Stacked Filters for Fewer False Positives in Approximate Membership Queries

Kyle Deeds
kdeeds@college.harvard.edu
Harvard University

Brian Hentschel
bhentschel@g.harvard.edu
Harvard University

Stratos Idreos
stratos@seas.harvard.edu
Harvard University

ABSTRACT

Approximate Membership Query (AMQ) data structures such as Bloom Filters are used across a variety of applications in business and science to provide cheap probabilistic set membership checks with limited false positives and no false negatives. Because these filters typically reside in memory, it is a critical open problem to reduce their space requirements as data continues to grow. Recent work in learned bloom filters shows that we can substantially reduce the size of filters using ML. However, the amount of compression is heavily dependent on the structure in the dataset and learned filters can suffer under shifting workloads, for which they lack a bound on the expected false positive rate.

In this work, we present a new filter family which we call Stacked Filters. We show that Stacked Filters achieve similar compression ratios and false positive rates to learned filters while remaining nearly as robust to noisy data and shifting workloads as traditional filters. Our construction works by stacking alternating filters built from both the positive set and a (partial) set of known negative elements. Only elements that are incorrectly accepted by previous layers are added to later ones, exponentially reducing the size of the layers as you descend down the stack. The resulting stack has a very low false positive rate for elements in the set of known negatives, and, therefore, achieves a significant reduction in the expected false positive rate when the set of known negatives captures a large proportion of the negative queries.

Compared to a traditional Bloom Filter, our construction reduces the number of bits needed to achieve a given expected false positive rate. Compared to learned filters, our construction provides resiliency to changing input query distributions, and utility for hashed or un-learnable data. We give a detailed description and analysis of Stacked Filters as well as detailed experimentation in two applications: URL blacklisting and LSM trees.

CCS CONCEPTS

•Information systems → Data structures; •Mathematics of computing → *Probabilistic algorithms;* **•Networks →** Network algorithms;

KEYWORDS

Bloom Filters, URL Blacklisting, AMQ Structures, Data Systems

ACM Reference format:
Kyle Deeds, Brian Hentschel, and Stratos Idreos. 2020. Stacked Filters for Fewer False Positives in Approximate Membership Queries. In *Proceedings of SIGMOD '20: ACM SIGMOD/PODS International Conference on Management of Data, Woodstock, NY, June 03–05, 2018 (SIGMOD '20),* 14 pages.
DOI: 10.1145/1122445.1122456

1 INTRODUCTION

Filters are Everywhere. The storage and retrieval of items in a data set is one of the most fundamental operations in computer science applications, systems, and ad-hoc programs. For large data sets, the raw data is usually stored over a slow medium (e.g., on disk because not everything fits in memory) or distributed across the nodes of a network. Because of this, it is critical for performance to limit accesses to the full data set when the queried element is not present in the data set. This is the exact utility of approximate membership query (AMQ) structures, which provide answers to whether a queried element is in some data set with no false negatives and a limited number of false positives in exchange for remaining small enough to be memory resident. Thus AMQ structures are effectively filters that help applications reduce the number of calls to expensive operations, and they are are used in myriad applications such as distributed joins[21][20], web caching[10], prefix matching[7], deduping data[6], DNA classification[23], detecting flooding attacks[12], Bitcoin transaction verification[13], and LSM trees[5], amongst many others[24].

The Traditional Approach: Using the Positive Set. Traditional AMQ structures such as Bloom, Quotient, or Cuckoo

Filters require only the encoded data set in order to be constructed[1][18][9]. For example, a Bloom Filter starts with an array of bits set to 0 and hashes the elements of the encoded data set, referred to henceforth as the positive set, to various positions in an array of bits, setting those bits to 1. Queries are then applied to the filter by using the same hash functions and return positive if all the bits the query element hashes to are all set to 1. This ensures that every element which has been entered into the array is counted as a positive, but it allows for false positives if a negative element is hashed entirely to positions which were set to 1 during construction.

Learning from the Negative Set. As data sizes grow, it is critical to minimize the size of these filters further so that we can represent more data while keeping the filter in memory. In addition, traditional filters do not take into account the characteristics of the existing application. They allow tuning knobs such as the number of bits per data entry but cannot account for the kinds of queries expected in a particular application. Recent advances in the area of Learned Filters make significant steps forward for both problems. A Learned Filter utilizes application workload knowledge, i.e. past positive and negative queries.[15][17] It trains a binary classifier on a sample of past queries, and stores any misclassified elements from the positive set in a backup bloom filter. The backup filter retains the property that set membership queries which return False are certain, i.e. there are no false negatives. When a new element is queried against the learned filter, it is first checked against the classifier and accepted outright if the classifier labels it as a positive. If the classifier labels it as a negative, the element is checked against the backup filter and accepted or rejected based on the filter's decision. Using this method, the required memory to achieve a given false positive rate can be reduced by upwards of 60% depending on the training data available and how learnable the distinction between positive and negative elements is. This approach often works because an effective binary classifier can correctly classify a large number of the positive elements without querying against, and therefore needing to enter them into, the backup filter, while accepting very few negative elements. Because the size of the backup filter scales linearly with the number of elements entered into it and models are often relatively small in comparison, this means that the space allocated to the filter and model can be much less than that of an equivalent standard filter.

The Open Problem. While Learned Filters offer a solution to the problem of size and allow for the utilization of workload knowledge, they also bring a new set of open challenges which can be very critical for certain applications. First, Learned Filters come with increased computational overhead. Training and querying a complex model such as a neural network is substantially more expensive than creating and querying a hash based filter. Second, by definition a Learned Filter is tailored to a particular workload. However, there are many applications where the workload my shift over time, e.g., as data and query patters evolve; for Learned Filters this would mean that the model must be retrained or otherwise suffer a dramatic degradation in the false positive rate. Overall for applications with dynamic workloads, periodic workload shifts, or where by nature we need to continuously recompute the filters (e.g., during every compaction in LSM-tree based key-value stores), state of the art Learned Filters pose several open problems. At the same time, using traditional filters such as Bloom Filters brings us back to having a large memory footprint and being unable to use workload knowledge to reduce the false positive rate.

The Solution: Stacked Filters. Ideally, a filter would be as small and as accurate for targeted applications as a learned filter while at the same time as robust to workload changes and efficient in CPU usage as traditional Bloom Filters. In other words, it would have the best properties of both worlds. In this paper, we show that this is possible to achieve.

We introduce a new class of filters which we call Stacked Filters and we show how it blends the best properties of Learned Filters and traditional filters. Like Learned Filters, Stacked Filters utilize workload knowledge of past positive and negative queries. A Stacked Filter is constructed as a series of Bloom Filters alternately representing elements of the positive and negative sets. At each layer of a Stacked Filter after the first, the local AMQ, generally a Bloom Filter, contains only elements which have been labeled falsely by the filter in the previous layer (the first layer stores the full positive set).

Because every layer of the Stacked Filter only represents elements which passed the membership checks of all previous layers, the filter of every layer becomes exponentially smaller as we move down the stack. Therefore, the additional overhead of the layers beyond the first is minimal, and, because the first layer can now be less precise, this results in a significant size reduction. At the same time, a query in the set of known negative elements has to pass the membership checks of all filters in order to appear as a false positive. When this set of known negatives represents a large portion of the negative query distribution, this design lowers the FPR dramatically, improving over traditional filters and matching Learned Filters.

