Weizmann Institute of Science

Thesis for the degree
Master of Science

Submitted to the Scientific Council of the
Weizmann Institute of Science
Rehovot, Israel

Tight Bounds for Sliding Bloom Filters

Author:
Eylon Yogev

Supervisor:
Prof. Moni Naor

November 2013

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Moni Naor. Without his guidance, encouragement, and extreme patience, this thesis never would have taken place. His vision and creativeness has inspired me in so many ways.

I would also like to thank my wife, Romi, for all of the love and support she has given me over the years. Without her nourishment, I never would have had the chance to succeed.

Abstract

A Bloom filter is a method for reducing the space (memory) required for representing a set by allowing a small error probability. In this thesis we consider a Sliding Bloom Filter: a data structure that, given a stream of elements, supports membership queries of the set of the last n elements (a sliding window), while allowing a small error probability and a slackness parameter.

The problem of sliding Bloom filters has appeared in the literature in several communities, but this work is the first theoretical investigation of it.

We formally define the data structure and its relevant parameters and analyze the time and memory requirements needed to achieve them. We give a low space construction that runs in $O(1)$ time per update with high probability (that is, for all sequences with high probability all operations take constant time) and provide an almost matching lower bound on the space that shows that our construction has the best possible space consumption up to an additive lower order term.

Contents

1 Introduction

Given a stream of elements, we consider the task of determining whether an element has appeared in the last n elements of the stream. To accomplish this task, one must maintain a representation of the last n elements at each step. One issue, is that the memory required to represent them might be too large and hence an approximation is used. We formally define this approximation and completely characterize the space and time complexity needed for the task.

In 1970 Bloom [Blo70] suggested an efficient data structure, known as the '*Bloom filter*', for reducing the space required for representing a set S by allowing a small error probability on membership queries. The problem is also known as the approximate membership problem (however, we refer to any solution simply as a 'Bloom filter'). A solution is allowed an error probability of ε for elements not in S (false positives), but no errors for members of S. In this thesis, we consider the task of efficiently maintaining a Bloom filter of the last n elements (called 'the sliding window') of a stream of elements.

We define an (n, m, ε)-*Sliding Bloom Filter* as the task of maintaining a Bloom filter over the last n elements. The answer on these elements must always be 'Yes', the m elements that appear prior to them have no restrictions i.e. any answer is accepted (m is a slackness parameter) and for any other element the answers must be 'Yes' with probability at most ε. In case m is infinite, all elements prior to the current window have no restrictions. In this case we write for short (n, ε)-Sliding Bloom Filter.

The problem was studied in several communities and various solutions were suggested. In this thesis, we focus on a theoretical analysis of the problem and provide a rigorous analysis of the space and time needed for solving the task. We construct a Sliding Bloom Filter with $O(1)$ query and update time, where the running time is worst case with high probability (see the theorems in Section 1.2 for precise definitions) and has near optimal space consumption. We prove a matching space lower bound that is tight with our construction up to an additive lower order term. Roughly speaking, our main result is figuring out the first two terms of the space required by a Sliding Bloom Filter: $n \log \frac{1}{\varepsilon} + n \cdot \max\left\{\log\log \frac{1}{\varepsilon}, \log \frac{n}{m}\right\}$

A simple solution to the task is to partition the window into blocks of size m and for each block maintain its own Bloom filter. This results in maintaining $\left\lceil \frac{n}{m} + 1 \right\rceil$ Bloom filters. To determine if an element appeared or not we query all the Bloom filters and answer 'Yes' if any of them answered positively. There are immediate drawbacks of this solution, even assuming the Bloom filters are optimal in space and time:

- Slow query time: $\left\lceil \frac{n}{m} + 1 \right\rceil$ Bloom filter lookups.
- High error probability: since an error can occur on each block, to achieve an effective error probability of ε we need to set each Bloom filter to have error $\varepsilon' = \frac{\varepsilon m}{n+m}$, which means that the total space used has to grow (relative to a simple Bloom filter) by roughly $n \log \frac{n+m}{m}$ bits (see Section 1.3).
- Sub-optimal space consumption for large m: the first two drawbacks are acute for small m, but when m is large, say $n = m$, then each block is large which results in a large portion of the memory being 'wasted' on old elements.

We overcome all of the above drawbacks: the query time is always constant and for *any* m the space consumption is nearly optimal. We also give an almost matching lower bound on the space that shows that our construction has the best possible space consumption up to an additive lower order term. We regard the lower bound as the main contribution of this work.

Sliding Bloom Filters can be used in a wide range of applications and we discuss two settings where they are applicable and have been suggested. In one setting, Bloom filters are used to quickly determine whether an element is in a local web cache [FCAB00], instead of querying the cache which may be slow. Since the cache has limited size, it usually stores the least recently used items (LRU policy). A Sliding Bloom Filter is used to represent the last n elements used and thus, maintain a representation of the cache's contents at any point in time.

Another setting consists of the task of identifying duplicates in streams. In many cases, we consider the stream to be unbounded, which makes it impractical to store the entire data set and answer queries precisely and quickly. Instead, it may suffice to find duplicates over a sliding window while allowing some errors. In this case, a Sliding Bloom Filter (with m set to infinity) suffices and in fact, we completely characterize the space complexity needed for this problem.

1.1 Problem Definition

Given a stream of elements $\sigma = x_1, x_2, ...$ from a finite universe U of size u, parameters n, m and ε, such that $n < \varepsilon u$, we want to approximately represent a sliding window of the n most recent elements of the stream. An algorithm A is given the elements of the stream one by one, and does not have access to previous elements that were not stored explicitly. Let $\sigma_t = x_1, \ldots, x_t$ be the first t elements of the stream σ and let $\sigma_t(k) = x_{\max(0,t-k+1)}, \ldots, x_t$ be the last k elements of the stream σ_t. At any step t the current window is $\sigma_t(n)$ and the m elements before them are $\sigma_{t-n}(m)$. If $m = \infty$ then define $\sigma_{t-n}(m) = x_1, \ldots, x_{t-n}$. Denote $A(\sigma_t, x) \in \{\text{'Yes'}, \text{'No'}\}$ the result of the algorithm on input x given the stream σ_t. We call A an (n, m, ε)-*Sliding Bloom Filter* if for any $t \geq 1$ the following two conditions hold:

1. For any $x \in \sigma_t(n)$: $\Pr[A(x) = \text{'Yes'}] = 1$
2. For any $x \notin \sigma_t(n + m)$: $\Pr[A(x) = \text{'Yes'}] \leq \varepsilon$

where the probability is taken over the internal randomness of the algorithm A. Notice that for an element $x \in \sigma_{t-n}(m)$ the algorithm may answer arbitrarily (no restrictions). See Figure 1.

An algorithm A for solving the problem is measured by its memory consumption, the time it takes to process each element and answer a query. We denote by $|A|$ the maximum number of bits used by A at any step. The model we consider is the unit cost RAM model in which the elements are taken from a universe of size u, and each element can be stored in a single word of length $w = \log u$ bits. Any operation in the standard instruction set can be executed in constant time on w-bit operands. This includes addition, subtraction, bitwise Boolean operations, left and right bit shifts by an arbitrarily number of positions,

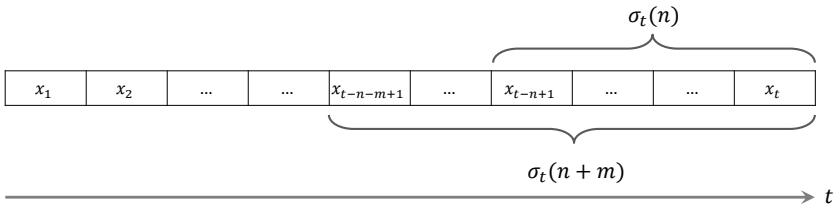

Figure 1: The sliding window of the last n and $n+m$ elements

and multiplication. The unit cost RAM model is considered the standard model for the analysis of the efficiency of data structures.

