Stacked Filters

Anonymous Author(s)

ABSTRACT

We present Stacked Filters, a new methodology to design filters. We show that Stacked Filters achieve similar sizes and false positive rates to Learned Filters while remaining robust to noisy data and shifting workloads similarly to traditional filters (e.g., Bloom Filters). Stacked Filters consist of a series of traditional filters which alternate between storing positive elements, i.e. elements that exist in the data set, and frequently queried negative elements (referred to as known negatives). Only elements that are incorrectly accepted by previous layers are added to later ones, exponentially reducing the size of the layers further down the stack. Therefore, known negatives need to pass through several layers of filters to appear as a false positive.

Using URL blacklisting data, as well as using synthetic data, we show that 1) compared to traditional filters, Stacked Filters reduce the number of bits needed to achieve a given expected false positive rate and 2) compared to Learned Filters, Stacked Filters provide computational efficiency, resiliency to changing query distributions, and utility regardless of the (lack of) pattern in the data. We also demonstrate the generalized nature of the design principles behind Stacked Filters by demonstrating results using Stacked Bloom Filters, Stacked Quotient Filters, and Stacked Cuckoo Filters.

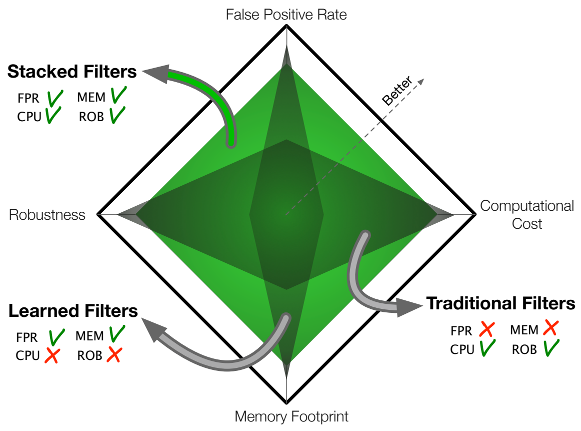

Figure 1: Stacked Fitlers improve on both Learned Filters and traditional filters by combining the best properties from both.

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. For large data sets, the raw data is usually stored over a slow medium (e.g., on disk) or distributed across the nodes of a network. Because of this, it is critical for performance to limit accesses to the full data set. That is, applications should be able to avoid accessing slow disk or remote nodes when querying for elements that are not present in the data. This is the exact utility of approximate membership query (AMQ) structures, also referred to as filters. They provide probabilistic answers to whether a queried element is in the data set with no false negatives and a limited number of false positives. In exchange for probabilistic answers, AMQ data structures have reduced data sizes as compared to the full set and are designed to be small enough to fit in memory. Thus AMQ structures help applications by allowing a quick but probabilistic check for set membership which reduces the number of expensive accesses to the full data for non-existent elements. AMQ data structures are used in a myriad of applications such as distributed joins [32, 34], web caching [16], prefix matching [13], deduping data [12], DNA classification [37], detecting flooding attacks [17], Bitcoin transaction verification [18], and LSM tree based key-value stores [10], amongst many others [38].

The Traditional Approach: Using the Positive Set. Traditional AMQ structures such as Bloom, Quotient, or Cuckoo Filters utilize only the data set during construction [5, 15, 28]. For example, a Bloom Filter starts with an array of bits set to 0 and hashes the elements of the 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 set to 1. For elements in the set, these bits were set to 1 during construction and so there are no false negatives; however, for elements not in the set, a false positive can occur if the element is hashed entirely to positions which were set to 1 during construction. If the hash functions are truly random, then every query-able value not in the data set has the same false positive chance. This makes traditional filters robust.

Learning from the Negative Set. As data sizes grow, it is critical to minimize the size of filters further so that we can represent more data while keeping filters in memory. However, because traditional filters do not take into account the characteristics of the existing application, they are confined by a theoretical lower bound on their size [9]. Learned Filters move beyond this lower bound by training a binary classifier on an expected workload, i.e. a set of positive and negative queries, and then store any false negatives in a backup bloom filter [23][27]. When a new element is queried against a Learned filter, it is first checked against the classifier and accepted outright if the classifier labels it as a positive. Otherwise, the element is checked against the backup filter and accepted or rejected based on the backup filter's decision. This guarantees that no positive elements are rejected because all positives incorrectly labeled as a negative by the classifier were added into the backup filter.

An effective classifier rejects almost all negative elements while only rejecting a small fraction of the positive elements. If so, the classifier generates very few false positives, and only a few positive elements need to be entered into the backup filter. This allows the backup filter to be smaller or more accurate than a traditional filter holding the full positive set. Using this method, the required memory to achieve a given false positive rate can be reduced by upwards of 60%.

The Open Problem. While Learned Filters offer a solution to the problem of size, they also bring a new set of open challenges which limit their utility in a wide array of traditional filter applications. First, Learned Filters rely on the existence of a learnable distinction between the positive and negative sets. This poses a problem for applications which work with encrypted data or machine generated identifiers. Second, the use of complex models adds significant computational overhead. For instance, training and querying a neural network is several orders of magnitude more expensive than creating and querying a hash based filter. Third, by definition a Learned Filter is tailored to a particular workload. However, there are many applications where the workload may shift over time, e.g., as data and query patterns evolve. For Learned Filters, the model must be retrained or otherwise they suffer a degradation in the false positive rate.