On the other hand, the CPU cost of Stacked Filters is much lower than Learned Filters. With Stacked Filters we only need to check a small number of traditional, highly performant filters compared to the numerous computations required to go through a neural network in a Learned Filter.

Additionally, by tuning the false positive rates of the AMQs used in the stack, the approach of alternating stacked filters can be tuned to be robust to workload shift, as traditional filters are, or be tuned to optimize performance on the current workload.

Comparison with Learned Approaches. The method of stacked alternative filters uses only hashing does not encode any structural information about the negative set. Therefore, unlike learned filters, alternating stacked filters does not generalize the FPR improvement to unseen negatives. However, by using traditional hashing, stacked alternating filters remove all the complexities of trying manage learned filters. As we show in Section 6, they are fast and easy to construct. Furthermore, by tuning the filter sizes in the stack, it is possible to optimize the expected false positive rate for the current workload while providing strong guarantees about the expected false positive rate under workload shift.

Contributions.. The contributions of this paper are as follows:

- We demonstrate that no single AMQ structure is optimal under all workloads, and that these structures can be made significantly more space efficient when tailored to a specific workload.
- We show that the Stacked Filter achieves comparable space efficiency to Learned Bloom Filter variants while maintaining computational performance near that of state of the art traditional filters by taking advantage of an alternating, layered structure which only stores the false positives of previous layers. The utility of this is verified in two practical contexts: URL blacklisting and LSM trees.
- We provide a theoretical analysis of the Stacked Filter's false positive rate, space, and computational cost as well as a theoretical guarantee of its resiliency under changing workloads.
- We explain how the Stacked Filter structure changes the solution space of AMQ structures under various workloads using the definition of performance optimal filtering provided in Lang, et al.[16]

2 BACKGROUND

2.1 Notation and Metrics

We now provide the necessary background. We first introduce notation used throughout the paper and metrics that are critical for describing the behavior of the filters. Then we explain in more detail how traditional Bloom Filters work.

Notation. Let $\mathcal{U}$ be the universe of elements, such as the domain of strings or integers. Let P be an input set of elements, which we will refer to as the positive elements, and let $N = U - P$ be the set of negative elements. Additionally, let $N_c \subset N$ be a set of chosen negatives. We will be building AMQ data structures, which we will denote by S, and which we treat as a function from $U \rightarrow \{0, 1\}$. As an AMQ data structure, we have $S(x) = 1 : \forall x \in P$ and we will be interested in minimizing the number of false positives, which are the event $S(x) = 1, x \notin P$. We note that the AMQ structure S is itself random; different instantiations of S will produce different data structures, either because the hash functions used have randomly chosen parameters or because the machine learning model used in learned bloom filters is stochastic.

Metric 1: False Positive Bound. The traditional guarantee of an AMQ data structure S is to bound $E_S[P(S(x) = 1|x \notin P)]$ for any x chosen independently of the creation of S. For traditional AMQ data structures such as Bloom Filters, this assumption is identical to x independent of the hash functions used in the creation of S. The produced bound we will describe as the *false positive bound* (FPB)

Metric 2: Expected False Positive Rate. Given a distribution D over N which captures the query probabilities for elements in N, the *expected false positive rate* (EFPR) is $E_{x \sim D}[E_{S|D}[P(S(x) = 1|x \notin P)]]$. The EFPR captures in a distribution specific manner the rate at which false positives are expected to occur.

Optimization through Expected False Positive Rate. Since the expected false positive rate correlates directly with system throughput, it makes the most sense to optimize it. Given the metric's direct reliance on D, it makes sense to build our AMQ structure S in a way that exploits D: namely, for $x \in N$ with higher chance of being queried, it should lower the probability that x is a false positive. This can be achieved through machine learning, as in [15], or via hashing, as we show in Section 3.

Robustness through Bounding False Positive Probability. Optimizing the expected false positive rate should help system throughput, however, it opens up a system to concerns about workload shift. Traditional AMQ data structures can act as a safeguard against such a shift. To see this, note that $S \perp D$ by assumption and so $E_{x \sim D}[E_{S|D}[P(S(x) = 1|x \notin P)]] \leq E_{x \sim D}[E_{S|D}[\epsilon]] = \epsilon$ regardless of what D is. Thus, AMQ data structures which provide a false positive bound provide an upper bound on the expected false positive rate for any workload D chosen independently of the AMQ structure S.

2.2 Bloom Filters

A Bloom filter consists of an array of bits initially set to 0 and allows two operations, insertion and lookup. When inserting an element, the element is hashed using k hash functions to k separate bit locations in the array. The bits at those locations are set to 1. When looking up an element, the element uses the same k hash functions to get k bit locations. It then checks these locations and if any location contains a 0 bit, the query returns 0. If all k bits are set, the query returns 1. The potential for a false positive arises from the possibility that an element in the negative set is hashed to k locations which have been flipped to 1 by elements in the positive set. The larger an array is, the smaller the chance of k collisions happening is and thus the smaller the FPR.

Bloom Filter FPB/EFPR calculation. If we assume that the k hash functions used in the Bloom Filter are perfectly random, then all $x \notin P$ have the same probability of being a false positive. Thus, the false positive bound and the expected false positive rate for the Bloom filter are equal, and they are both equal to the probability that any $x \notin P$ produces a false positive.

Let m be the size of our Bloom filter, n be the size of P, and k be the number of hash functions our Bloom Filter uses. Under our assumption of perfect hash functions, the probability that any bit is still 0 after all elements have been inserted is $(1 - \frac{1}{m})^{nk} \approx e^{-kn/m}$. The probability that all k checked bits are 0, using the above approximation and that the number of bits is close to its expectation, is then

$$P(S(x) = 1 | x \notin P) = (1 - e^{-\frac{nk}{m}})^k$$

It is common in practice to then set k to the value which minimizes this equation, which is

$$k = \frac{m}{n} \log(2) \tag{1}$$

In all subsequent equations, we assume that k has this value; for experimental purposes, k must be an integer and we set k to the nearest integer value of (1). Using our non-integer value of k, the EFPR of a Bloom filter is then

$$P(S(x) = 1 | x \notin P) = e^{-\frac{m}{n}(\ln 2)^2} \tag{2}$$

Bloom Filter space requirements. While above we deduced the EFPR from the size of the filter, it is often helpful to set an EFPR first and then determine the filter size necessary to produce it. If we let α be our expected false positive rate and invert 2, then we get

$$m = \frac{-n \ln \alpha}{(\ln 2)^2} \tag{3}$$

As before, this equation is often non-integer. In this case, we round m to the nearest integer.

For a more comprehensive description and analysis of Bloom filters, including a justification of the approximations made above, see [2].

3 STACKED FILTER ALGORITHMS

We begin with some terminology to simplify the following discussion. In this paper, each filter in the stack is referred to as a layer, L_i. Each layer encodes either positive elements or negative elements, and the first layer encodes positive elements. If the layer encodes negative elements we refer to it as a negative layer and if it encodes positive elements we refer to it as a positive layer. Further, we index the layers from zero where the zero-eth layer is the first one that elements are tested against, and the false positive rate of the ith layer is α_i. Because the final filter must be constructed from P in order for the FNR to equal zero, we only examine odd numbers of layers. Lastly, the total number of layers is T_L.

This section describes the algorithms necessary to construct the stacked filters, lookup elements from them, insert positive elements into them, and, if the underlying AMQ structure supports deletion, delete positive elements from them. In each case, we first present the simpler 3-layer case, then expand to an arbitrary number of layers where N denotes the number of layers. For the moment, we assume that the FPR and size of each layer has been predetermined, however, we cover choosing the allocation of space between layers in Section 5.