An element not in S on which the data structure accepts is called a false positive. At any point in time, the fraction of false positives in U is called the false positive rate.

1.2 Our Contributions

We provide tight upper and lower bounds to the (n, m, ε)-problem. In fact, we achieve space optimality up to the second term. Our first contribution is a construction of an efficient Sliding Bloom Filter: it has query time $O(1)$ worst case and update time $O(1)$ worst case with high probability, for the entire sequence. For $\varepsilon = o(1)$ the space consumption is near optimal: the two leading terms are optimal in constants.

Theorem 1.1. *For any $m > 0$, and sufficiently large n there exists an (n, m, ε)-Sliding Bloom Filter having the following space and time complexity on a unit cost RAM:*

***Time**: Query time is $O(1)$ worst case. For any polynomial $p(n)$ and sequence of at most $p(n)$ operations, with probability at least $1 - 1/p(n)$, over the internal randomness of the data structure, all insertions are performed in time $O(1)$ worst case.*

***Space**: the space consumption is: $(1 + o(1))\left(n \log \frac{1}{\varepsilon} + n \cdot \max\left\{\log \frac{n}{m}, \log\log \frac{1}{\varepsilon}\right\}\right)$.*
In particular, for constant error ε we get that the space consumption is: $n \log\left(\frac{n}{m}\right) + O(n)$. Otherwise, for sub-constant ε that satisfies $\varepsilon = 2^{-O(\log^{1/3} n)}$ we get that:

1. *If $m \geq \varepsilon n$ then the space consumption is: $n \log \frac{1}{\varepsilon} + n \cdot \max\left\{\log \frac{n}{m}, \log\log \frac{1}{\varepsilon}\right\} + O(n)$*
2. *If $m < \varepsilon n$ then the space consumption is: $n \log \frac{1}{\varepsilon} + (1 + o(1)) n \log \frac{n}{m}$*

The challenge we face is achieving constant time operations while space consumption remains very tight. In designing our algorithm we assemble ideas from several previous works along with new ones. The basic skeleton of the algorithm shares ideas with the work of Zhang and Guan [ZG08], however, their algorithm is based on the traditional Bloom filter and has immediate drawbacks: running time is super-constant and the space is far from optimal. To get an error probability of ε they use $M = O(n \log n \log \frac{1}{\varepsilon})$ bits, and moreover this is assuming the availability of truly random hash functions.

Thorup [Tho11] considered a similar data structure of hash tables with timeouts based on linear probing. He did not allow any error probability nor any slackness (i.e. $\varepsilon = 0$ and

$m = 0$ in our terminology). The query time, as is general for linear probing, is only constant in expectation, and the space is only optimal within a constant factor.

Pagh, Pagh and Rao [PPR05] showed that the traditional construction of a Bloom filter can be replaced with a construction that is based on dictionaries. The dictionary based Bloom filter has the advantage that its running time and space consumption are completely determined by the dictionary itself, and it does not assume availability of truly random functions. Given the developments in succinct dictionaries, using this alternative has become more appealing.

Our algorithm is conceptually similar to the work of Zhang and Guan. However, we replace the traditional implementation of the Bloom filter with a dictionary based one. As the underlying dictionary, we use the state of the art dictionary given by Arbitman, Naor and Segev [ANS10], known as Backyard Cuckoo Hashing. Then we apply a similar method of lazy deletions as used by Thorup on the Backyard Cuckoo Hashing dictionary. Moreover, we introduce a slackness parameter m and instead of storing the exact index of each element we show a trade-off parameter c between the accuracy of the index stored and the number of elements we store in the dictionary. Optimizing c along with the combined methods described gives us the desired result: constant running time, space consumption of nearly $n \log \frac{1}{\varepsilon} + n \cdot \max\left\{\log\log \frac{1}{\varepsilon}, \log \frac{n}{m}\right\}$ which is optimal in both leading constants and no assumption on the availability of truly random functions. We inherit the implementation complexity of the dictionary, and given an implementation of one, it is relatively simple to complete the algorithm's implementation.

Our second contribution, and technically the more involved one, is a matching space lower bound. We prove that if $\varepsilon = o(1)$ then any Sliding Bloom Filter must use space that is within an additive low order term of the space of our construction, regardless of its running time.

Theorem 1.2. *Let A be an (n, m, ε)-Sliding Bloom Filter where $n < \varepsilon u$, then*

1. *If $m > 0$ then $|A| \geq n \log \frac{1}{\varepsilon} + n \cdot \max\left\{\log \frac{n}{m}, \log\log \frac{1}{\varepsilon}\right\} - O(n)$*
2. *If $m = \infty$ then $|A| \geq n \log \frac{1}{\varepsilon} + n \log\log \frac{1}{\varepsilon} - O(n)$*

From Theorems 1.1 and 1.2 we conclude that making m larger than $n/\log \frac{1}{\varepsilon}$ does not make sense: one gets the same result for any value in the range $[n/\log \frac{1}{\varepsilon}, \infty)$. When m is small (less than εn), then the dominant expression in both the upper and lower bounds is $n \log\left(\frac{n}{m}\right)$.

The lower bound is proved by an encoding argument which is a common way of showing lower bounds in this area (see for example [PSW13]). Specifically, the idea of the proof is to use A to encode a set S and a permutation π on the set corresponding to the order of the elements in the set. We consider the number of steps from the point an element is inserted to A to the first point where A answers 'No' on it, and we define λ to be the sum of n such lengths. If λ is large, then there is a point where A represents a large portion of S, which benefits in the encoding of S. If λ is small, then A can be used as an approximation of π, thus encoding π precisely requires a small amount of bits. In either case, the encoding must be larger than the entropy lower bound[1] which yields a bound on the size of A. The optimal

[1]The entropy lower bound is base 2 logarithm of the size of the set of all possible inputs. In our case, all possible pairs (S, π).

value of the trade-off between representing a larger set or representing a more accurate ordering is achieved by our construction. In this sense, our upper bound and lower bound match not only by 'value' but also by 'structure'.

1.3 Related Work and Background

The data structure for the approximate set membership as suggested by Bloom in 1970 [Blo70] is relatively simple: it consists of a bit array which is initiated to '0' and k random hash functions. Each element is mapped to k locations in the bit array using the hash functions. To insert an element set all k locations to 1. On lookup return 'Yes' if all k locations are 1. To achieve an error probability of ε for a set of size n Bloom showed that if $k = \log \frac{1}{\varepsilon}$ then the length of the bit array should be roughly $1.44n \log \frac{1}{\varepsilon}$ (where the 1.44 is an approximation of $\log_2(e)$). Since its introduction Bloom filters have been investigated extensively and many variants, implementations and applications have been suggested. We call any data structure that implements the approximate set membership a 'Bloom filter'. A comprehensive survey (for its time) is Broder and Mitzenmacher [BM03].

A lot of attention was devoted for determining the exact space and time requirements of the approximate set membership problem. Carter et al. [CFG+78] proved an entropy lower bound of $n \log \frac{1}{\varepsilon}$, when the universe U is large. They also provided a reduction from approximate membership to *exact* membership, which we use in our construction. The retrieval problem associates additional data with each element of the set. In the static setting, where the elements are fixed and given in advance, Dietzfelbinger and Pagh propose a reduction from the retrieval problem to approximate membership [DP08]. Their construction gets arbitrarily close to the entropy lower bound.