Overall, for applications with strict performance requirements, unlearnable data, or where 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 utilize workload knowledge to reduce the false positive rate.

The Solution: Stacked Filters. We introduce a new class of filters, Stacked Filters, which attain similar size and false positive rates to Learned Filters while sacrificing very little of the computational efficiency and robustness enjoyed by traditional filters. Like Learned Filters, Stacked Filters utilize knowledge of the query distribution to be more space efficient. However, rather than using an ML model, Stacked Filters use a series of traditional filters alternately representing elements of the positive and negative sets. At each layer of a Stacked Filter after the first, the local filter contains only elements which have been incorrectly accepted by the previous layer (the first layer stores the full positive set).

Because every layer of a Stacked Filter only represents elements which passed the checks of all previous layers, every layer becomes exponentially smaller as we move down the stack. Therefore, the storage footprint of the layers beyond the first is very small, and, because the first layer can now be less precise, this results in a significant size reduction. At the same time, a known negative has to be a false positive for every positive layer in order to appear as a false positive for the Stacked Filter as a whole. When this set of known negatives represents a significant portion of the negative query distribution, the FPR improves dramatically, outperforming traditional filters and matching Learned Filters. Figure 1 shows conceptually how Stacked Filters outperform all other filters, while maintaining their good properties.

Contributions. The contributions of this paper are:

- We introduce Stacked Filters, a new way to design filters as multi layer filter structures, and show that its benefits generalize to diverse traditional filter structures including Bloom, Quotient, and Cuckoo Filters.
- We show that by emphasizing or de-emphasizing the first filter in its stack, Stacked Filters can be tuned to emphasize robustness to workload shift or tuned to optimize performance on the current workload, depending on application requirements.
- We show that since the layers down the stack become exponentially smaller in size, Stacked Filters retain almost all the robustness of traditional filters under the same space budget while generating significantly better false positive rates on their target workload.
- We show that the computational cost of Stacked Filters is drastically lower than Learned Filters. We also show that compared to Learned Filters, Stacked Filters achieve superior robustness to changing workloads.
- Finally, we demonstrate that for the common filtering scenario of protecting systems from expensive data accesses to persistent storage such as SSDs or hard disks, Stacked Filters dominate both traditional filters and Learned Filters by striking the best balance of memory footprint, query time, and false positive rate.

2 NOTATION AND METRICS

We first introduce notation used throughout the paper and metrics that are critical for describing the behavior of filters. Table 1 lists the key variables and metrics.

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. We will be building filter data structures, which we will denote by F, and which we treat as a function from $U \rightarrow \{0, 1\}$. As shorthand, we will often say x is accepted by F if $F(x) = 1$, and similarly, we will say x is rejected by F if $F(x) = 0$.

As an AMQ data structure, we have $F(x) = 1 : \forall x \in P$ and we are interested in minimizing the number of false positives, which are the event $F(x) = 1, x \notin P$. The AMQ structure F is itself random; different instantiations of F produce different data structures, either because the hash functions used have randomly chosen parameters or because the machine learning model used in Learned Filters is stochastic.

Metric 1: Expected False Positive Bound (EFPB). A traditional guarantee of an AMQ data structure F is to bound $E_F[\mathbb{P}(F(x) = 1 | x \notin P)]$ for any x chosen independently of the creation of F. 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 F. The produced bound we will describe as the *expected false positive bound*[1].

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

Optimization via 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 F 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.

Robustness via Bounding False Positive Probability. Optimizing the expected false positive rate helps system throughput, however, it brings concerns about workload shift. Traditional filters can act as a safeguard against such a shift. To see this, note that $F \perp\!\!\!\perp D$ by assumption and so

$$E_{x\sim D}[E_{F|D}[\mathbb{P}(F(x) = 1 | x \notin P)]] \leq E_{x\sim D}[E_{F|D}[\epsilon]] = \epsilon$$

regardless of what D is. Thus, AMQ data structures which provide an expected false positive bound provide an upper bound on the expected false positive rate for any workload D chosen independently of the AMQ structure F.

Memory - False Positive Tradeoff. For all AMQ data structures, their expected false positive rate and expected false positive bound can be made arbitrarily close to 0 with enough memory, and there exists a tradeoff between the amount of space taken up and the false positive rate provided. Thus for purposes of comparison, we always report EFPR and EFPB with respect to a space budget.

Computational Performance. Besides their space and false positive rate, AMQ structures desire computational performance much faster than the cost to access the data it protects.

[1]This has traditionally been referred to as *the false positive rate*. We use this separate name to distinguish it from the expected false positive rate.

Notation	Definition
N	Set of all negative elements
P	Set of all positive elements
N_k	Negatives used to construct a Stacked Filter
N_u	The complement of N_k, i.e. $N \setminus N_k$
s	Size of a filter in bits/element
$c_{i,A}$	Cost of inserting an element from set A
$c_{q,A}$	Cost of querying a filter for an element in A

Metric	Definition
EFPR	Expected false positive rate of an AMQ structure given a specific query distribution
EFPB	Lower bound on the EFPR of an AMQ structure for queries chosen independently of the filter
QPS_P	Queries per second for positive elements
QPS_N	Queries per second for negative elements

Table 1: Notation used throughout the paper

In this paper, we report throughput in queries per second as our main computational metric.