Algorithm 1 Construct_Filter(P, N_k)

```
Require: $N_k$ and $P$ are the set of negative and positive elements known at construction time.
    $L$ is the array of AMQ structures which makes up the Stacked Filter.
    $T_L$ is the total number of layers.
1:  $S_{N_k}$ = Array($N_k$)   // active negative elements
2:  $S_P$ = Array($P$)   // active positive elements
    // Construct the layers in the filter sequentially.
3:  for $i = 0$ to $i = T_L - 1$ do
4:    if $i \mod 2 = 0$ then
5:      $L[i]$.insert($S_P$) // Add all active positive elements to the current positive filter.
6:      $c = 0$
        // Retain negative elements that are false positives of $L[i]$.
7:      for $j = 0$ to $j = S_{N_k}$.length do
8:        if $L[i]$.lookup($S_{N_k}[j]$) = 1 then
9:          $S_{N_k}[c] = S_{N_k}[j]$
10:         $c = c + 1$
11:   else
12:     $L[i]$.insert($S_{N_k}$) // Add all active negative elements to the current negative filter.
13:     $c = 0$
        // Retain positive elements that are false positives of $L[i]$.
14:     for $j = 0$ $j = S_P$.length do
15:       if $L[i]$.lookup($S_P[j]$) = 1 then
16:         $[c] = S_P[j]$
17:         $c = c + 1$
```

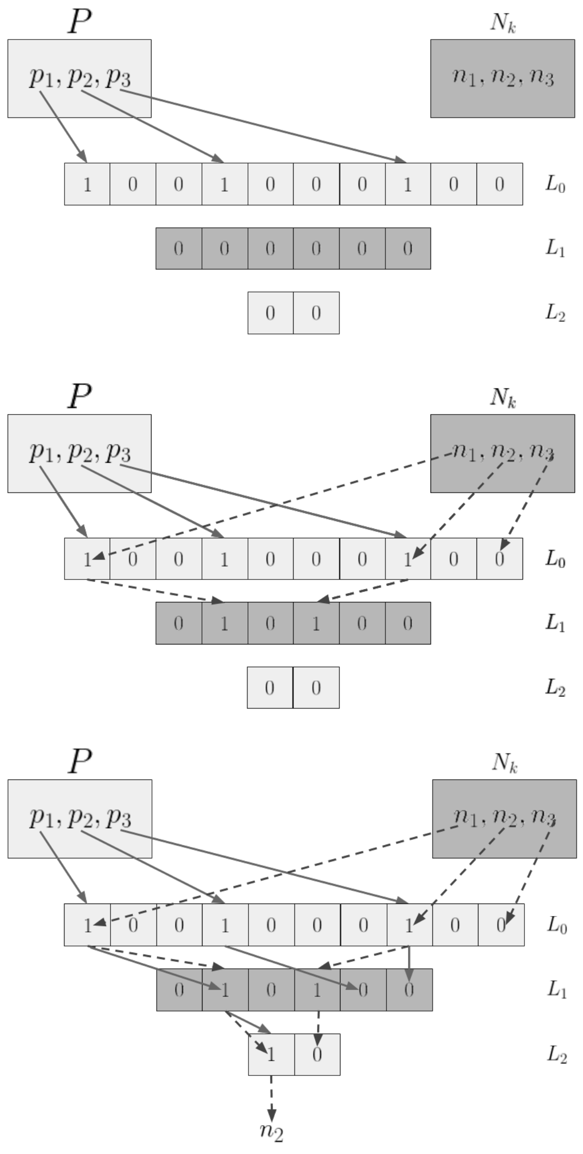

Figure 1: Construction Algorithm

3.1 Construction

We first present the construction algorithm for a simple three layer filter. The first layer is created as it would be in the standard bloom filter by simply inserting each of the elements from the positive set P into L_0. After L_0 is constructed, we take elements from N_k and test them on L_0. For each element in N_k which appears as a false positive for the Bloom Filter in L_0, we add it to the Bloom Filter in L_1. Finally, we take the elements of P and test them against L_1. If they appear as false positives of L_1 (i.e. they appear falsely to be a known negative), then we add them to L_2. This completes the construction of a 3-layer filter.

An example of this construction can be seen in Figure 1 using 3 positive and 3 negative elements. For simplicity, each bloom filter in Figure 1 uses only a single hash function. We can note that of the three elements in N_k, n_2 is the only false positive for the stacked filter because it is the only element which passes through both positive layers.

From this example, a few aspects are important to note. First of all, because of the need to identify the false positives of previous layer before the creation of the next layer, the construction algorithm occurs sequentially. Second, L_1 and L_2 are only storing the false positives of L_0 and L_1, respectively, which means that they are on the order of one hundred times smaller than they would be if they were to store their whole set at the same FPR. Lastly, if the FPR of filters higher in the stack is lower, then even fewer elements reach the lower levels which means that they can in turn can have lower FPRs. This phenomenon will mean that this filter design is particularly effective in comparison to other designs at lower FPRs.

To extend this to an arbitrary number of layers, the process involves keeping an array of the elements of P and N_k that "survive" each layer of the filter and testing/adding only those elements further down the stack. Let i be even, as in the first branch of the if-statement in Algorithm 1. At layer L_i, insert every element in the array for P, then test every item in the array for N_k elements, remove any that are rejected by the filter L_i, and insert the once which pass through L_i into L_{i+1}. Next, test each element in the array for P and remove any that are rejected by layer L_{i+1}. In order to build the entire stack of filters, begin this process at L_0 with every element of P and N_k in their respective arrays, and terminate when the final filter in the stack has been constructed.

Algorithm 2 Lookup(x)

```
Require: L is the array of AMQ structures which makes up
    the Stacked Filter.
    $T_L$ is the total number of layers.
    // Iterate through the layers until one rejects $x_P$.
1: for $i = 0$ to $i = T_L - 1$ do
2:   if L[i].lookup(x)==0 then // If rejected by a positive/negative layer, reject/accept the element.
3:     if $i \mod 2 = 0$ then
4:       return 0
5:     else
6:       return 1
7: return 1 // If end is reached, accept.
```

3.2 Lookup

The lookup algorithm consists of checking the element against each layer of the stack sequentially then returning when 1)

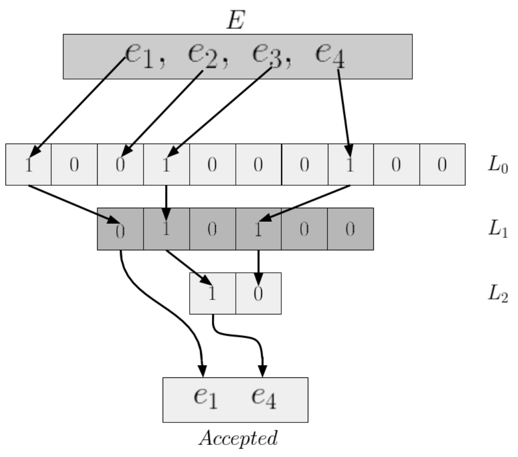

Figure 2: Lookup Algorithm

it is rejected from a layer or 2) it has passed through every layer. If it is rejected from a positive layer then the element is rejected from the whole stack, and if it is rejected from a negative layer or makes it to the end it marked as accepted by the whole stack.