In the dynamic case, Lovett and Porat [LP10] proved that the entropy lower bound cannot be achieved for any *constant* error rate. They show a lower bound of $C(\varepsilon) \cdot n \log \frac{1}{\varepsilon}$ where $C(\varepsilon) > 1$ depends only on ε. Pagh, Segev and Wieder [PSW13] showed that if the size n is not known in advance then at least $(1 - o(1))n \log \frac{1}{\varepsilon} + \Omega(n \log \log n)$ bits of space must be used. The Sliding Bloom Filter is in particular also a Bloom Filter in a dynamic setting, thus the [LP10] and [PSW13] bounds are applicable.

As discussed, Pagh, Pagh and Rao [PPR05] suggested an alternative construction for the Bloom filter. They used the reduction of Carter et al. to improve the traditional Bloom filter in several ways: Lookup time becomes $O(1)$ independent of ε, has succinct space consumption, uses explicit hash functions and supports deletion. In the dynamic setting for a constant ε we do not know what is the leading term in the memory needed, however, for any sub-constant ε we know that the leading term is $n \log \frac{1}{\varepsilon}$: Arbitman, Naor and Segev present a solution, called 'Backyard Cuckoo Hashing', which is optimal up to an additive lower order term (i.e., it is a succinct representation) [ANS10]. Thus, in this thesis we focus on sub-constant ε.

The model of sliding windows was first introduced by Datar et al. [DGIM02]. They consider maintaining an approximation of a statistic over a sliding window. They provide an efficient algorithm along with a matching lower bound.

Data structures for problems similar to the Sliding Bloom Filters have been studied in the literature quite extensively over the past years. The simple solution using $m = n$ consists of two large Bloom filters which are used alternatively. This method known as *double*

buffering was proposed for classifying packets caches [CLF04]. Yoon [Yoo10] improved this method by using the two buffers simultaneously to increase the capacity of the data structure. Deng and Rafiei [DR06] introduced the Stable Bloom filter and used it to approximately detect duplicates in stream. Instead of a bit array they use an array of counters and to insert an element they set all associated counters to the maximal value. At each step, they randomly choose counters to decrease and hence older element have higher probability of being decreased and eventually evicted over time. Metwally et al. [MAEA05] showed how to use Bloom filters to identify duplicates in click streams. They considered three models: Sliding Windows, Landmark Windows and Jumping Windows and discuss their relations. A comprehensive survey including many variations is given by Tarkoma et al. [TRL12]. However, as far as we can tell, no formal definition of a Sliding Bloom Filter as well as a rigorous analysis of its space and time complexity, appeared before.

2 The Construction of a Succinct Sliding Bloom Filter

Our algorithm uses a combination of transforming the approximate membership problem to the exact membership problem plus a solution to the retrieval problem. On an input x, we store $h(x)$, for some hash function h, in a dynamic dictionary and in addition store some information on the last time where x appeared. We consider the stream to be divided into generations of size n/c each, where c is a parameter that will be optimized later. The first n/c elements are generation 1, the next n/c elements are generation 2 etc. The current window contains the last n elements and consists of at most $c+1$ different generations. Therefore, at each step, we maintain a set S that represents the last $c+1$ generations (that is, at most $n+n/c$ elements) and count the generations mod $(c+1)$. In addition to storing $h(x)$, we associate $s = \log(c+1)$ bits indicating the generation of x. Every n/c steps, we delete elements associated with the oldest generation. We adjust c to optimize the space consumption while requiring $n/c \leq m$.

In this section, we describe the algorithm in more detail. We first present the transformation from approximate to exact membership (Section 2.1). We define a dynamic dictionary and the properties we need from it in order to implement our algorithm (Section 2.3). Then, we describe the algorithm in two stages, using any dictionary as a black box. The memory consumption is merely the memory of the dictionary and therefore we use one with succinct representation. At first, in Section 2.3, the running time will not be optimal and depend on c (which is not a constant), even if we use optimal dictionaries. Then, in Section 2.4, we describe how to eliminate the dependency on c as well as deamortizing the algorithm, making the running time constant for each operation. This includes augmenting the dictionary, and thus it can no longer be treated as a black box. We prove correctness and analyze the resulting memory consumption and running time.

2.1 Approximate Membership and Exact Membership

Carter et al. [CFG+78] showed a transformation from approximate membership to exact membership that works as follows. We want to represent a set S of size n and support membership queries in the following manner: For a query on $x \in S$ we answer 'Yes' and for

$x \notin S$ we answer 'Yes' with probability at most ε. Choose a hash function $h \in \mathcal{H}$ from a universal family of hash functions mapping $U \to [n/\varepsilon]$. Then for any S of size at most n it holds that for any $x \in U$:

$$\Pr_h[h(x) \in h(S)] \leq \sum_{y \in S} \Pr_h[h(x) = h(y)] \leq n\frac{\varepsilon}{n} = \varepsilon$$

where the first inequality comes from a union bound and the second from the definition of a universal hash family. This implies that storing $h(S)$ suffices for solving the approximate membership problem. This dictionary-based construction and the traditional construction can be viewed as lying on a spectrum - the former writes many bits in one location, whereas the latter writes one bit in many locations.

To store $h(S)$ we use an exact dictionary $\mathcal{D}$, which supports `insert` (including associated data), `delete` and `update` procedures (the update procedure can be simulated by a delete followed by an insert). While most dictionaries support these basic procedures, we require $\mathcal{D}$ to additionally support the ability of *scanning*. We further discuss these properties in the next section.

Number of false positives: We note that in addition to the error bound on each element, we can bound the total number of false positives in the universe. Any hash family $\mathcal{H}$ divides the universe to $\lceil n/\varepsilon \rceil$ 'bins', and the number of false positives is the total number of elements in any bin containing an element from S. If $\mathcal{H}$ divides U to (roughly) equally sized bins, each of size at most $\lceil \varepsilon u/n \rceil$, then the total number of false positives is at most $n \cdot \lceil \varepsilon u/n \rceil \leq \varepsilon u + n$. A simple example of such a hash family can be obtained by choosing a prime $p \geq u$ then defining $\mathcal{H}$ to be $h_a(x) = ((ax \mod p) \mod \lceil n/\varepsilon \rceil)$, where a is a random integer modulo p with $a \neq 0$ [CW79]. In this case, the bound holds with certainty for *any* function $h \in \mathcal{H}$. This property is not guaranteed by the traditional construction of Bloom, and we further discuss it in Section 3.

2.2 Succinct Dynamic Dictionary

The information-theoretic lower bound on the minimum number of bits needed to represent a set S of size n out of M different elements is $\mathcal{B} = \mathcal{B}(M, n) = \left\lceil \log \binom{M}{n} \right\rceil = n \log M - n \log n + O(n)$. A succinct representation is one that uses $(1 + o(1))\mathcal{B}$ bits [Dem07]. A significant amount of work was devoted for constructing dynamic dictionaries over the years and most of them are appropriate for our construction. Some have good theoretical results and some emphasize the actual implementation. In order for the reduction to compete with the Bloom filter construction (in terms of memory consumption) we must use a dynamic dictionary with succinct representation. There are several different definitions in the literature for a *dynamic* dictionary. A static dictionary is a data structure storing a finite subset of a universe U, supporting only the `member` operation. In this thesis, we refer to a dynamic dictionary where only an upper bound n on the size of S is given in advance and it supports the procedures `member`, `insert` and `delete`. The memory of the dictionary is measured with respect to the bound n.

