds imply that the AMQ must fix close to this expected number with high probability in each round. This implies that there must be a round where the AMQ fixes a large number of original absolute false positives. The next lemma formalizes this intuition.

For a given Q, let $\Phi(Q, \mathcal{S}^*, \rho)$ be the maximal-sized set of query elements (out of Q) that the AMQ has fixed simultaneously in any state. For $1 \leq i \leq t$, let $\mathbf{L}_i$ be the state of the AMQ after query x_i. Then we let $\Phi(Q, \mathcal{S}^*, \rho) = \text{FIX}(\mathbf{L}_{t'}, \mathcal{S}^*, \rho)$ for the smallest t' such that $|\Phi(Q, \mathcal{S}^*, \rho)| \geq \text{FIX}(\mathbf{L}_{t''}, \mathcal{S}^*, \rho)$ for any t''.

The following lemma shows that the AMQ must, at the beginning of some round in the first $O(n)$ queries by the adversary, fix $\widetilde{\Omega}(n\varepsilon_0)$ false positives.

Lemma 25. *Consider an AMQ of size $m \leq n \log 1/\varepsilon_0 + 4n$. For any set Q satisfying $Q \cap \text{OFP}(\mathcal{S}^*, \rho) = \Omega(n\varepsilon_0)$, there exists a round $T(Q, \rho)$ and a state $\mathbf{L}_{T(Q,\rho)}$ at the beginning of round $T(Q, \rho)$ such that $|\text{FIX}(\mathbf{L}_{T(Q,\rho)}, \mathcal{S}^*, \rho)| = \Omega(n\varepsilon_0/\log_\varepsilon \varepsilon_0)$ w.h.p., that is,*

$$\Pr_{\rho}\left[|\Phi(Q, \mathcal{S}^*, \rho)| = \Omega(n\varepsilon_0/\log_\varepsilon \varepsilon_0) \;\middle|\; |Q \cap \text{OFP}(\mathcal{S}^*, \rho)| = \Omega(n\varepsilon_0)\right] \geq 1 - 1/\text{poly}(n).$$

Round $T(Q, \rho)$ is reached in at most $O(n)$ total queries.

Proof: The definition of sustained false positive rate means the AMQ must fix an ε fraction of queries in each round with high probability via Chernoff bounds.

Then, by Observation 21, with overwhelming probability we cannot fix a query x until we see x again (as it's unlikely we saw an element in a given witness set in the meantime). Thus, if an element is not fixed at the beginning of a round, it will not be fixed when it is queried during that round; in other words, the aforementioned fixed elements are all fixed when the round begins. See the full version [2] for details. ■

For simplicity, let $\varepsilon_0' = \varepsilon_0/\log_\varepsilon \varepsilon_0$. (This does not affect our final bounds.) The next lemma follows from Lemmas 24 and 25 and shows that (for most ρ), most query sets Q satisfy Lemma 25 with high probability.

Lemma 26. *Given an AMQ of size $m \leq n \log 1/\varepsilon_0 + 4n$ and set $\mathcal{S}^*$ such that $|\text{OFP}(\mathcal{S}^*, \rho)| \geq u\varepsilon_0$, for all but a*

$1/\text{poly}(n)$ *fraction of* Q*, there exists a round* $T(Q,\rho)$ *such that the AMQ is forced to fix* $\Omega(n\varepsilon_0')$ *original absolute false positive queries w.h.p. over* ρ*, that is,*

$$\Pr_{\rho}\left[\Pr_{Q\in\mathcal{Q}}[|\Phi(Q,\mathcal{S}^*,\rho)| = \Omega(n\varepsilon_0')] \geq 1 - \frac{1}{\text{poly}(n)}\right] \geq 1 - 1/\text{poly}(n).$$

Small AMQs cannot fix too many original absolute false positives. Next, we show that for a randomly-chosen y, knowing $y \notin \mathcal{S}^*$ is unlikely to give information to the AMQ about which set $\mathcal{S}^*$ it is storing. In particular, an AMQ may try to rule out false positives that may be correlated to a query y. We rule out this possibility in the following lemma. On a random query, the AMQ is very unlikely to fix a given false positive. We use this to make a w.h.p. statement about the number of fixed elements using Markov's inequality in our final proof.