Figure 2 demonstrates using this process to test four random elements against the stacked filter constructed previously. The elements are first tested against L_1 and all succeed except for e_2 which is marked negative because it was rejected from a positive layer. Next, the surviving elements are tested against L_2, and this time e_1 is the only one which is rejected. However, because it is rejected from a negative layer e_1 is marked as a positive and returns. Finally, e_3 and e_4 are tested against L_3 and because e_3 is rejected by a positive layer it is marked as a negative. On the other hand, because e_4 has successfully passed through every layer, it is marked as a positive.

This is extended to an arbitrary number of layers by simply continuing the process of testing against the filters sequentially, marking elements as a positive if they are rejected by a negative layer or make it through every layer, and marking elements as negative if they are rejected by a positive layer.

3.3 Insert Positive Element

Insertion follows the same general path as the construction algorithm except instead of a vector of positive and negative elements it simply does it for a single positive element. The positive element X_p is added to L_0 then tested against L_1. If it passes it is then added to L_2, and it continues being tested against negative layers and added to the positive layers after them until it is rejected from a positive layer or has reached the end of the stack.

Algorithm 3 Insert(x_P)

```
Require: $L$ is the array of AMQ structures which makes up
    the Stacked Filter.
    $T_L$ is the total number of layers.
    $x_P$ is a positive element.
1: for $i = 0$ to $i = T_L - 1$ do // Add $x_P$ to positive layers,
    until it is rejected by a negative layer.
2:   if $i \mod 2 = 0$ then
3:     L[i].insert($x_P$)
4:   else
5:     if L[i].lookup($x_P$)=0 then
6:       return
```

3.4 Delete Positive Element

While the standard bloom filter does not support deletion, many recent AMQ structures such as cuckoo filters and counting quotient filters have support for deletion, so an algorithm is presented here to allow deletion from the stacked filter structure if the individual layers support deletion. It follows the same structure as the insertion algorithm, however, when the insertion algorithm would insert the element into a layer, the deletion algorithm deletes the element from that layer.

Algorithm 4 Delete(x_P)

```
Require: $L$ is the array of AMQ structures which makes up
    the Stacked Filter.
    $T_L$ is the total number of layers.
    $x_P$ is a positive element.
1: for $i = 0$ to $i = T_L - 1$ do // Delete $x_P$ from positive
    layers, until it is rejected by a negative layer.
2:   if $i \mod 2 = 0$ then
3:     L[i].insert($x_P$)
4:   else
5:     if L[i].delete($x_P$)=0 then
6:       return
```

4 THEORETICAL ANALYSIS

4.1 Runtime Analysis

Construction Algorithm Runtime. The expected runtime of the construction algorithm can be calculated by first calculating the expected number of lookups and inserts for one element of P and one element of N_K then multiplying those expectations by $|P|$ and $|N_k|$ and adding the result together. Each element of P will be added to every positive layer it reaches and call the lookup operation of every negative layer it reaches in the algorithm. Therefore, the expected number of lookup operations for each positive element, x_P, is equal to the expected number of negative layers it reaches in the course of the algorithm. This can be found by the summation of the expectation of indicator random variables representing the event that each negative layer is reached. Let I_{i,x_P} be an indicator random variable representing the event that the element x_P reached layer i in the course of the construction algorithm.

$$\text{Lookups}(x_P) = \sum_{i=1}^{\frac{T_L-1}{2}} I_{2i-1,x_P}$$

By the fundamental bridge of probability, the expectation of the indicator random variable is equal to the probability of the event it represents. Therefore, the expectation of the above expression becomes,

$$\mathbb{E}(\text{Lookups}(x_P)) = \mathbb{E}(\sum_{i=1}^{\frac{T_L-1}{2}} I_{2i-1,x_P}) = 1 + \sum_{i=2}^{\frac{T_L-1}{2}} \prod_{j=1}^{i-1} \alpha_{2j-1}$$

The expected number of inserts can be found in the same manner except counting the number of positive layers the element reaches,

$$\mathbb{E}(\text{Inserts}(x_P)) = \mathbb{E}(\sum_{i=0}^{\frac{T_L-1}{2}} I_{2i,x_P}) = 1 + \sum_{i=1}^{\frac{T_L-1}{2}} \prod_{j=1}^{i} \alpha_{2j-1}$$

The number of lookup operations required to insert a negative element,x_{N_k}, is equal to the number of positive layers it reaches, with the exception of the final one. So, similarly to the positive case, we sum the expectation of indicator random variables, $I_{i,x_{N_K}}$ representing whether an arbitrary negative element, x_{N_K}, reaches the ith layer,

$$\mathbb{E}(\text{Lookups}(x_{N_k})) = \mathbb{E}(\sum_{i=0}^{\frac{T_L-1}{2}-1} I_{2i,x_{N_k}}) = 1 + \sum_{i=1}^{\frac{T_L-1}{2}-1} \prod_{j=0}^{i-1} \alpha_{2j}$$

Similarly, the expected number of inserts is equal to the number of negative layers X_N reaches,

$$\mathbb{E}(\text{Inserts}(x_{N_k})) = \mathbb{E}(\sum_{i=1}^{\frac{T_L-1}{2}} I_{2i-1,x_{N_k}}) = \sum_{i=1}^{\frac{T_L-1}{2}} \prod_{j=0}^{i-1} \alpha_{2j}$$

With this, we find the expected total number of lookups and inserts during the construction of the stacked filter,

$$\mathbb{E}(\text{Lookups}) = |N_k|(1+ \sum_{i=1}^{\frac{T_L-1}{2}-1} \prod_{j=0}^{i-1} \alpha_{2j}) + |P|(1+ \sum_{i=2}^{\frac{T_L-1}{2}} \prod_{j=1}^{i-1} \alpha_{2j-1})$$

$$\mathbb{E}(\text{Inserts}) = |N_k|(\sum_{i=1}^{\frac{T_L-1}{2}} \prod_{j=0}^{i-1} \alpha_{2j}) + |P|(1 + \sum_{i=1}^{\frac{T_L-1}{2}} \prod_{j=1}^{i} \alpha_{2j-1})$$

Lookup Algorithm Runtime. The number of single-filter lookup operations required to evaluate the lookup algorithm on the whole stack is determined by which layer of the stack the element is rejected from, if any. If the element x is rejected at layer L_i, then the lookup algorithm required $i + 1$ single-filter lookup operations. Therefore, the expected number of lookup operations is equal to the expected number of layers reached. This quantity depends on whether x is a positive, known negative, or unknown negative, so we handle each case separately. However, the technique is the same as the one used in the analysis of the construction runtime where, in order to get the expected number of layers reached, we sum the expectation of the indicator random variables representing the element reaching each layer. For a positive element, the number of lookups is equal to twice the number of negative layers reached, plus one for the final positive layer if it passes through every negative layer. This is true because for each negative layer it reaches it has passed through the positive layer immediately preceding, i.e. $E(I_{2i,x_P}) = E(I_{2i+1,x_P})$.

$$\mathbb{E}(\text{Lookups}(x_P)) = \mathbb{E}(2(\sum_{i=1}^{\frac{T_L-1}{2}} I_{2i-1,x_P}) + I_{T_L-1,x_P}) = 2+2\sum_{i=2}^{\frac{T_L-1}{2}} \prod_{j=1}^{i-1} \alpha_{2j-1} + \prod_{j=1}^{\frac{T_L-1}{2}}$$

For a known negative, the number of lookups is equal to twice the number of negative layers reached plus one because it passes through every negative layer it reaches and is checked against the following positive layer,

$$\mathbb{E}(\text{Lookups}(x_{N_k})) = \mathbb{E}(I_{0,x_{N_k}} + 2(\sum_{i=1}^{\frac{T_L-1}{2}} I_{2i-1,x_P}) = 1+2\sum_{i=1}^{\frac{T_L-1}{2}} \prod_{j=0}^{i-1} \alpha_{2j}$$

Finally, an unknown negative has not been entered into any layer of the stack, so it must be a false positive of each layer in order to continue down the stack. Therefore, the probability that it reaches any layer is unique, so we sum the indicator random variables for every layer,

$$\mathbb{E}(\text{Lookups}(x_{N \setminus N_k})) = \mathbb{E}(\sum_{i=0}^{T_L-1} I_{i,x_{N \setminus N_k}}) = 1 + \sum_{i=1}^{T_L-1} \prod_{j=0}^{i-1} \alpha_j$$

As the false positive rate of the filters approaches zero, the expected number of lookups for negative elements rapidly approaches 1, and expected the number of lookups for positive elements rapidly approaches 2.