Traditional Filters. For traditional filters, the expected false positive bound and the expected false positive rate are equal. Additionally, for all traditional filters, picking a false positive rate α then determines the size of the filter, and so their size in bits per element is denoted by $s(\alpha)$.

3 STACKED FILTERS

We first provide the intuition behind the design of Stacked Filters and provide terminology before formally presenting Stacked Filters and their algorithms.

3.1 Stacked Filters Intuition

A New View on Filters. The traditional view of filters is that they are built on a set S, and contain no false negatives for S. An alternative view of a filter is that it returns that an item is certainly in $\overline{S}$ (the complement of S), or that an item's set membership is unknown. In Stacked Filters, we use this new way of thinking about filter design to repeatedly prune the set of elements in $\mathcal{U}$ whose set membership is undecided.

Known And Unknown Negatives. A second component of Stacked Filters design is utilizing workload knowledge. Stacked Filters take in a set of frequently queried negative elements and give them a lower false positive rate. This set of frequently queried negatives is denoted by N_k and called the known negative set. Its complement, $N \setminus N_k$ is denoted by N_u and is called the unknown negative set.

Stacking Filters. Stacked Filters are built by stacking a sequence of filters $L_1, L_2, \ldots,$ with odd filters being constructed on subsets of P and even filters on subsets of N_k. We begin by describing an example of a 3-layer filter using Figure 2.

The first filter in the stack, L_1, is constructed using the full set P similarly to a traditional filter except with fewer bits per element in order to reserve space for the following

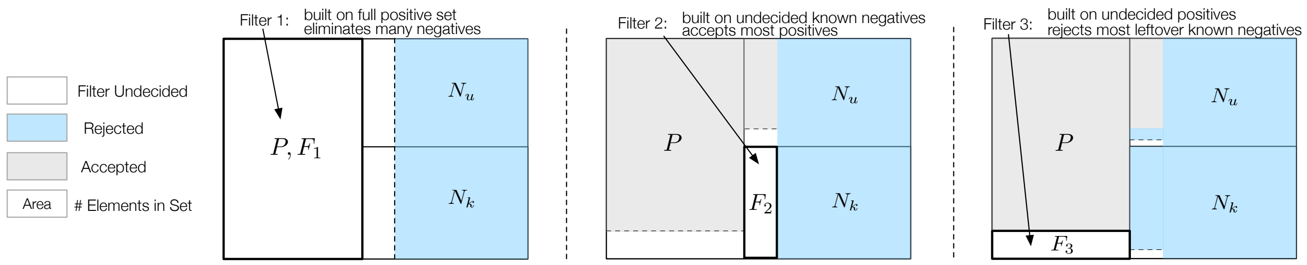

Figure 2: A conceptual diagram of the space of elements each filter uses during construction, and of the sets they accept/reject. As we descend down the stack (from left to right in the figure), filters become exponentially smaller.

layers. Conceptually, L_1 partitions the universe U. Items that L_1 rejects are known to be in N and can be rejected by the Stacked Filter. Items accepted by L_1 can have set membership of P or N and thus their status is unknown. This is depicted conceptually on the left hand side of Figure 2. If the Stacked Filter ended here after a single filter, as is the case for all traditional filters, all undecided elements would be accepted.

Instead, Stacked Filters build a second filter, L_2, using the set of known negatives N_k whose set membership is undecided after querying L_1. This can be computed since N_k is available during construction. The subset of N_k needed to create L_2 is depicted in Figure 2; noticeably, it is much smaller than the full set N_k because most of N_k is rejected by L_1. At query time, items which are still undecided after L_1 are passed to L_2. If the filter rejects the item, the item is definitely in $\overline{N_k}$, which is $P \cup N_u$. Since this set contains both positives and negatives, Stacked Filters assume that the rejected elements of L_2 are in P in order to maintain a zero false negative rate. The result is that L_2 accepts the majority of P but also creates a small amount of false positives from N_u. A conceptual view of the set of items accepted by L_2 can be seen by the grey area in Figure 2.

L_3 is built using the set of positives P whose set membership would be undecided after querying L_2. Because L_2 pruned the set of positives down to this much smaller set (depicted as the box L_3 in Figure 2), the third filter can be very small. At query time, it performs the same operations as L_1, elements rejected by L_3 are certainly in $\overline{P} = N$ and so are rejected. A conceptual depiction of how many items are rejected by L_3 is seen by comparing the right and middle boxes of Figure 2. L_3 rejects a significant portion of N_k, and additionally a small amount of N_u, driving down the number of elements whose set membership is undecided.

The queries which reach the end of the stack and still have unknown set membership need to be accepted to avoid false negatives. Thus, all white space at the end of Figure 2 would become accepts. This is similar to a traditional filter, which can be seen by comparing to the left side of Figure 2.

The Effectiveness of Stacking. Given a space budget, the traditional filter approach is to use P to construct as performant a single filter as possible, thus pushing the dotted line under Filter 1 of Figure 2 as far as possible to the left. Because the size needed to reach a given EFPR for a filter is a function of the size of its input set, moving the line further to the left incurs a storage penalty dependent on the size of the full set of positives. Additionally, the filter does not distinguish between the sets N_u and N_k and so has the same EFPR for both known and unknown negatives.