Lemma 27. *Let* $(x_1,\ldots,x_t)$ *be a sequence of queries taken from a uniformly-sampled random set* $Q \subset \mathcal{U}$ *of size* n*, and let* $\mathbf{L}'$ *be the state of the AMQ after these queries. For an element* $y \in \mathcal{U}$*, the probability over* Q *that* y *is a fixed false positive after these queries is* n^2/u*. That is,* $\Pr_{Q\in\mathcal{Q}}[y \in \text{FIX}(\mathbf{L}',\mathcal{S}^*,\rho)] \leq n^2/u$.

Proof: If $y \in \text{FIX}(\mathbf{L}',\mathcal{S}^*)$, then by Observation 21, for any witness set S' of y, $S' \cap Q$ must be nonempty. Fix a single witness set S'. For a given $s' \in S'$ and $x' \in Q$, $\Pr(s' = x') = 1/u$. Taking the union bound over all n^2 such pairs (s',x') we achieve the lemma. ■

Final lower bound. We prove the desired lower bound.

Theorem 5. *Any adaptive 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 whp to maintain any constant sustained false-positive rate* $\varepsilon < 1$.

Proof: Assume by contradiction that $m \leq n\log 1/\varepsilon_0 + 4n$. Applying Observation 22 to the state $\mathbf{L}_{T(Q,\rho)}$ from Lemma 25 we obtain

$$2^m \geq \left|\{\text{FIX}(\mathbf{L}_{T(Q,\rho)},\mathcal{S}^*,\rho) \mid Q \subset \mathcal{U}, |Q| = n\}\right|.$$

We lower bound how many distinct fixed sets the AMQ needs to store for a given ρ.

By definition, $\Phi(Q,\mathcal{S}^*,\rho) \subseteq \text{FIX}(\mathbf{L}_{T(Q,\rho)},\mathcal{S}^*,\rho)$. But, the set of fixed elements cannot be much bigger than the set of fixed queries. Consider an arbitrary $x \in \mathcal{U}$. By Lemma 27, x is a fixed false positive with probability at most n^2/u. Thus, $\mathbf{E}_{Q\in\mathcal{Q}}[|\text{FIX}(\mathbf{L}_{T(Q,\rho)},\mathcal{S}^*,\rho),\mathcal{S}^*,\rho)|] = n^2$. By Markov's inequality, $\Pr_{Q\in\mathcal{Q}}\left[|\text{FIX}(\mathbf{L}_{T(Q,\rho)},\ \mathcal{S}^*,\rho)| = O(n^2)\right] = \Omega(1)$. Then there exists a set[6] $\mathcal{Q}^* = \left\{Q \subseteq \mathcal{U} \;\middle|\; |Q| = n,\ \text{FIX}\ (\mathbf{L}_{T(Q,\rho)},\mathcal{S}^*,\rho) = O(n^2)\right\}$ such that $|\mathcal{Q}^*| = \Omega(|\mathcal{Q}|)$. Thus, $\left|\left\{\text{FIX}\left(\mathbf{L}_{T(Q,\rho)},\mathcal{S}^*,\rho\right) \mid Q \in \mathcal{Q}\right\}\right| \geq |\{\Phi(Q,\mathcal{S}^*,\rho) | Q \in \mathcal{Q}^*, |\Phi(Q,\mathcal{S}^*,\rho)| = \Omega(n\varepsilon_0')\}| / \binom{O(n^2)}{\Omega(n\varepsilon_0')}$.