4.2 Zero False Negative Rate

An element is only rejected in the lookup algorithm if it is rejected from a positive layer. Therefore, if every positive layer that a positive element is tested against in the lookup algorithm contains that element, then it cannot be a false negative. The construction algorithm guarantees that this is the case for every positive element, x_P. This can be seen in the following way.

Suppose that x_P is tested against the positive layer L_i in the lookup algorithm. This implies that x_P is not rejected

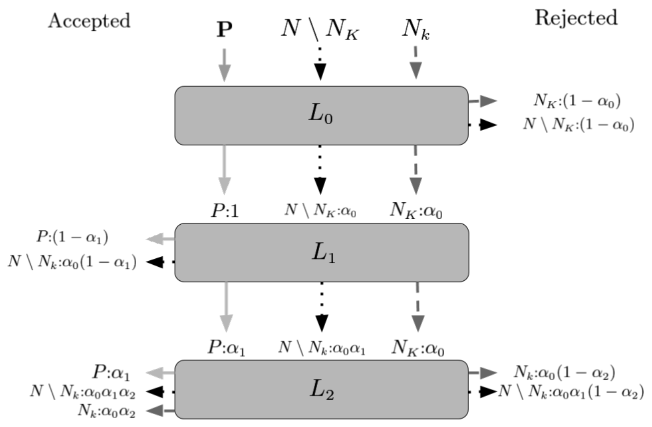
Accepted
P
$N \setminus N_K$
N_k
Rejected
L_0
N_K:$(1-\alpha_0)$
$N \setminus N_K$:$(1-\alpha_0)$
P:1
$N \setminus N_K$:α_0
N_K:α_0
P:$(1-\alpha_1)$
$N \setminus N_k$:$\alpha_0(1-\alpha_1)$
L_1
P:α_1
$N \setminus N_k$:$\alpha_0\alpha_1$
N_K:α_0
P:α_1
$N \setminus N_k$:$\alpha_0\alpha_1\alpha_2$
N_k:$\alpha_0\alpha_2$
L_2
N_k:$\alpha_0(1-\alpha_2)$
$N \setminus N_k$:$\alpha_0\alpha_1(1-\alpha_2)$

Figure 3: Lookup Algorithm Pathway Probabilities

by any earlier layer. Therefore, it must have been in the array representing the "surviving" elements of P when L_i is populated because there is no layer higher in the stack which could have rejected it and caused it to be removed from the array. This means that it must have been inserted into L_i

Further, neither insertion nor deletion can cause a false negative. First, they cannot cause existing positives to generate false negatives. Neither algorithm modifies the negative layers, so they have no effect on which elements the negative layers accept or reject. Therefore, it cannot cause an existing positive element to reach a positive layer that it did not reach prior to the insertion which is the only way that it could be falsely rejected. Second, the element which is inserted will not be a false negative during the lookup operation. If it reaches positive layer L_i during lookup, then it must not have been rejected by any earlier negative layers. This implies that it would have reached and been inserted into L_i during the insert algorithm. Lastly, an element which is deleted cannot be a false negative because it is no longer a part of the positive set.

4.3 Expected False Positive Rate

Because the stacked filter algorithm effectively produces two false positive rates, one for known negatives, and one for unknown negatives, we provide a unified expression for the expected false positive rate conditioned on the distribution of the negative queries. However, the only necessary parameter of that distribution for this calculation is the probability that an arbitrary negative query will be within the set of known negatives which we denote as ψ,

$$\psi = P(x \in N_k | x \in N)$$

Further, let $FP(x)$ be the event that the filter produces a false positive on the input of x. Given this, we can use the law of total probability to get,

$$P(FP(x)|x \in N, \psi) = \psi P(FP(x)|x \in N_k) + (1-\psi) P(FP(x)|x \in N \setminus N_k)$$

Because a known negative is entered into, and therefore passes through with probability 1, any negative layer that it reaches in the lookup algorithm, the only way that it could be accepted is by passing through the entire stack. This requires that the element is a false positive for each positive layer. Further, as the hash functions for the filters are independent, the event that the element is a false positive for any two layers is independent. Therefore, we take the product of the FPRs of each positive layer to get the expression for the false positive probability of a known negative,

$$P(FP(x)|x \in N_k) = \prod_{i=0}^{\frac{N-1}{2}} \alpha_{2i}$$

An unknown negative element, however, could be accepted as a positive by being rejected from negative layers or by making it through the entire stack. Therefore, the total false positive probability for an unknown negative element is the following,

$$P(FP(x)|x \in N \setminus N_k) = \prod_{i=0}^{T_L-1} \alpha_i + \sum_{i=1}^{\frac{T_L-1}{2}} (1-\alpha_{2i-1}) \prod_{j=0}^{2(i-1)} \alpha_j$$

Combining these two, we get the conditional false positive rate of,

$$P(FP(x)|x \in N, \psi) =$$
$$\psi \prod_{i=0}^{\frac{T_L-1}{2}} \alpha_{2i} + (1-\psi)\left(\prod_{i=0}^{T_L-1} \alpha_i + \sum_{i=1}^{\frac{T_L-1}{2}} (1-\alpha_{2i-1}) \prod_{j=0}^{2(i-1)} \alpha_j\right)$$

4.4 Traditional False Positive Rate

In addition to the conditional false positive rate, we can still use the false positive rate of the initial layer as an upper bound on the traditional false positive rate. This is because both known and unknown negative queries must register as a false positive for the first layer before reaching any further point at which they may be be marked positive. Therefore, for any arbitrary negative query x, we still have the traditional a priori upper bound on the false positive rate,

$$P(FP(x)|x \in N) \le \alpha_0$$

4.5 Tolerance to Changes in Negative Query Distribution

The tolerance of the false positive rate to changes in the query distribution can be quantified using a measure of distance between distributions called *total variation difference*.