Under the same space constraint, a Stacked Filter constructs multiple filters built on exponentially decreasing set sizes. Because the set sizes decrease dramatically across layers, the first filter L_1 receives almost as many bits as a standard filter and eliminates only slightly fewer negatives. With the extra space, the Stacked Filter builds a whole sequence of extra filters. The extra filters are built on small subsets of P and N_k, as for instance can be seen by L_2 and L_3 in Figure 2, and incur storage penalties dependent on these much smaller set sizes. These extra filters then eliminate more negatives, in particular from N_k, and end up more than compensating for the small amount of space given up initially. The result is that overall a lower false positive rate can be achieved for a given space budget or the same false positive rate can be achieved using significantly less space.

3.2 General Stacked Filter Construction

Deeper Stacked Filters. We formalize the construction of Stacked Filters and extend it to an arbitrary number of filters. Going forward, we refer to the traditional filters used within a Stacked Filter as layers, and we limit the use of the term filter to the full Stacked Filter. In addition, we use T_L to refer to the number of layers. Lastly, we make the distinction between positive layers and negative layers based on whether they contain elements from the positive set or negative set.

Algorithm 1 gives the algorithm for constructing a Stacked Filter. The process begins with the full set of positive elements S_P and the full set of known negatives S_{N_k}. Construction proceeds layer by layer. At each layer, one of the sets is inserted then the other set is filtered. Filtering consists of querying the layer for each element in the set and keeping

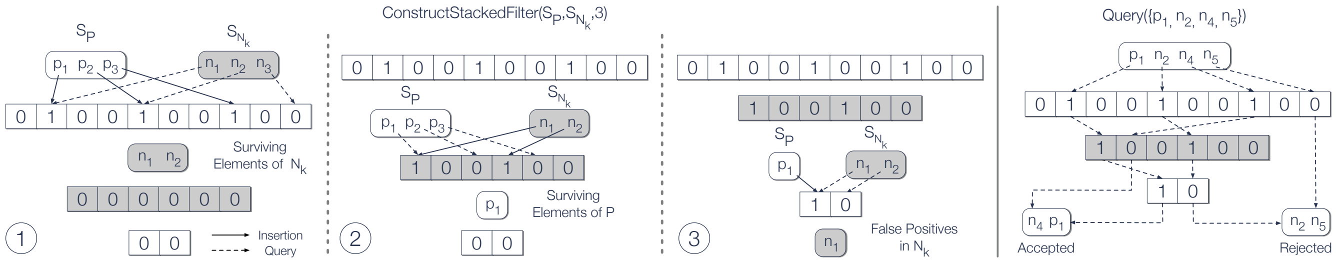

Figure 3: Stacked Filters are built in layer order, with each layer containing either positives or negatives. The set of elements to be encoded at each layer decreases as construction progresses down the stack. For queries, the layers are queried in order and the element is accepted or rejected based on whether the first layer to return not present is negative or positive.

only the elements that generate a false positive. At the start, S_P is inserted into the first layer, and S_{N_k} is filtered against it. After this, the sets alternate roles for each layer. So, the surviving elements of S_{N_k} are inserted into the second layer and every following even layer while being filtered by every odd layer. Similarly, the surviving elements of S_P are filtered by every even layer and inserted into every odd layer.

Figure 3 shows an example of a Stacked Filter built using Bloom filters where for simplicity, each Bloom Filter uses only a single hash function. The example Stacked Filter has 3 layers, and is built using 3 positive elements and 3 negative elements. The first panel shows the positives being inserted into the first layer and the negatives subsequently being filtered against it which removes n_3 from the negative set. Then, the sets switch roles with the negatives being inserted into L_2 and the positives being filtered against it. Finally, the single surviving positive element is inserted into the third filter, and we can see that n_1 is the only known negative that generates a false positive after the third layer. If there were more layers, n_1 would continue down the stack.

As the positive and negative sets are filtered, the filter sizes become smaller and smaller. For instance, Figure 3 shows that L_3 is smaller than L_2 which is smaller than L_1. In the example, this decrease in size is gradual; however, this is for illustrative purposes. In practice, false positive rates of filters are generally very low, around 1%. In this case only 1/100th of the negative elements would reach the second layer, and similarly only 1/100th of the positive elements would reach the third layer. The result is that filter sizes dramatically decrease as we descend down the stack, and that later layers add only slightly to the overall size of a Stacked Filter.

3.3 Querying a Stacked Filter

Algorithm 2 gives the algorithm for querying a Stacked Filter. Querying for an element x starts with the first layer and goes through the layers in ascending order. For every layer, if the element is accepted by the layer, it continues to the next layer. If it is rejected by a positive layer, the element is rejected by the Stacked Filter. If the element is rejected by a negative layer, the element is accepted by the Stacked Filter (note the reversal from positive layers). If the element reaches the end of the stack, i.e. it was accepted by every layer, then the Stacked Filter accepts the element.