Now, we count the number of distinct $\Phi(Q,\mathcal{S}^*,\rho)$. Let $\mathcal{Z} = \{Z \subseteq \text{OFP}(\mathcal{S}^*,\rho) \mid |Z| = \Theta(n\varepsilon_0')\}$. We show that with high probability a randomly chosen set $Z \in \mathcal{Z}$ belongs to the set of $\Phi(Q,\mathcal{S}^*,\rho)$ for some Q (because we choose Q uniformly at random, this probabilistic argument lower bounds the number of such Z immediately). Recall that $|\text{OFP}(\mathcal{S}^*,\rho)| > u\varepsilon_0$.

$$\begin{aligned}
&\Pr_{Z\in\mathcal{Z}}[\exists Q \in \mathcal{Q}^* \text{ with } Z \subseteq \Phi(Q,\mathcal{S}^*,\rho)] \\
&\quad\geq \Pr_{\substack{Z\in\mathcal{Z}\\ Q\in\mathcal{Q}^* \text{ with } Q\supset Z}}[Z \subseteq \Phi(Q,\mathcal{S}^*,\rho)\}] \qquad (1)\\
&\quad\geq \Pr_{\substack{Z\in\mathcal{Z}\\ Q\in\mathcal{Q}^*\\ Q\supset Z}}\left[Z \subseteq \Phi(Q,\mathcal{S}^*,\rho) \middle| |\Phi(Q,\mathcal{S}^*,\rho)| = \Omega(n\varepsilon_0')\right]\\
&\qquad\cdot \Pr_{Q\in\mathcal{Q}^*}\left[|\Phi(Q,\mathcal{S}^*,\rho)| = \Omega(n\varepsilon_0') \;\middle|\; |Q\cap\text{OFP}(\mathcal{S}^*,\rho)| = \Omega(n\varepsilon_0)\right]\\
&\qquad\cdot \Pr_{Q\in\mathcal{Q}^*}[|Q\cap\text{OFP}(\mathcal{S}^*,\rho)| = \Omega(n\varepsilon_0)] \qquad (2)\\
&\quad\geq \left(1\Big/\binom{n}{\Omega(n\varepsilon_0')}\right)\left(1 - \frac{1}{\text{poly}(n)}\right). \qquad (3)
\end{aligned}$$

The first term in step (3) uses the fact that if the AMQ was forced to fix $\Omega(n\varepsilon_0')$ queries out of query set Q, then the probability that a randomly chosen Z corresponds to this forced query set is at most 1/(total number of possible forced subsets of Q). There can be at most $\binom{n}{\Omega(n\varepsilon_0')}$ such subsets. The second and third term in step (3) follow from Lemma 26 and Lemma 24 respectively—because $|\mathcal{Q}^*| = \Omega(|\mathcal{Q}|)$, a simple probabilistic argument shows that the probability over $Q \in \mathcal{Q}^*$ rather than $Q \in \mathcal{Q}$ retains the high probability bounds. Thus, we lower bound the total number of distinct forced sets (ignoring the lower-order $1 - 1/\text{poly}(n)$ term) as

$$\left|\left\{\Phi(Q,\mathcal{S}^*,\rho) \mid Q \in \mathcal{Q}^*\right\}\right| \geq \binom{u\varepsilon_0}{\Omega(n\varepsilon_0')}\Big/\binom{n}{\Omega(n\varepsilon_0')}.$$

Putting it all together,

$$2^m \geq \binom{u\varepsilon_0}{\Omega(n\varepsilon_0')}\Big/\binom{n}{\Omega(n\varepsilon_0')}\binom{O(n^2)}{\Omega(n\varepsilon_0')}.$$

Taking logs and simplifying,

$$m = \Omega\left(n\varepsilon_0'\left(\log\frac{u\log(1/\varepsilon_0)}{n} - \log\frac{n}{\varepsilon_0'}\right)\right).$$

We have $\log(u\log(1/\varepsilon_0)/n) - \log(n/\varepsilon_0') = \Omega(\log u)$ and $\varepsilon_0' = \Omega(\varepsilon_0/\log(1/\varepsilon_0))$. Thus, $m = \Omega\left(\frac{n\varepsilon_0\log u}{\log(1/\varepsilon_0)}\right)$.

Using the definition of ε_0, we have two cases.

1) If $1/n^{1/4} \geq (\log^2\log u)/\log u$, then $m = \Omega(n\log n\log u/\log\log n)$.

[6] $\mathcal{Q}^*$ is a function of ρ and $\mathcal{S}$; we omit this to simplify notation.