Let P be the initial query distribution and P' be the query distribution after some change, and let ψ and ψ' be the probability that a negative query lies in N_k under each distribution. The total variation distance, $\delta(P, P')$, is equal to the maximum difference in the probability of any event under P and P'. This directly bounds the difference in ψ between two distributions because ψ and ψ' represent the probability of the same event.

$$|\psi - \psi'| \leq \delta(P, P')$$

The difference in the false positive rates under P and P' can then be bounded,

$$\underset{P,P'}{\Delta} P(FP(x)|x \in N) =$$

$$(\psi - \psi') \prod_{i=0}^{\frac{T_L-1}{2}} \alpha_{2i} - (\psi - \psi')(\prod_{i=0}^{T_L-1} \alpha_i + \sum_{i=1}^{\frac{T_L-1}{2}} (1 - \alpha_{2i-1}) \prod_{j=0}^{2(i-1)} \alpha_j)$$

$$\underset{P,P'}{\Delta} P(FP(x)|x \in N) \leq$$

$$\delta(P, P')| \prod_{i=0}^{\frac{T_L-1}{2}} \alpha_{2i} - \prod_{i=0}^{T_L-1} \alpha_i - \sum_{i=1}^{\frac{T_L-1}{2}} (1 - \alpha_{2i-1}) \prod_{j=0}^{2(i-1)} \alpha_j|$$

Therefore, the false positive rate changes linearly with respect to the total variation distance from the initial distribution. Additionally, via the Pinsker Inequality, we know that the square root of half the KL distance, a measure of statistical distance often used in machine learning and other fields, is an upper bound on the total variation distance as well which means that the change in false positive rate is sublinear in the KL distance between the initial distribution and the changed distribution.[11]

5 OPTIMIZATION OF SPACE DISTRIBUTION BETWEEN LAYERS

5.1 Formulating the Optimization Problem

The problem of how to distribute bits between the various layers of the stacked filter does not lend itself easily to analytic optimization methods. This is because the expression for the size necessary to generate an expected FPR consists of polynomials multiplied by logarithms that do not necessarily result in a closed form solution. Therefore, we choose to use modern methods of nonlinear optimization which allow for both a nonlinear objective function and constraints. The objective function $f(\alpha)$ takes in the list of per-layer false positive rates and outputs the conditional false positive rate of the stacked filter.

$$f(\alpha) = \psi \prod_{i=0}^{\frac{N-1}{2}} \alpha_{2i} + (1 - \psi)(\prod_{i=0}^{N-1} \alpha_i + \sum_{i=1}^{\frac{N-1}{2}} (1 - \alpha_{2i-1}) \prod_{j=0}^{2(i-1)} \alpha_j)$$

The constraint function,$S(\alpha)$ is the cumulative size necessary to produce each layer at the FPR specified in the input subtracted from the total space allotted to the stacked filters. As mentioned in the background section, each layer's size is found by rounding the expression for the optimal number of hash functions to the closest integer then inverting the exact expression for the false positive rate of a bloom filter given above. Therefore, if n elements are expected to be inserted into the ith layer, the size of a single layer would be,

$$S_i(\alpha_i) = \frac{1}{1 - (1 - \epsilon^{\frac{1}{k_i}})^{\frac{1}{n_i k_i}}} \quad k_i = \text{round}(\frac{-\log(\alpha_i)}{\log(2)})$$

The number of elements expected to be inserted into a given layer depends on how many "survive" to that point of the construction algorithm and whether L_i is constructed from P or N_k. If i is even, we get,

$$n_i = |P| \prod_{j=1}^{\frac{i}{2}} \alpha_{2j-1}$$

If i is odd,

$$n_i = |P| \prod_{j=0}^{\frac{i-1}{2}} \alpha_{2j}$$

This formulation presumes a set amount of memory dedicated to the filter and attempts to generate the lowest possible FPR. However, using the nonlinear optimization packages, it is simple to set a desired maximum FPR and instead optimize the space taken up by the filter.

Computation Penalty Functions. In addition to the total compression, users are also interested in reducing the computational workload induced by the filter. To manage this, it is quite simple to insert a penalty coefficient to the objective function which tracks the expected number of single-layer filter lookups required for a positive, known negative, and unknown negative element. The calculation for these values is given above in the lookup algorithm discussion, but the penalty coefficient is structured as the following where c_P,c_{N_k}, and $c_{N \setminus N_k}$ are chosen depending on the cost of slow computation for each element type.

$$C = (1 + c_P \mathbb{E}(\text{Lookups}(x_P)) + c_{N_k} \mathbb{E}(\text{Lookups}(x_{N_k})) + c_{N \setminus N_k} \mathbb{E}(\text{Lookups}(x_{N \setminus N_k})))$$

Implementation. While there exist a variety of suitable optimization algorithms and packages available, we chose to use the NLOPT package's implementation of the ISRES algorithm for the global search and the COBYLA algorithm for the local search to polish the result[14][22][19]. Because diminishing returns are achieved from additional layers, there is rarely a reason to use more than five layers, so the optimization problem is relatively tractable despite its nonlinearity. Further, as they are probabilities, each α_i is bounded between

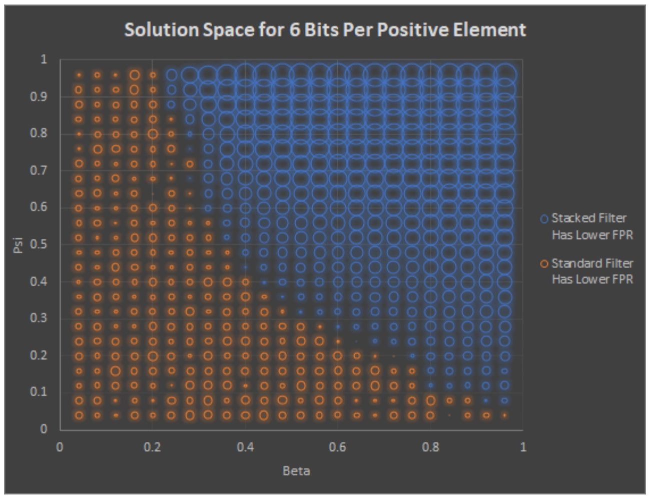

Figure 4: 6 Bits Per Element

zero and one. Lastly, to find a starting point for the optimization, we first run the optimization with the restriction that all levels have the same FPR and calculate the FPR of a standard one-layer filter of the given size. Then, depending on which has a lower overall FPR, we either start optimizing at that α, or we begin with all bits allocated to the first level by setting the FPR of every later level to 1.

6 PRELIMINARY RESULTS

Experimental Setup. The items inserted into the filter and tested against it are 32 bit integers drawn from a uniform random distribution. We did not remove duplicates because the probability of duplicates was very small.

We implemented a standard bloom filter with double hashing and hashed the values using the CityHash 64 bit hash function with each filter in the stack using a unique randomized seed value. The standard bloom filter was then used as the layers of the stacked filter and the comparison was made a traditional single layer filter and stacked filters with 3, 5, or 7 layers with a focus on FPR and number of lookups.