Algorithm 1 ConstructStackedFilter(S_P, S_{N_k}, T_L)

```
Require: $S_P, S_{N_k}$: sets of positive, known negative elements
 1: // Construct the layers in the filter sequentially.
 2: for $i = 1$ to $T_L$ do
 3:     if $i \bmod 2 = 1$ then                    // layer positive
 4:         // Insert positive set, Filter negative set
 5:         $S_r = \{\}$
 6:         for $x_p \in S_P$ do
 7:             $L$.INSERT($x_p$)
 8:         for $x_n \in S_{N_k}$ do
 9:             if $L$.LOOKUP($x_n$) = 1 then
10:                 $S_r = S_r \cup \{x_n\}$
11:         $S_{N_k} = S_r$
12:     else
13:         // Insert negative set, Filter positive set
14:         $S_r = \{\}$
15:         for $x_n \in S_{N_k}$ do
16:             $L$.INSERT($x_n$)
17:         for $x_p \in S_p$ do
18:             if $L$.LOOKUP($x_p$) = 1 then
19:                 $S_r = S_r \cup \{x_p\}$
20:         $S_P = S_r$
21: return $L$
```

We now show that the above algorithm is correct, and explain why such a construction helps lower the FPR compared to a single filter. Throughout, we will refer to Figure 3, which shows an example of querying for 4 elements from the filter constructed in Figure 3. The four elements are a positive element p_1 and three negative elements n_2, n_4, and n_5, of which only n_2 is in the known negative set N_k.

Querying a Positive Element. First, we show that Stacked Filters maintain the crucial property of AMQ structures of having no false negatives. For every positive element x_p, x_p is added into each positive layer until it either hits the end of the stack, or is rejected by a negative layer and therefore accepted by the Stacked Filter. Querying the Stacked Filter for x_p follows the same path, and so has the same outcomes. Either x_p makes it through every positive and negative layer

Algorithm 2 Query(x)

```
Require: $x$: the element being queried
1: // Iterate through the layers until one rejects $x$.
2: for $i = 1$ to $T_L$ do
3:     if $L_i(x) = 0$ then
4:         if $i \bmod 2 = 1$ then
5:             return reject                // Layer positive, reject x
6:         else
7:             return accept                // Layer negative, accept x
8: return accept                            // No layer rejected, accept $x$
```

to reach the end of the stack, and is accepted, or it it makes it through every positive and negative layer until a negative layer rejects it, in which case it is accepted. In our example Figure 3, the positive element p_1 is present in L_1, and is a false positive for the known negative set in L_2. Because p_1 is a false positive for L_2, it was inserted into L_3 during construction. Therefore, L_3 cannot reject p_1, so it will make it to the end of the stack and be accepted.

Querying a Negative. For an unknown negative element $x_n \in N \setminus N_k$, it is in neither P nor N_k and so has high likelihood of being rejected by both positive and negative layers. As a result, the majority of unknown negatives are rejected at L_1, and most false positives occur from elements rejected at L_2. For the frequently queried elements of N_k, Stacked Filters do better as negative layers never reject known negatives. Thus, to appear as a false positive for the Stacked Filter, known negatives need to appear as false positives for each positive layer. This drives the false positive rate of known negative elements towards 0 quite quickly, as their false positive rate decreases exponentially in the number of layers. Figure 3 shows how this process works for unknown negatives n_4 and n_5 as well as known negative n_2.

3.4 Item Insertion and Deletion

Insertion of an element follows the same path as an element during the original construction of the filter. The positive element alternates between inserting itself into every positive filter, and checking itself against every negative filter, stopping at the first negative filter which rejects the element. Pseudocode is given in Algorithm 3.

If the underlying filter structures support deletions, as for example Cuckoo filters do, then a stack built using this filter supports the deletion of positive elements as well. The deletion algorithm follows the same pattern as the insertion algorithm, except it deletes instead of inserts the element at every positive layer. Pseudocode is given in Algorithm 4.

Insertion and Deletion of Known Negatives. Stacked filters do not support the insertion of new known negative elements, as this changes which positives are rejected at each layer. For instance, if a positive element during insertion is rejected at layer 2, it does not add itself to layers 3+. Inserting a new known negative element might change this, so that the element would now be a positive at layer 2; then

Algorithm 3 Insert(x_P)

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

Algorithm 4 Delete(x_P)

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

querying for the positive would turn up a false negative. Deleting known negative elements is possible, but doesn't provide much value (it does not lower the FPR or size of the filter).

3.5 Number and Size of Layers

There are three critical tuning parameters for Stacked Filters: T_L, $|N_k|$ and the size of each layer. To tune these parameters with respect to performance goals, we set a combination of desired minimization metrics and constraints where the metrics are size, EFPR, EFPB (worst case performance), and computation. We then use off-the-shelf optimization packages to determine values for each parameter. The equations which govern how parameter values relate to each metric are derived in Section 4; each equation is differentiable in terms of the given parameters but the equations are not convex. We describe two methods to search the parameter space depending on the underlying filter.

Number of Layers. In Stacked Filters, the false positive probability of known negatives is exponential in the number of levels (shown in Section 4), and the unknown negatives see no benefit from increasing stack depth. As a result, Stacked Filters see decreasing gains from increasing stack depth as the proportion of known negatives eliminated approaches 1. To capitalize on this, both optimization methods below start at a filter of 1 level and iteratively optimize Stacked Filters of increasing depth (i.e. 1,3,5, etc.). For each number of layers, the optimization algorithm returns how to best distribute a budget of bits across the layers. When improvement from increasing stack depth falls below some specified threshold, we stop increasing the number of layers. We note that because a single layer is an option, Stacked Filters encompass traditional filters and the optimization algorithm always returns a filter which is at least as good as a traditional filter.