2) If $1/n^{1/4} < (\log^2 \log u)/\log u$, then $m = \Omega(n \log\log u)$.

In case 1, we get a contradiction to the assumption that $m \le n \log 1/\varepsilon_0 + 4n$. In case 2 if $m \le n \log\log u + 4n$, then we obtain a bound of $\Omega(n \log\log u)$. ■

Matching upper bound. In [2], we prove the following lemma which shows that the above bound is tight. In short, we take a normal Bloom Filter with error probability ε_0, and augment it with a whitelist.

Lemma 28. *There exists an adaptive AMQ that can handle $O(n)$ adaptive queries using $O(\min\{n \log n, n \log\log u\})$ bits of space w.h.p.*

ACKNOWLEDGMENTS

This research was supported in part by NSF CCF 1114809, 1217708, 1218188, 1314633, 1439084, 1553385 1617618, 1637458, CCF-BSF 1716252, CNS 1408695, 1408782, IIS 1247726, 1251137, 1541613, NIH 1U01CA198952-01, ERC 614331, VILLUM Foundation 16582, and by Sandia National Labs, EMC, NetAPP.

We thank Rasmus Pagh for helpful discussions.

REFERENCES

[1] Y. Arbitman, M. Naor, and G. Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Foundations of Computer Science*, pages 787–796, 2010.

[2] M. A. Bender, M. Farach-Colton, M. Goswami, R. Johnson, S. McCauley, and S. Singh. Bloom filters, adaptivity, and the dictionary problem. *arXiv preprint arXiv:1711.01616*, 2018.

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

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

[5] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[6] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman. Exact and approximate membership testers. In *Symposium on Theory of Computing*, pages 59–65, 1978.

[7] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Symposium on Discrete Algorithms*, pages 30–39, 2004.

[8] S. Cohen and Y. Matias. Spectral bloom filters. In *International Conference on Management of Data*, pages 241–252. ACM SIGMOD, 2003.

[9] E. Demaine, F. der Heide, R. Pagh, and M. Pătraşcu. De dictionariis dynamicis pauco spatio utentibus. *Latin American Symposium on Theoretical Informatics*, pages 349–361, 2006.

[10] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *International Conference on Management of Data*, pages 25–36. ACM SIGMOD, 2006.

[11] D. Eppstein, M. T. Goodrich, M. Mitzenmacher, and M. R. Torres. 2-3 cuckoo filters for faster triangle listing and set intersection. In *Principles of Database Systems*, pages 247–260. ACM, 2017.

[12] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *International Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.

[13] M. Farach-Colton, R. J. Fernandes, and M. A. Mosteiro. Bootstrapping a hop-optimal network in the weak sensor model. *Transactions on Algorithms*, 5(4):37:1–37:30, Nov. 2009.

[14] S. Lovett and E. Porat. A lower bound for dynamic approximate membership data structures. In *Foundations of Computer Science*, pages 797–804, 2010.

[15] M. Mitzenmacher, S. Pontarelli, and P. Reviriego. Adaptive cuckoo filters. In *Workshop on Algorithm Engineering and Experiments*, pages 36–47, 2018.

[16] M. Naor and E. Yogev. Bloom filters in adversarial environments. In *Annual Cryptology Conference*, pages 565–584. Springer, 2015.

[17] A. Pagh, R. Pagh, and S. S. Rao. An optimal bloom filter replacement. In *Symposium on Discrete Algorithms*, pages 823–829, 2005.

[18] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. A general-purpose counting filter: Making every bit count. In *International Conference on Management of Data*, pages 775–787, 2017.

[19] E. Porat. An optimal bloom filter replacement based on matrix solving. In *International Computer Science Symposium in Russia*, pages 263–273, 2009.

[20] S. Tarkoma, C. E. Rothenberg, E. Lagerspetz, et al. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys and Tutorials*, 14(1):131–155, 2012.

[21] Y. Zhu, H. Jiang, and J. Wang. Hierarchical bloom filter arrays (HBA): a novel, scalable metadata management system for large cluster-based storage. In *IEEE International Conference on Cluster Computing*, pages 165–174, 2004.