The filters were tested under workloads which varied in the proportion of positives to known negatives and in the value of ψ, the amount of the negative query distribution's cdf captured in N_k. The former proportion is given the label β in figures where,

$$\beta = \frac{|P|}{|P| + |N_k|}$$

Compression Comparison to the Standard Filter. Based on the experimental results, we see that the stacked filter achieves greater compression than the standard filter under a wide variety of workloads. Figures 2-4 show which filter performs better under a variety of values for *psi* and *beta* for

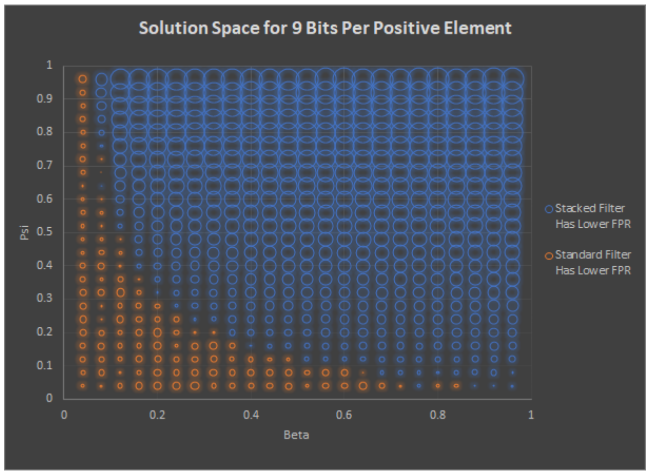

Figure 5: 9 Bits Per Element

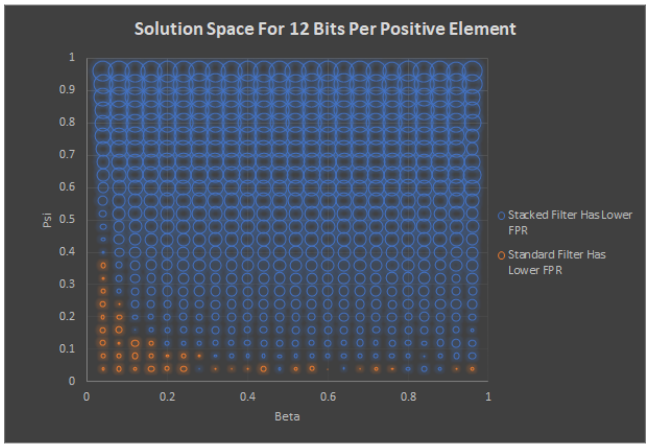

Figure 6: 12 Bits Per Element

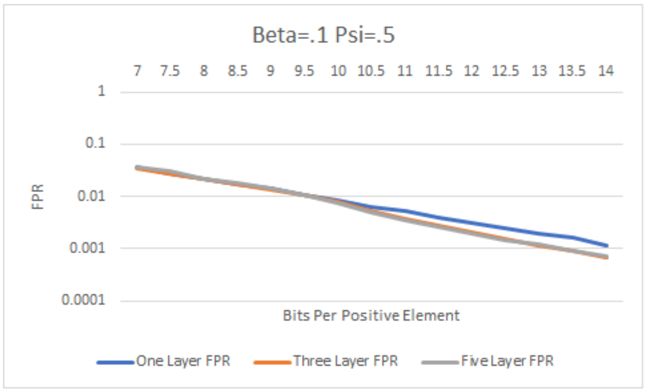

Figure 7: Beta = .1 Psi = .5

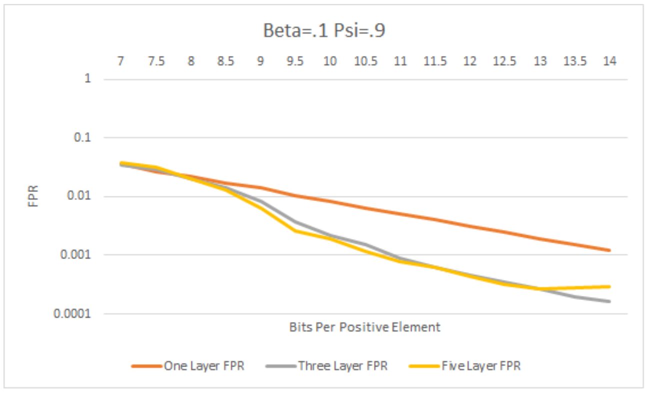

Figure 8: Beta = .1 Psi = .9

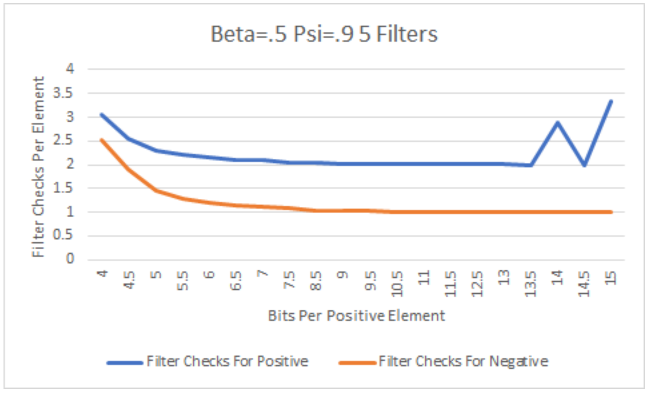

Figure 9: Computation 5 Layers Beta = .5 Psi = .9

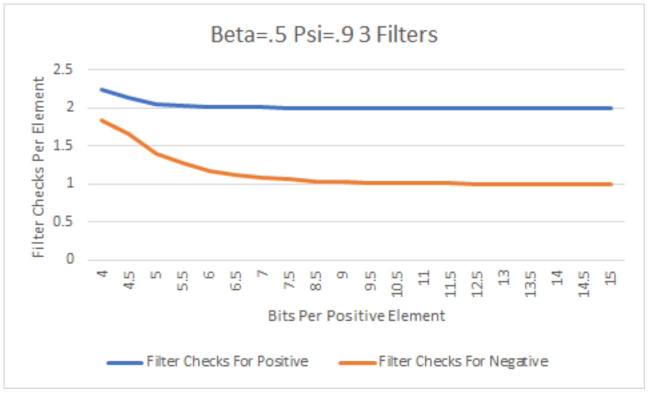

Figure 10: Computation 3 Layers Beta = .5 Psi = .9

different sizes of filter. We can see that the more bits given to the filter the more the stacked filter outperforms the standard filter. As a heuristic, if half of the negative distribution can be captured in the same number of elements as the positive set, then the stacked filter will always outperform the standard filter.

In addition, we can see how the false positive rates of the stacked and standard filters diverge when the size of the filters is increased in figures 5 and 6. The size of this divergence is based primarily on the value of *psi* which makes sense because the stacked filter is only provides a lower FPR for the elements of the set of known negatives, so the potential gains are determined by what proportion of negative queries lie in that set. When the false positive rates are approximately equal prior to their divergence, it is because the stacked filters are mimicking a single filter by allocating all bits to a single positive filter and allowing the FPR of the other filters to equal 1.

Computation Comparison to the Standard Filter. Figures 7 and 8 show the number of filter lookups required for a positive element vs a negative element for the 5 layer and 3 layer stacked filters respectively. For negative elements, the performance approaches the standard filter as bits are added because more and more negative elements are only queried against the first filter of the stack. Positive elements on the other hand approach 2 filter lookups on average because they pass through the first filter with probability 1 and then are generally thrown out by the second filter. Because the time to reject a negative element is the primary measure of speed in bloom filters, this is a very promising finding which shows that the computational performance of stacked filters and standard filters will be very similar.

6.1 Experiment Proposals

6.1.1 Basic Integer Tests. First, the basic concept is validated by hashing 64 bit integers. Further, the solution space is explored in the simple theoretical context to give intuition for its application.

Integer Tests - Experiment 1. The first three figures will demonstrate the solution space at an FPR of 1%,.01%, and .001%. It will be the same format as figures 2-4 except with FPR set constant instead of size.

Integer Tests - Experiment 2. Next, we will emulate some of the graphs from "Performance-Optimal Filtering:Bloom Overtakes Cuckoo at High Throughput". In particular, we will presume two different pairs of psi and beta then recreate the graph of workload. In order to properly do this, we will need to include implementations of the stacked filter design where the layers consist of cuckoo filters and register blocked bloom filters.

6.1.2 URL Blacklisting. **URL Blacklisting - Experiment 1.** First, we will make two graphs with the space required on the y axis psi on the x axis and plot our filter against the learned filter and a standard filter which will simply be horizontal lines. We will fix the FPR at 1% and .01% and beta at .5 .