Number of Known Negatives. The number of known negatives used in construction has an impact on performance, so we include it as an optimization parameter. We let $|N_k|$ vary, then calculate the proportion of queries aimed at N_k. The elements are always chosen in descending order of query frequency, and the proportion of the query distribution captured is the probability mass of the most popular $|N_k|$ elements.

Continuous Approximation Optimization. Traditional filters differ in tuning options that affect layer size. For some, such as Bloom Filters, their size can be any integer number of bits m, regardless of the number of elements in the set they encode. For these filters, we perform optimization with m a continuous variable and then round m to the nearest integer post optimization. For this solution, we need to work around the non-convexity of our metric equations. To do so, we start by assuming that each layer has equal FPR (these then determine layer sizes), which reduces the complexity of each equation and reduces the number of local minima. We then solve this parameterization for the minimum value of the given objective function under the specified constraints using the ISRES algorithm [21, 35]. Afterwards, we perform local search using the COBYLA algorithm without the constraint that each layer have equal FPR [30].

Integrality Based Optimization. For other filters, such as Cuckoo and Quotient Filters, each element is hashed to a fingerprint value; and these fingerprints are given an integer number of bits which determines the false positive rate and size of the filter. Thus, there is only a discrete set of possible false positive rates for each layer, as opposed to the much larger set of options available to filters such as Bloom Filters; this reduces their flexibility in terms of the false positive rates they can achieve, but this smaller search space is much easier to optimize over. Additionally, fingerprint bit sizes of above 20 bits create false positive rates less than 10^{-6}, and so there is no practical need to search beyond this value. Thus, for these filters, we can run the search by using all discrete combinations of Stacked Filters where each layer is a traditional filter using between 1 and 20 fingerprint bits.

More details about both optimization methods, e.g. inputs and governing equations, are in the appendix.

4 ANALYSIS OF STACKED FILTERS

We now present a detailed analysis of Stacked Filters to show their strong properties with respect to all four critical metrics: FPR, size, computation time, and robustness.

Notation. To describe traditional filters in Section 2, we fixed a false positive rate α and noted that the size is a function of α. Additionally, since the false positive chance of every element is the same, α is the expected false positive rate under any workload, and is the false positive bound.

For Stacked Filters, EFPR and EFPB are no longer equal, and their size, EFPR, EFPB, and expected computation are all functions of the sequence of false positive rates given at every layer. Thus, for layers $L_1, \ldots, L_{T_L}$ which have corresponding false positive rates $\alpha_1, \ldots, \alpha_{T_L}$, we let $\vec{\alpha} = (\alpha_1, \ldots, \alpha_{T_L})$ and write Stacked Filters metrics as a function of $\vec{\alpha}$. To distinguish them from the metrics for the base filter, metrics for Stacked Filters are denoted with a prime at the end, so for instance, the size and EFPR of a Stacked Filter are $s'(\vec{\alpha})$ and $EFPR'(\vec{\alpha})$.

For ease of presentation, we use a dummy FPR of $\alpha_0 = 1$. Additionally, as a special case we will often consider that all α_i values have the same value α.

4.1 Improvements in FPR on N_k

First, we show that for the set of known negatives, Stacked Filters improve upon the false positive rate of traditional filters by an exponential amount. As a result, when the known negatives are heavily queried Stacked Filters provide substantial improvement over traditional filters.

For Stacked Filters, all known negatives in N_k have one expected false positive rate, and all unknown negatives in $N_u = N \setminus N_k$ have a second expected false positive rate. To calculate the total EFPR for a Stacked Filter, we will need a new variable ψ which captures the probability that a negative query from distribution D is in N_k, i.e. $\psi = \mathbb{P}(x \in N_k | x \in N)$.

Known Negatives. For $x \in N_k$, a Stacked Filter returns 1 if and only if every filter returns 1. The probability of this happening is

$$\mathbb{P}(F(x) = 1 | x \in N_k) = \prod_{i=0}^{(T_L-1)/2} \alpha_{2i+1}$$

In the case that each α value is the same, then this is $\alpha^{(T_L+1)/2}$ and the EFPR of known negatives decreases exponentially in the number of layers.

Unknown Negatives. For $x \in N_u$, its total false positive probability is the sum of the probability that it is rejected by each negative layer, plus the probability it makes it through the entire stack. For negative layer $2i$, the probability of rejecting this element is $\prod_{j=1}^{2i-1} \alpha_j * (1 - \alpha_{2i})$, where the first factor is the probability of making it to layer $2i$ and the second factor is the probability that this layer rejects x. Summing up these terms and adding in the probability of making it through the full stack, we have

$$\mathbb{P}(F(x)=1|x\in N_u)=\prod_{i=1}^{T_L}\alpha_i+\sum_{i=1}^{(T_L-1)/2}\prod_{j=1}^{2i-1}\alpha_j(1-\alpha_{2i})$$

In the common event that alpha values are small, this is well approximated by

$$\mathbb{P}(F(x)=1|x\in N_u)\approx\alpha_1(1-\alpha_2)$$

For instance, if $\alpha=0.01$ for all layers, the maximum error this approximation would attain for any value of T_L is 10^{-6}.

Expected False Positive Rate. Since N_u and N_k partition N, the EFPR of a Stacked Filter is

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

A convenient approximation to the false positive rate in the case that all α values are equal and the value is small is

$$EFPR=\psi\alpha^{\frac{T_L+1}{2}}+(1-\psi)\alpha(1-\alpha)$$

Thus, the false positive rate for known negatives is exponential in the number of layers, whereas the unknown negatives have EFPR close to the FPR of the first filter.