In addition to storing $h(S)$, we assume $\mathcal{D}$ supports associating data with each element. Specifically, we want to store s-bits of data with each element, where s is fixed and known in

advance. Finally, we assume the dictionary supports *scanning*, that is, the ability to go over the associated data of all elements of the dictionary, and delete the element if needed. Using the scanning process, we scan the generations stored in the dictionary and delete elements of specific generations.

Several dynamic dictionaries can be used in our construction of a Sliding Bloom Filter. The running time and space consumption are directly inherited from the dictionary, making it an important choice. We use the 'Backyard Cuckoo Hashing' construction of [ANS10] (but other alternative are possible). It supports `insert` and `delete` in $O(1)$ worst case with high probability while having a succinct representation. Implicitly in their work, they support associating any fixed number of bits and scanning. When s-bits of data are associated with each $x \in S$, the representation lower bound becomes $\mathcal{B} + ns$ bits. For concreteness, the memory consumption of their dictionary is $(1 + o(1))(\mathcal{B} + ns)$, where the $o(1)$ hides the expression $\frac{\log\log n}{\log^{1/3} n}$.

2.3 An Algorithm with Dependency on ε

Initiate a dynamic dictionary $\mathcal{D}$ of size $n' = n\left(1 + \frac{1}{c}\right)$ as described above. Let $\mathcal{H} = \{h : U \to [n'/\varepsilon]\}$ be a family of universal hash functions and pick $h \in \mathcal{H}$ at random. At each step maintain a counter ℓ indicating the current generation and a counter i indicating the current element in the generation. At every step i is increased and every n/c steps i is reset back to 0 and ℓ is increased mod $(c+1)$.

To insert an element x check if $h(x)$ exists in $\mathcal{D}$. If not then insert $\langle h(x), \ell\rangle$ (insert $h(x)$ associated with ℓ) into $\mathcal{D}$. If $h(x)$ is in $\mathcal{D}$, then update the associated data of $h(x)$ to ℓ. Finally, update the counters i and ℓ. If ℓ has increased (which happens every n/c steps) then *scan* $\mathcal{D}$ and delete all elements with associated data equal to the new value of ℓ.

To query the data structure on an element x, return whether $h(x)$ is in $\mathcal{D}$. See Algorithm 1 for pseudo-code of the insert and lookup procedures.

Correctness: We first notice that $\mathcal{D}$ is used correctly and never represents a set of size larger than n'. In each step we either insert an element to generation ℓ or move an existing element to generation ℓ. In any case, each generation consists of at most n/c elements in $\mathcal{D}$. Each n/c we evict a whole generation, assuring no more than $c+1$ generations are present in the dictionary at once. Thus, at most n' elements are represented at any given step.

Next we prove that for any time t the three conditions in Theorem 1.1 hold. The first condition follows directly from the algorithm. Assume $h(x)$ is inserted with associated generation $\ell = j$. Notice that its associated generation can only increase. $h(x)$ will be deleted only when ℓ completes a full cycle and its value is j again, which takes at least n steps. Thus, for any $x \in \sigma_t(n)$, $h(x)$ is in $\mathcal{D}$ and the algorithm will always answer 'Yes'.

For the second condition assume that $x \notin \sigma_t(n+m)$ and notice that $n+m$ is at least $c+1$ generations. Assume w.l.o.g. that $S = \{y_1, \ldots, y_{n'}\}$ (S could have less than n' elements) is the set of elements represented in $\mathcal{D}$ at time t. Then $\Pr[h(x) = y_i] = \frac{\varepsilon}{n'}$ for all $i \in [n']$. Therefore, using a union bound we get that the total false positive probability is

$$\Pr[A(x) = \text{'Yes'}] = \Pr[h(x) \in h(S)] \leq \sum_{i=1}^{n'} \Pr[h(x) = y_i] \leq \varepsilon$$

```
Insert(x):
 1: if h(x) is a member of 𝒟 then
 2:     update h(x) to have data ℓ
 3: else
 4:     insert ⟨h(x), ℓ⟩ into 𝒟
 5: end if
 6: maintain counters i and ℓ
 7: if the value of ℓ has changed then
 8:     scan 𝒟 and delete elements of generation ℓ
 9: end if

Lookup(x):
 1: procedure MEMBER(x)
 2:     if h(x) is a member of 𝒟 then
 3:         return 'Yes'
 4:     else
 5:         return 'No'
 6:     end if
 7: end procedure
```

Algorithm 1: Pseudo-code of the Insert and Lookup procedures

Memory consumption: The bulk of memory is used for storing $\mathcal{D}$. In addition, we need to store two counters i and ℓ and the hash function h, which together take $O(\log n)$ bits. $\mathcal{D}$ stores n' elements out of $M = [n'/\varepsilon]$ while associating each with $s = \log c$ bits. Using the 'Backyard Cuckoo Hashing' dictionary yields a total space of

$$(1+o(1))\left(\mathcal{B}\left(\frac{n'}{\varepsilon}, n'\right) + n's\right) \quad = \quad (1+o(1))\cdot n\left(1+\frac{1}{c}\right)\left(\log\frac{1}{\varepsilon} + \log c + 1\right)$$

We minimize this expression, as a function of c, and get that the minimum is at the solution to $c - \log c = \log\frac{1}{\varepsilon} - 1$. An approximate solution is $c = \log\frac{1}{\varepsilon}$ and requiring that $n/c \leq m$ yields that $c = \max\left\{\log\frac{1}{\varepsilon}, m/n\right\}$ and the total space is

$$(1+o(1))\left(n\log\frac{1}{\varepsilon} + n\cdot\max\left\{\log\log\frac{1}{\varepsilon}, \log\frac{n}{m}\right\}\right)$$

as required. As mentioned, the $o(1)$ hides the term $\frac{\log\log n}{\log^{1/3} n}$, therefore if $\varepsilon = 2^{-O(\frac{\log^{1/3} n}{\log\log n})}$ then the product of $o(1)$ with $n\log\frac{1}{\varepsilon}$ is $O(1)$. If $m \geq \varepsilon n$ then the product of $o(1)$ with $n\max\left\{\log\log\frac{1}{\varepsilon}, \log(n/m)\right\}$ is $O(1)$ as well. Thus, we can write the space consumption as:

$$n\log\frac{1}{\varepsilon} + n\cdot\max\left\{\log\log\frac{1}{\varepsilon}, \log\frac{n}{m}\right\} + O(n).$$

Otherwise, if $m < \varepsilon n$ then we can write it as:

$$n\log\frac{1}{\varepsilon} + (1+o(1))n\log\frac{n}{m}.$$

If $\varepsilon = O(1)$ then $n \log \frac{1}{\varepsilon} = O(n)$ and $n \log \log \frac{1}{\varepsilon} = O(n)$ and we can write it as:

$$n \log \frac{n}{m} + O(n).$$

Running time: Assume that $\mathcal{D}$ supports $O(1)$ running time worst case for all procedures. The lookup procedure performs a single query to $\mathcal{D}$ and hence always runs in $O(1)$. In the insert procedure, every n/c steps, the value of ℓ is updated and we scan all elements in $\mathcal{D}$ deleting old elements. For any other step, the running time is $O(1)$. Therefore, the total running time for n/c steps is $O(n')$, which is $O(c)$ amortized running time. If $m \geq \log \frac{1}{\varepsilon}$ then $c = \log \frac{1}{\varepsilon}$ and the running time is $\log \frac{1}{\varepsilon}$, otherwise it is $O\left(\frac{n}{m}\right)$, which in both cases is not constant. We now show how to eliminate the large step, making the running time $O(1)$ worst case. Using the 'Backyard Cuckoo Hashing' dictionary we get that the total running time including the dictionary's operations is $O(1)$ worst case with high probability (over internal randomness of the dictionary).