URL Blacklisting - Experiment 2. Second, we will provide a graph which shows stacked filters with Psi at .75 and Beta at .1, .2, and .5. This graph will mirror figure 10 in "The Case for Learned Index Structures" with FPR on the x axis and filter size on the Y axis. We can then superimpose the learned bloom filter graph to compare the methods directly under different conditions.

6.2 LSM Trees

LSM Trees - Experiment 1. First, we need to examine whether this system allows for smaller LSM tree bloom filters with the same number of disk accesses. In order to show this, it is important to examine performance of this system under different patterns of data access. This can be summarized by a graph with the disk accesses on the y axis, with the space allotted to the system on the x axis, and lines showing the performance of the standard filter compared with the stacked filter design under various parametrizations of the zipfian distribution for data access patterns. Depending on the time constraints, it seems reasonable to allow the system to be aware of the data access pattern and say that modeling data access patterns is out of the scope of this paper.

LSM Trees - Experiment 2. Second, we need to understand whether this improves the speed of the LSM tree overall. For this, we will repeat the graph above but measure it in lookups per second rather than disk accesses to ensure that the potential for multiple filter checks does not notably decrease the overall performance.

7 RELATED WORK

Bloom filters were originally proposed by Burton Bloom in 1970 to solve a dictionary problem involving identifying a subset of words that require hyphenation.[1] However, in the last 20 years, they have been integrated into a variety of applications where they are used for everything from semi-joins to distributed caching.[2] Further, there has been a flurry of research proposing extensions, optimizations, and replacements for bloom filters.

In this section, we first discuss replacements for bloom filters which improve on the space efficiency of the standard filter under the usual constraints. Then we will consider novel designs which make gains in space efficiency by allowing for relaxations of the traditional problem such as allowing some number of false negatives or assuming a learnable structure in the set of elements. Finally, we will discuss Bloomier filters which expand bloom filters to allow for the encoding of multiple sets.

Bloom Filter Alternatives.

In order to increase the space efficiency of approximate membership query structures beyond that of standard bloom filters, the Cuckoo Filter has been suggested as a replacement for the standard bloom filter.[9] Assuming a false positive rate of ϵ, this structure provides a space requirement of approximately $1.05 * \log_2(1/\epsilon) + 2.1$ bits per element as compared with the standard bloom filter's $1.44 * \log_2(1/\epsilon)$ bits per element. Further, it allows for deletions which standard bloom filters do not allow. Another suggested replacement for the bloom filter is the Counting Quotient Filter which improves upon the original Quotient Filter.[18] This similarly achieves a space requirement of $1.05 * \log_2(1/\epsilon) + 2.37$ while permitting better storage locality as well as dynamic resizing and counting for "free". Because of these benefits, it is a promising direction for future research to use these new filters as layers within the stacked filter design proposed in this paper. In particular, the dynamic resizing made possible by the QCF could allow for on-the-fly re-allocation of space between the layers based upon changes in the negative query distribution.

However, there has been shown to be a lower bound on any structure attempting this problem without any relaxed constraints. The space constraints must be at least $\log_2(1/\epsilon)$ which motivates exploration of new formulations of the problem which could remain widely applicable while allowing for further gains in space efficiency.[3]

AMQs with Relaxed Constraints.

One expansion of the problem which allows for a significant reduction in FPR is to permit some small number of false negatives in order to greatly reduce the number of false positives. This is explored in depth in the paper that introduces Retouched Bloom Filters.[8] While it can significantly increase the efficiency of the filter, the two-sided error provided by Retouched Bloom Filters cannot be easily accomodated in many of the existing applications for AMQs because it does not provide any answers with certainty.

A different approach is taken when creating Learned Bloom Filters which attempt to use a machine learned binary classifier to take advantage of the natural structure of the set being represented before storing any false negatives produced by it in a backup bloom filter.[15] This method appears to allow for significant space savings when the elements of the positive set have a learnable pattern to them which differs from the negative set. However, many traditional bloom filter applications do not provide sets with neatly learnable characteristics. For instance, consider applications where bloom filters store identifiers that are purposely uncorrelated from user's information for privacy reasons. Further, both the training and use of the models remains considerably slower than state of the art AMQ designs.

Bloomier Filters.

The Bloomier Filter provides an interesting design counterpoint to the design in this paper because it expands bloom filters to represent multiple sets with one-sided error for each.[4] Each of the input elements are coded with membership in one of these sets and it is guaranteed that during lookup each input element will be mapped back to its original set. Elements not in the original set, however, will be mapped to their own ”unknown” category with probability $1 - \epsilon$ or to a random selection of the input sets with probability ϵ. The stacked filter design suggested in this paper effectively encodes two sets, positive elements and known negative elements. However, while it provides the guarantee of one-sided error for the former, it allows for two-sided error in its representation of the latter rather than allocating the bits needed to provide one-sided error. Modifying this algorithm to provide fewer guarantees about the negative set while maintaining the one-sided error for the positive set is an interesting avenue for future work.

REFERENCES

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

[2] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[3] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 59–65. ACM, 1978.

[4] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39. Society for Industrial and Applied Mathematics, 2004.

[5] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94. ACM, 2017.

[6] F. Deng and D. Rafiei. Approximately detecting duplicates for streaming data using stable bloom filters. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 25–36. ACM, 2006.

[7] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 201–212. ACM, 2003.

[8] B. Donnet, B. Baynat, and T. Friedman. Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives. In *Proceedings of the 2006 ACM CoNEXT conference*, page 13. ACM, 2006.

[9] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.

[10] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[11] A. A. Fedotov, P. Harremoës, and F. Topsoe. Refinements of pinsker's inequality. *IEEE Transactions on Information Theory*, 49(6):1491–1498, 2003.

[12] D. Geneiatakis, N. Vrakas, and C. Lambrinoudakis. Utilizing bloom filters for detecting flooding attacks against sip based services. *computers & security*, 28(7):578–591, 2009.

[13] A. Gervais, S. Capkun, G. O. Karame, and D. Gruber. On the privacy provisions of bloom filters in lightweight bitcoin clients. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 326–335. ACM, 2014.

[14] S. G. Johnson. Nlopt.

[15] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[16] H. Lang, T. Neumann, A. Kemper, and P. Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proceedings of the VLDB Endowment*, 12(5):502–515, 2019.

[17] M. Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In *Advances in Neural Information Processing Systems*, pages 464–473, 2018.

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

[19] M. J. D. Powell. A direct search optimization method that models the objective and constraint functions by linear interpolation. *Advances in Optimization and Numerical Analysis*, page 51fi?!67, 1994.

[20] D. L. Quoc, I. E. Akkus, P. Bhatotia, S. Blanas, R. Chen, C. Fetzer, and T. Strufe. Approxjoin: Approximate distributed joins. In *ACM Symposium of Cloud Computing (SoCC) 2018*, 2018.

[21] S. Ramesh, O. Papapetrou, and W. Siberski. Optimizing distributed joins with bloom filters. In *International Conference on Distributed Computing and Internet Technology*, pages 145–156. Springer, 2008.

[22] T. Runarsson and X. Yao. Search biases in constrained evolutionary optimization. *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, 35(2):233fi?!243, 2005.

[23] H. Stranneheim, M. Käller, T. Allander, B. Andersson, L. Arvestad, and J. Lundeberg. Classification of dna sequences using bloom filters. *Bioinformatics*, 26(13):1595–1600, 2010.

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