4.2 Smaller Filter Sizes

We now analyze the space of Stacked Filters and show how they can be much smaller than traditional filters for the same FPR or achieve substantially better FPR for the same size.

Size of a Stacked Filter given the FPR at each layer. For every positive layer, an element from P is added to the layer if it appears as a false positive in every previous negative layer. Thus, the size of all positive layers is

$$\sum_{i=0}^{\frac{T_L-1}{2}}s(\alpha_{2i+1})*|P|*(\prod_{j=0}^{i}\alpha_{2i})$$

Similarly, negatives appear in a layer if they are false positives for every prior positive layer and so the size of all negative layers is

$$\sum_{i=1}^{\frac{T_L-1}{2}}s(\alpha_{2i})*|N_k|*(\prod_{j=1}^{i}\alpha_{2i-1})$$

The total space for a Stacked Filter is then the sum of these two equations. Because α_i values are small, the products in parenthesis go to 0 quite quickly in both equations and so the total size of a Stacked Filter is comparable to the size of the first filter in the stack.

Size when each layer has equal FPR. In the case that all α values are the same, we can use a geometric series bound on both arguments above, giving

$$s'(\vec{\alpha})\le s(\alpha)*(\frac{1}{1-\alpha}+\frac{|N_k|}{|P|}\frac{\alpha}{1-\alpha}) \quad (2)$$

where $s'(\alpha)$ represents the size in bits per (positive) element.

Stacked Filters are smaller than traditional filters. When all α values are equal and α is small, we see that the term inside the parenthesis in (2) is very close to 1. As a result, the Stacked Filter produces almost no space overhead compared to a traditional filter with FPR α. For instance if $|N_k|=|P|$ and $\alpha=0.01$ so that every filter has a 1% false positive rate, then a Stacked Filter of infinitely many filters has only a 2% space overhead compared to a traditional filter. At the same time, if the workload contains any set of frequently queried elements, then the EFPR of the Stacked Filter is substantially lower than the traditional filter, as seen when comparing $\psi\alpha^{\frac{T_L-1}{2}}+(1-\psi)\alpha(1-\alpha)$ to α. As a result, if we increase the FPR at each layer so that the EFPR of the traditional and Stacked Filter are equal, the Stacked Filter is much smaller.

4.3 Comparable Computational Costs

We now analyze computational costs and show that Stacked Filters impose only a small overhead in filter probe and construction time. In the common case that base data accesses are the bottleneck, such as disk reads or network accesses, then Stacked Filters bring a massive benefit in performance by dramatically reducing the number of expensive data accesses needed. For all equations, the computational cost represents the average computational cost.

Stacked Filter Construction Costs. For both positives and known negatives, the construction algorithm is best analyzed in pairs, wherein the same number of elements are inserted at one layer and then queried against the next. For positives, every element is inserted into L_1 and then checked against L_2. The false positives from L_2 are then inserted into L_3 and checked against L_4, and so on. The total cost, in terms of base filter operations, is $|P|(c_i+c_q)+|P|\alpha_2(c_i+c_q)+|P|\alpha_2\alpha_4(c_i+c_q)+\ldots$. In more concise notation, this cost is

$$|P|(c_i+c_q)\big(\sum_{i=0}^{\frac{T_L-1}{2}-1}\prod_{j=0}^{i}\alpha_{2i}\big)+|P|c_i\prod_{j=0}^{\frac{T_L-1}{2}}\alpha_{2i}$$

where the final term comes from the last layer insertions.

For negative layers, the analysis is similar, but the paired layers are instead 2 and 3, 4 and 5, and so on, with known negatives having the first layer unpaired instead of the last. The number of operations to insert negatives is then $|N_k|c_q+|N_k|(c_i+c_q)\alpha_1+|N_k|(c_i+c_q)\alpha_1\alpha_3+\ldots$, or more concisely,

$$|N_k|c_q+|N_k|(c_i+c_q)\sum_{i=0}^{\frac{T_L-1}{2}}\prod_{j=1}^{i}\alpha_{2j-1}$$

The total construction cost is the sum of the operations for positive and known negative elements.

In the case that the α values are all equal, the total cost can be bounded using geometric series by

$$|P|(c_i + c_q)\frac{1}{1-\alpha} + |N|c_q + |N|(c_i + c_q)\frac{\alpha}{1-\alpha}$$

In this case, comparing against the cost for a single base filter of $c_i * |P|$, the major overheads can be seen as the cost of querying a base filter for both every positive and known negative element once, plus a small overhead from stacking.

Query Costs. The analysis of querying a Stacked Filter for a positive or known negative is almost exactly the same as the analysis for constructing a stacked filter, except that inserts are replaced with queries. For instance, when querying a positive, it queries both the first two layers with certainty, the next two layers with probability α_2 (in the case it was a false positive for L_2), the two after that with probability $\alpha_2\alpha_4$ and so on, giving us the equation:

$$c'_{q,P} = 2c_q\Big(\sum_{i=0}^{\frac{T_L-1}{2}-1} \prod_{j=0}^{i} \alpha_{2i}\Big) + c_q \prod_{j=0}^{\frac{T_L-1}{2}} \alpha_{2i}$$

with the cost in terms of how many base filter queries are needed to query for a single positive element.