2.4 Reducing the Running Time to Constant Amortized

We describe how to reduce the running time of the algorithm to be constant amortized, that is performing $O(n')$ operations over the course of n' steps. The main load of the algorithm of Section 2.3 stems from the need to scan the entire dictionary every n/c steps to delete old elements. In order to reduce the time devoted to each step we modify the algorithm to perform only a single scan every n' steps.

First, we extend the range of the generations counter ℓ to loop between 0 and $2c + 2$ (instead of between 0 and $c + 1$). This way, after n' elements (which is $c + 1$ generations) we can still distinguish between elements of the last $c + 1$ generations and the $c + 1$ generations before them, that need to be deleted. Now, every n' steps we scan the entire dictionary, deleting elements that are more than $c + 1$ generations old.

This reduces the number of scans performed, however too many elements are present in the dictionary at once. At any moment, only the $c + 1$ recent generations are considered active and the rest slated to be deleted. The dictionary is initialized to be of size n' and since we do not delete old elements immediately, there might be more than n' elements present in the dictionary. However, the number of *active* elements present will never exceed n'. Thus, we need the dictionary to support lazy deletions. That is, it should be able to consider non-active elements as deleted so that they do not interfere with other operations: whenever a non-active element is encountered it is simply deleted. The result is that as long as there are less than n' active elements present, an insert operation will always succeed. Moreover, we change the Lookup procedure to return 'Yes' on input x only if $h(x)$ exists in $\mathcal{D}$ **and** its associated generation is active.

Not all dictionaries can support lazy deletions. In Section 2.6, for concreteness, we describe how to modify the 'Backyard Cuckoo Hashing' dictionary to support it, and discuss a family of dictionaries that may be modified as well.

2.5 Deamortizing the Construction

Currently, as the construction is described, every n' steps we perform a long scanning process. We would like to spread this process over a sequence of n' steps, making every step run in

constant time with no exceptions. To achieve this, we need the scanning process to support running in small steps while allowing other operations to run concurrently.

We should be able to save the scanning state, then allow other operations to run and finally restore the state and continue with scanning process. Instead of scanning all the n' elements in one step, we scan two elements at each step and save the scanning index so that we are able to continue from that point. At any step, all present elements will be scanned after at most $n'/2$ steps. Thus, once an element has become old it will be scanned after at most n' steps: $n'/2$ steps until the next scanning process begins, and $n'/2$ steps for the process to end.

It is not enough to only spread the load of work over n' sequential steps. There is another issue to solve: the scanning is done in small steps concurrently with other operations, which might make it miss elements the have been moved by other operations. For example, an insert procedure might move an element from one cell to another, which was already scanned, causing the scanning process to miss this element. We assume that the scanning process succeeds in scanning all elements nevertheless. In the next section, we show how this can be achieved by the [ANS10] dictionary and others.

2.6 Using a Concrete Dictionary

We discuss implementing the requirements of lazy deletions and concurrent scanning needed in Sections 2.4 and 2.5 in the 'Backyard Cuckoo Hashing' construction (see the pseudo-code in Figure 2 of their paper). Their scheme contains 4 components: an array T_0 of bins of size d, two arrays, T_1 and T_2 which are used for Cuckoo hashing and a queue Q. Each element is implicitly stored in one of the 4 components, in what we call a 'cell'. Each bin of T_0 contains d cells, any entry of the arrays T_1, T_2 is a cell and any element in Q is a cell. A cell's content plus its location and some other easily accessible information determines the element uniquely. During an insert operation elements might move to a new cell within their component or to a cell in a different components. To support lazy deletions we need to modify the insert procedure to check whether any encountered element is old and needs to be deleted. The insert procedure of [ANS10] first seeks for a vacant cell in a bin in T_0. We modify the procedure so that if an old element is present in the bin, it is deleted and replaced with the new element. The arrays T_1 and T_2 are used for Cuckoo hashing, where elements move from one array to the other until they all have a valid cell. We make a similar modification so that before moving an element between the arrays, it is checked and deleted if necessary. We also handle the queue in the same manner. The results is the non active elements do not interfere with new elements being inserted in to the dictionary. If there is less than n' active elements present, the insert procedure will succeed.

The second requirement we need to implement is concurrent scanning. Scanning is performed by enumerating all cells one by one. In each step we scan a few cells and save the index of the last scanned cell so that we can continue from same point in the next step. Note that between two steps of the scanning process elements may move around by an insert procedure, making the scanning miss elements. However, recall that the insert procure was modified to check each element it encounters. This way, no element is missed: each element is scanned either by the scanning process or by an insert procedure.

After these modifications, the 'Backyard Cuckoo Hashing' dictionary supports all needed

requirements for our construction of an (n, ε)-Sliding Bloom Filter. The additional memory required for the modifications is negligible (only $O(\log n)$ for the counters).

We analyze the running time of the modified construction. At each step, we scan two elements and delete them if necessary. The delete operation always takes constant time. The insert procedure was modified to delete old accessed elements when encountered. Since the insert operation takes constant time in worst case with high probability, then with the same probability, it will access only a constant number of cells. Hence, the insert procedure increases only by a constant factor. Similarly, the modified lookup procedure also remains constant. Overall, all operations are constant in the worst case, while the insert operation has constant running time, with high probability. This completes the proof of Theorem 1.1.

It is not clear whether all dictionaries can be modified to support these requirements, since the dictionary might have some implicit representation of various elements using the same memory space. However, many dictionaries are indeed applicable. The main properties of [ANS10] used in that the memory is divided into cells, and different elements are store in different cells. It seems reasonable to assume that any dictionary with similar properties may be modified to support these requirements.

3 A Tight Space Lower Bound

In this section we present a matching space lower bound to our construction. For simplicity, we first introduce what we call the 'absolute false positive assumption'. We define it and use it in the proof of Section 3.1, and in Section 3.2 we show how to get the same lower bound without it.

Recall that at any point in time, the false positive rate is the fraction of false positive elements. According to the definition of a Sliding Bloom Filter, we are not assured that there are no 'bad' points in time where the false positive rate is much higher than its expectation, and in fact it could get as high as 1.

We call the property that throughout the lifetime of the data structure at *all* points in time the false positive rate is at most ε the *absolute false positive assumption*. This assumption is a desirable property from a Sliding Bloom Filter and reasonable constructions, including ours[2] in Section 2, enjoy it.

An (artificial) example of a Sliding Bloom Filter for which the assumption does not hold can be obtained by taking any (n, ε)-Sliding Bloom Filter and modifying it such that it chooses a random index $k \in [1, n]$ and at step k of the stream it always answers 'Yes'. This results in an $(n, \varepsilon + \frac{1}{n})$-Sliding Bloom Filter in which there will *always* be some point at which the false positive rate is high.

3.1 Proof Under the Absolute False Positive Assumption

Theorem 3.1. *Let A be an (n, m, ε)-Sliding Bloom Filter where $n < \varepsilon u$. If for any stream σ it holds that*

$$\Pr[\exists i \leq 3n : |\{x \in U : A(\sigma_i, x) = \text{'}Yes\text{'}\}| \geq n + 2\varepsilon u] \leq \frac{1}{2}$$

[2]See discussion at Section 2.1

then

1. *If* $m > 0$ *then* $|A| \geq n \log \frac{1}{\varepsilon} + n \cdot \max\left\{\log \frac{n}{m}, \log\log \frac{1}{\varepsilon}\right\} - O(n)$
2. *If* $m = \infty$ *then* $|A| \geq n \log \frac{1}{\varepsilon} + n \log\log \frac{1}{\varepsilon} - O(n)$

Proof. Let A be an algorithm satisfying the requirements in the statement of the theorem. The main idea of the proof is to use A to encode and decode a set $S \subset U$ and a permutation π on the set (i.e. an ordered set). Giving S to A as a stream, ordered by π, creates an encoding of an approximation of S and π: S is approximated by the set of all the elements for which A answers 'Yes' (denoted by $\mu_A(S)$), and π is approximated by the number of elements needed to be added to the stream in order for A to "release" each of the elements in S (that is, to answer 'No' on it). Then, to get an exact encoding, we encode only the elements of S from within the set $\mu_A(S)$. To get an exact encoding of π we encode only the difference between the location i of each element and the actual location it has been released. The key is to find the point where A best approximates S and π *simultaneously*.

Denote by A_r the algorithm with fixed random string r and let $\mu_{A_r}(\sigma) = \{x : A_r(\sigma, x) = \text{'Yes'}\}$. We show that w.l.o.g. we can consider A to be deterministic. Let $V = \{\sigma : |\sigma| = 2n\}$ be the set of all sequences of $2n$ *distinct* elements, and let $V(r) \subseteq V$ be the subset of inputs such that $|\mu_{A_r}(\sigma_i)| \leq n + 2\varepsilon u$ for all $1 \leq i \leq 3n$. Since we assumed that for any σ we have that $\Pr_r[\exists i \leq 3n : |\mu_{A_r}(\sigma_i)| \geq n + 2\varepsilon u] \leq \frac{1}{2}$ then there must exist an r^* such that $|V(r^*)| \geq |V|/2$. Thus, we can assume that A is deterministic and encode only sequences from $V(r^*)$. Then the encoding lower bound changes from $\log |V|$ to $\log(|V|/2) = \log|V| - 1$. This loss of 1 bit is captured by the lower order term $O(n)$ in the lower bound, and hence can be ignored.

Notice that r^* need not be explicitly specified in the encoding since the decoder can compute it using the description of the algorithm A (which may be part of its fixed program). From now on, we assume that A is deterministic (and remove the A_r notation) and assume that for any $\sigma \in V(r^*)$ we have that $\mu_A(\sigma) \leq n + 2\varepsilon u \leq 3\varepsilon u$.

We now make an important definition:

$$\ell(\sigma, x) = \min\left\{\arg\min_k \{\exists y_1, \ldots, y_k \in U : A(\sigma y_1 \cdots y_k, x) = 0\}, n, m\right\}$$

$\ell(\sigma, x)$ is the minimum number of elements needed to be added to σ such that A answers 'No' on x. Notice that $\ell(\sigma, \cdot)$ can be computed for any set S given the representation of $A(\sigma)$.

We encode any set S of size $2n$ and a permutation $\pi : [2n] \to [2n]$ using A. After encoding S we compare the encoding length to the entropy lower bound of $\mathcal{B}(u, 2n) + \log((2n)!)$. Consider applying π on (some canonical order of) the elements of S and let $x_1, \ldots, x_{2n}$ be the resulting elements of S ordered by π. For any $i > 2n$ let $x_i = x_{i-2n}$, then for any $k \geq 1$ define the sequence $\sigma_k = x_1, \ldots, x_k$. Let $\phi(\sigma_k) = \mu(\sigma_k) \cap S$ and define

$$\Delta(\sigma_k, i) = \ell(\sigma_k, x_i) + (k - n) - i$$

Notice that, given $A(\sigma_k)$, $\Delta(\sigma_k, i)$, k and n one can compute the position i of the element x_i. Define

$$\lambda_k = \sum_{i=k-n+1}^{k} \Delta(\sigma_k, i), \text{ and } \lambda = \max_{n \le k \le n} \lambda_k$$

If $m \ge n$ (or $m = \infty$) then $0 \le \lambda \le n^2$, otherwise $0 \le \lambda \le nm$

Lemma 3.2. *Let $k \in [n, 2n]$ then*

$$\sum_{j=k}^{k+n-1} |\phi(\sigma_j)| \ge n^2 + \lambda_k.$$

Proof. Instead of summing over $\phi(\sigma_j)$, we sum over x_i and count the number of $\phi(\sigma_j)$ such that $x_i \in \phi(\sigma_j)$. For $k-n+1 \le i \le k$ we know that $x_i \in \sigma_k(n)$ and by the definition of $\ell(\sigma_k, x_i)$ we get that $x_i \in \phi(\sigma_k), \ldots, \phi(\sigma_{k+\ell(\sigma_k,x_i)-1})$. For $k+1 \le i \le k+n-1$ we know that $x_i \in \phi(\sigma_i), \ldots, \phi(\sigma_{k+n-1})$. Therefore:

$$\begin{aligned}
\sum_{j=k}^{k+n-1} |\phi(\sigma_j)| &\ge \sum_{i=k-n+1}^{k} \ell(\sigma_k, x_i) + \sum_{i=k+1}^{k+n-1} (k+n-i) \\
&= \sum_{i=k-n+1}^{k} \ell(\sigma_k, x_i) + \frac{n(n-1)}{2} \\
&= \sum_{i=k-n+1}^{k} [\ell(\sigma_k, x_i) + k - n - i] + n^2 \\
&= \sum_{i=k-n+1}^{k} \Delta(\sigma_k, i) + n^2 = \lambda_k + n^2
\end{aligned}$$

□

By averaging, we get that for any k there exist some $j \in [k, k+n-1]$ such that $|\phi(\sigma_j)| \ge n + \frac{\lambda_j}{n}$. Let k^* be such that $\lambda = \lambda_{k^*}$, then we know that there exist some $j^* \in [k^*, k^*+n-1]$ such that $|\phi(\sigma_j)| \ge n + \frac{\lambda}{n}$. Note that j^* satisfies $n \le j \le k^* + n - 1 \le 3n$ which is in the range of indices of the false positive assumption.

We include the memory representation of $A(\sigma_{j^*})$ in the encoding. The decoder uses this to compute the set $\mu(\sigma_{j^*})$, which by the absolute false positive definition we know that $|\mu(\sigma_{j^*})| \le 3\varepsilon u$. Since $|\phi(\sigma_{j^*})| \ge n + \frac{\lambda}{n}$, we need only $\mathcal{B}(3\varepsilon u, n + \frac{\lambda}{n})$ bits to encode $n + \frac{\lambda}{n}$ elements of S out of them. The remaining $n - \frac{\lambda}{n}$ elements are encoded explicitly using $\mathcal{B}(u, n - \frac{\lambda}{n})$ bits. This completes the encoding of S.

To encode π we need the decoder to be able to extract i for each x_i. For any $x_i \in \sigma_{j^*}(n)$ the decoder uses $A(\sigma_{j^*})$ and computes $\ell(\sigma_{j^*}, x_i)$. Now, in order for the decoder to exactly decode i we need to encode all the $\Delta(\sigma_{j^*,i})$'s. Since $\sum_{i=j^*-n+1}^{j^*} \Delta(\sigma_{j^*,i}) = \lambda_{j^*} \le \lambda$ we can encode all the $\Delta(\sigma_{j^*,i})$'s using $\log \binom{n+\lambda}{n}$ bits (balls and sticks method), and the remaining elements' positions will be explicitly encoded using $n \log n$ bits. Denote by $|A|$ the number

of bits used by the algorithm A. Comparing the encoding length to the entropy lower bound we get

$$|A| + \log \binom{3\varepsilon u}{n + \frac{\lambda}{n}} + \log \binom{u}{n - \frac{\lambda}{n}} + \log \binom{\lambda + n}{n} + n \log n \geq \log \binom{u}{2n} + \log ((2n)!)$$

and therefore

$$|A| \geq (n + \frac{\lambda}{n}) \log \frac{1}{\varepsilon} + (n + \frac{\lambda}{n}) \log n + (n - \frac{\lambda}{n}) \log (n - \frac{\lambda}{n}) - n \log (\lambda + n) - O(n)$$

Consider two possible cases for λ. If $\lambda \leq 0.9n^2$ then we get

$$|A| \geq (n + \frac{\lambda}{n}) \log \frac{1}{\varepsilon} + 2n \log n - n \log (\lambda + n) - O(n)$$

The minimum of this expression, as a function of λ, is achieved at $\lambda = \frac{n^2}{\log \frac{1}{\varepsilon}} - n$. If $m \geq \frac{n}{\log \frac{1}{\varepsilon}} - 1$ then the minimum can be achieved and we get that

$$|A| \geq n \log \frac{1}{\varepsilon} + n \log \log \frac{1}{\varepsilon} - O(n).$$

Otherwise, if $m < \frac{n}{\log \frac{1}{\varepsilon}} - 1$ then $\lambda \leq mn \leq \frac{n^2}{\log \frac{1}{\varepsilon}} - n$ and minimum value will be achieved at $\lambda = nm$ which yields the required lower bound:

$$|A| \geq n \log \frac{1}{\varepsilon} + n \log \frac{n}{m} - O(n).$$

If $0.9n^2 < \lambda \leq n^2$ then $m \geq 0.9n \geq \frac{n}{\log \frac{1}{\varepsilon}}$. Thus, we get that

$$|A| \quad \geq \quad (n + \frac{\lambda}{n}) \log \frac{1}{\varepsilon} - (n - \frac{\lambda}{n}) \log n + (n - \frac{\lambda}{n}) \log (n - \frac{\lambda}{n}) - O(n)$$

the minimum of this expression, as a function of λ between the given range is achieved at $\lambda = 0.9n^2$ which yields

$$|A| \geq n \log \frac{1}{\varepsilon} + n \log \log \frac{1}{\varepsilon} - O(n)$$

as required. □

In the proof, we encoded a sequence of length $2n$ and we assumed that the false positive assumption holds for any such sequence. However, the only property used was the number of bits required for encoding any possible sequence. Since the lower bound includes a $O(n)$ term, we conclude that the theorem holds even for smaller sets of sequences resulting in a larger constant hidden in the $O(n)$ term. In particular, we get the following corollary, which we use to prove Theorem 1.2:

Corollary 3.3. *Let W be a subset of sequences of length $2n$ such that $\log|W| = 2n\log u - O(n)$. Let A be an (n, m, ε)-Sliding Bloom Filter where $n < \varepsilon u$. If for any $\sigma \in W$ it holds that*

$$\Pr[\exists i \leq 3n : |\{x \in U : A(\sigma_i, x) = \text{`}Yes'\}| \geq n + 2\varepsilon u] \leq \frac{1}{2}$$

then

1. *If $m > 0$ then $|A| \geq n\log\frac{1}{\varepsilon} + n \cdot \max\left\{\log\frac{n}{m}, \log\log\frac{1}{\varepsilon}\right\} - O(n)$*
2. *If $m = \infty$ then $|A| \geq n\log\frac{1}{\varepsilon} + n\log\log\frac{1}{\varepsilon} - O(n)$*

3.2 Removing the Absolute False Positive Assumption

We show how we can remove the 'absolute false positive' assumption while maintaining the same lower bound as in the original theorem. Towards this end, we construct a new data structure A' which uses multiple instances of A. The new data structure A' will work only on a specific subset of all inputs, however, we show that the number of such inputs is approximately the same and hence the same entropy lower bound holds, up to a larger constant hidden in the $O(n)$ term. Moreover, we show that on these inputs, the absolute false positive assumption holds and thus we can apply Theorem 3.3 on A'. We restate and prove the theorem.

Theorem 1.2 (Restated). *Let A be an (n, m, ε)-Sliding Bloom Filter where $n < \varepsilon u$, then*

1. *If $m > 0$ then $|A| \geq n\log\frac{1}{\varepsilon} + n \cdot \max\left\{\log\frac{n}{m}, \log\log\frac{1}{\varepsilon}\right\} - O(n)$*
2. *If $m = \infty$ then $|A| \geq n\log\frac{1}{\varepsilon} + n\log\log\frac{1}{\varepsilon} - O(n)$*

Proof. In order to prove the result we need to reduce the probability of having many false positives to roughly $1/n$. To obtain this sort of bound, we partition the sequence into several subsequences on which we apply the original Sliding Bloom Filter independently. The motivation of using multiple instances of A is to introduce independence between different sets of inputs.

Let U' be a universe composed of w copies of U, where w will be determined later. We denote each copy by U_i and we call it a world. Each world is of size u and U' is of size $u' = wu$. We consider only sequences such that each chunk of w elements in the sequence contain exactly one element from each world. These are the only sequences the are valid for A', and we denote them by W. Let $A_1, \ldots, A_w$ be w independent instances of the algorithm A with parameters (n, m, ε).

A' works by delegating each input to the corresponding instance of A. On input $x \in U_i$ we insert x into A_i, and on query $x \in U_i$ we query A_i and return its answer.

Claim 3.4. *Let $n' = (n-1)w$ and $m' = (m+2)w$. Then, A' is an (n', m', ε)-Sliding Bloom Filter for any sequence $\sigma \in W$.*

Proof. We show that the two properties hold for any time t in the sequence. Let $\sigma \in W$, let $x \in U'$ such that $x \in U_i$, and let σ^i be the sequence σ limited to elements in U_i. A' answers

by A_i and thus $\Pr[A'(\sigma, x) = 1] = \Pr[A_i(\sigma^i, x) = 1]$. We analyze $\Pr[A_i(\sigma^i, x) = 1]$ in the different cases.

Since each chuck of w elements contain one element from each world, each data structure will contain at most $n'/w + 1 = n$ elements of the current window. Moreover, each element of the window will be present in one of the A_i's. If $x \in \sigma_t(n')$ is in the current window, then since each A_i has no false negatives we have $\Pr[A_i(\sigma^i, x) = \text{`Yes'}] = 1$.

Now suppose $x \notin \sigma_t(n' + m')$ is not in the current window and not in the m' element beforehand. If $x \notin \sigma_t$ then, by the false positive probability of A_i, we have that $\Pr[A_i(\sigma^i, x) = \text{`Yes'}] \leq \varepsilon$. Otherwise, $x \in \sigma_t$ but $x \notin \sigma_t(n' + m')$, and therefore at least $n' + m'$ elements have arrived after x. Thus, each A_i has received at least $\frac{n'+m'}{w} - 1 = n + m$ elements, and so $x \notin \sigma_t^i(n + m)$. Since each A_i is an (n, m, ε)-Sliding Bloom Filter we have that $\Pr[A_i(\sigma_i, x) = \text{`Yes'}] \leq \varepsilon$ □

We have shown that A' satisfies that properties of an (n', m', ε)-Sliding Bloom Filter for sequences of W. Now, we show that the false positives assumption holds for A' under sequences of W.

For any sequence $\sigma \in W$, for all $1 \leq i \leq w$, let X_i be a random variable indicating the number of false positives of A_i in U_i. Let $X = \sum_{i=1}^{w} X_i$ be the total number of false positives of A'. We bound the probability that X is too high.

Claim 3.5. *For $w = \frac{\log(6n')}{\varepsilon^2}$ we have that $\Pr[X > 3\varepsilon u'] \leq \frac{1}{6n'}$*

Proof. Define $Y_i = \frac{1}{u} \sum_{j=1}^{i} [X_i - \mathbb{E}[X_i]]$ and $Y_0 = 0$. Since $\mathbb{E}[X_i] \leq \varepsilon u$ we get that

$$\Pr[X > 3\varepsilon u'] = \Pr\left[\sum_{i=1}^{w} X_i - \sum_{i=1}^{w} \mathbb{E}[X_i] > 3\varepsilon u' - \sum_{i=1}^{w} \mathbb{E}[X_i]\right] \leq \Pr[Y_w > 2\varepsilon w]$$

To bound Y_w, we use Azuma's inequality (in the form of [AS11, Theorem 7.2.1]). First, note that $\{Y_i : i = 1, \ldots, w\}$ is a martingale:

$$\mathbb{E}[Y_{i+1} - Y_i | Y_1, \ldots, Y_i] = \frac{1}{u}\mathbb{E}[X_{i+1} - \mathbb{E}[X_i] | Y_1, \ldots, Y_i] = \frac{1}{u}\mathbb{E}[X_i - \mathbb{E}[X_i]] = 0.$$

Moreover, we have $|Y_{i+1} - Y_i| = |\frac{1}{u}(X_{i+1} - \mathbb{E}[X_i])| \leq 1$. Thus, by Azuma's inequality we get

$$\Pr[X > 3\varepsilon u'] \leq \Pr[Y_w > 2\varepsilon w] \leq e^{-4\varepsilon^2 w} \leq \frac{1}{6n'}$$

which holds for $w \geq \frac{\log(6n')}{\epsilon^2}$. □

Claim 3.6. *The false positive assumption holds for A' for valid sequences. Namely, for any sequence $\sigma \in W$ it holds that*

$$\Pr[\exists i \leq 3n' : |\{x \in U : A'(\sigma_i, x) = \text{`}Yes\text{'}\}| \geq n' + 3\varepsilon u'] \leq \frac{1}{2}$$

Proof. By the previous claim, we know that for any i: $\Pr[|\{x \in U : A'(\sigma_i, x) = \text{`}Yes'\}| \geq n' + 3\varepsilon u'] \leq \frac{1}{6n'}$. Using a union bound we get that

$$\Pr[\exists i \leq 3n' : |\{x \in U : A'(\sigma_i, x) = \text{`}Yes'\}| \geq n' + 3\varepsilon u'] \leq 3n' \cdot \frac{1}{6n'} = \frac{1}{2}.$$

□

We have shown that the false positive assumption holds for all sequences in W. To apply Theorem 3.3 we are left to show that the entropy lower bound is large enough, namely:

Claim 3.7. $\log |W| = 2n' \log u' - O(n')$

Proof. We count the number of possible sequences in W of length $2n'$. First we need to choose $2n'/w$ elements from each of the w worlds. Then, for each $2n'/w$ chosen elements, we divide them between the $2n'/w$ chucks of the sequence, and finally we count all possible orderings of each chuck. Altogether we get:

$$\binom{u}{\frac{2n'}{w}}^w \cdot \left(\frac{2n'}{w}!\right)^w \cdot (w!)^{\frac{2n'}{w}}$$

The entropy lower bound is:

$$2n' \log (u'/2n') + O(n') + 2n' \log (2n'/w) - 2n' + o(n') + 2n' \log w - 2n' + o(n') =$$

$$2n' \log u' - O(n')$$

□

Let $|A'|$ be the memory consumption of A'. We can now apply Theorem 3.3 on A' with the set of sequences W and parameters n', m', u' and get

$$|A'| \geq n' \log \frac{1}{\varepsilon} + n' \cdot \max \left\{ \log \frac{n'}{m'}, \log \log \frac{1}{\varepsilon} \right\} - O(n').$$

Since $|A'| = \sum_{i=1}^{w} |A_i|$ we get that there exist some i such that

$$|A_i| \geq n'/w \log \frac{1}{\varepsilon} + n'/w \cdot \max \left\{ \log \frac{n'/w}{m'/w}, \log \log \frac{1}{\varepsilon} \right\} - O(n'/w) =$$

$$n \log \frac{1}{\varepsilon} + n \cdot \max \left\{ \log \frac{n}{m}, \log \log \frac{1}{\varepsilon} \right\} - O(n)$$

Since A_i is an (n, m, ε)-Sliding Bloom Filter we get the desired lower bound. □

References

[ANS10] Yuriy Arbitman, Moni Naor, and Gil Segev, *Backyard cuckoo hashing: Constant worst-case operations with a succinct representation*, FOCS, 2010, pp. 787–796.

[AS11] Noga Alon and Joel H. Spencer, *The probabilistic method*, Wiley Series in Discrete Mathematics and Optimization, Wiley, 2011.

[Blo70] Burton H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, ACM Press **13** (1970), no. 7, 422–426.

[BM03] Andrei Z. Broder and Michael Mitzenmacher, *Survey: Network applications of Bloom filters: A survey*, Internet Mathematics **1** (2003), no. 4, 485–509.

[CFG+78] Larry Carter, Robert W. Floyd, John Gill, George Markowsky, and Mark N. Wegman, *Exact and approximate membership testers*, STOC, 1978, pp. 59–65.

[CLF04] Francis Chang, Kang Li, and Wu-chang Feng, *Approximate caches for packet classification*, INFOCOM, 2004.

[CW79] J Lawrence Carter and Mark N Wegman, *Universal classes of hash functions*, Journal of computer and system sciences **18** (1979), no. 2, 143–154.

[Dem07] Erik Demaine, *Lecture notes for the course "Advanced data structures"*, available at http://courses.csail.mit.edu/6.851/spring07/scribe/lec21.pdf (2007).

[DGIM02] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani, *Maintaining stream statistics over sliding windows*, SIAM Journal on Computing **31** (2002), no. 6, 1794–1813.

[DP08] Martin Dietzfelbinger and Rasmus Pagh, *Succinct data structures for retrieval and approximate membership*, ICALP, 2008, pp. 385–396.

[DR06] Fan Deng and Davood Rafiei, *Approximately detecting duplicates for streaming data using stable Bloom filters*, SIGMOD, 2006, pp. 25–36.

[FCAB00] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder, *Summary cache: a scalable wide-area web cache sharing protocol*, IEEE/ACM Transactions on Networking **8** (2000), no. 3, 281–293.

[LP10] Shachar Lovett and Ely Porat, *A lower bound for dynamic approximate membership data structures*, FOCS, 2010, pp. 797–804.

[MAEA05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi, *Duplicate detection in click streams*, Proceedings of the 14th international conference on World Wide Web, ACM Press, 2005, pp. 12–21.

[PPR05] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao, *An optimal Bloom filter replacement*, SODA, 2005, pp. 823–829.

[PSW13] Rasmus Pagh, Gil Segev, and Udi Wieder, *How to approximate a set without knowing its size in advance*, arXiv report 1304.1188, to appear in FOCS (2013).

[Tho11] Mikkel Thorup, *Timeouts with time-reversed linear probing*, INFOCOM, 2011, pp. 166–170.

[TRL12] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz, *Theory and practice of Bloom filters for distributed systems*, IEEE Communications Surveys and Tutorials **14** (2012), no. 1, 131–155.

[Yoo10] MyungKeun Yoon, *Aging Bloom filter with two active buffers for dynamic sets*, IEEE Transactions on Knowledge and Data Engineering **22** (2010), no. 1, 134–138.

[ZG08] Linfeng Zhang and Yong Guan, *Detecting click fraud in pay-per-click streams of online advertising networks*, ICDCS, 2008, pp. 77–84.