For a known negative element, the cost of querying is similarly derived taking the analysis from construction costs and replacing inserts with queries. Thus, we have

$$c'_{q,N_k} = c_q + 2c_q \sum_{i=0}^{\frac{T_L-1}{2}} \prod_{j=1}^{i} \alpha_{2j-1}$$

For an unknown negative element, it can be rejected by both positive and negative layers. Thus, the probability of it reaching layer i is the product of the false positive rates for layers $1, \ldots, i-1$ and we have

$$c'_{q,N_u} = \sum_{i=0}^{T_L} \prod_{j=1}^{i} \alpha_i$$

As in all prior equations, the rate of convergence of each product term drives it quickly towards 0. For instance, if all layers have the same false positive rate, we can use geometric series on each term to produce:

$$c'_{q,P} \le \frac{2}{1-\alpha}c_q, \qquad c_{q,N_k} \le \frac{1+\alpha}{1-\alpha}c_q, \qquad c'_{q,N_u} \le \frac{1}{1-\alpha}c_q$$

4.4 The Robustness of Stacked Filters

The first layer provides robustness. For a Stacked Filter, any element in N is either in N_u or N_k, and so its probability of being a false positive is either $\mathbb{P}(S(x) = 1|x \in N_u)$ or $\mathbb{P}(S(x) = 1|x \in N_k)$. Since elements of N_u have higher chances of being a false positive, the EFPB of a Stacked Filter is $\mathbb{P}(S(x) = 1|x \in N_u)$. To approximate the EFPB, approximations for $\mathbb{P}(S(x) = 1|x \in N_u)$ can be used, such as $\alpha_1(1-\alpha_2)$ or just α_1. In both equations, the FPR of the first layer dominates, and it is the most important factor to the overall stack's EFPB. Since Stacked Filters with equal FPR at each layer have most of their allocated size in L_1, this means Stacked Filters have comparable worst case behavior to a traditional filter, even though Stacked Filters are tuned to a specific workload and traditional filters are not.

Performance change under workload shift. By looking at the EFPR, we can see how changes in distribution affect Stacked Filters. For an initial query distribution D with corresponding ψ, which changes to D' and corresponding ψ', the change in EFPR from D to D' depends only the change in ψ to ψ'. In particular, the change in EFPR is

$$(\psi' - \psi) * \big(\mathbb{P}(F(x) = 1|x \in N_k) - \mathbb{P}(F(x) = 1|x \in N_u)\big)$$

Thus, the performance drop in terms of expected false positive rate for a Stacked Filter changes linearly with the change in the proportion of queries aimed at known negatives.

Tuning for Expected Performance vs. Robustness. For Stacked Filters, there is a trade-off between best performance on the current workload and robustness to changes in workloads. Putting more bits on lower layers lowers the EFPR on the current workload, whereas putting more bits on the first layer increases robustness. The optimization algorithm shown in Section 3.5 can be adapted to assign sizes to each layer using a weighted combination of the expected performance and robustness or a constraint on robustness. We show this variation in detail in our appendix 9.

5 EXPERIMENTAL ANALYSIS

We now experimentally demonstrate that Stacked Filters offer dramatically better false positive rates compared to Bloom Filters, Cuckoo Filters, and Counting Quotient Filters for the same size, or, alternatively, they offer the same false positive rate at a drastically smaller size. We also show that Stacked Filters offer significantly better computational performance and robustness when compared to Learned Filters while offering a similar false positive rate and size.

Datasets. We evaluate filter techniques across two datasets. In the first, we compare Learned, Stacked, and traditional filters on URL blacklisting, the same application as [23] used when introducing Learned Filters. On the second dataset, we compare Stacked and traditional filters using synthetically generated integer data. By using synthetic data, we finely control key parameters such as universe size, ratio of positive to negative elements, and the amount of the queries aimed at the most frequently queried elements. For the synthetic data, we leave out Learned Filters as it is hard to simulate the learnability of real data with synthetic data; perfectly random integer data is too harsh whereas data generated according to a simple pattern is too easy to learn.

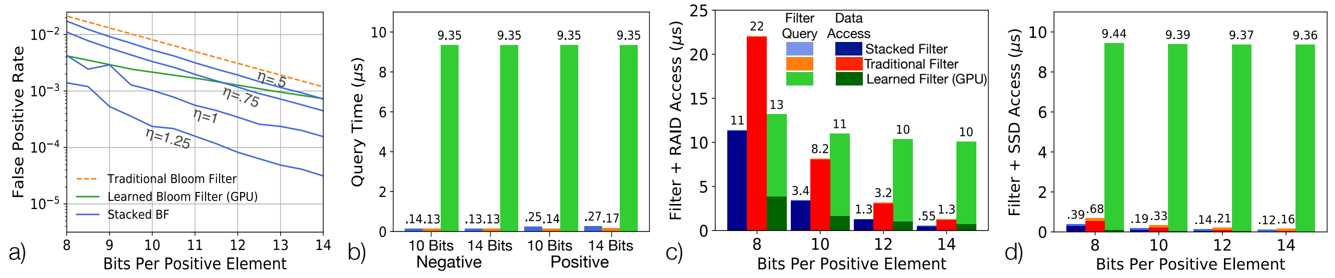

Figure 4: Stacked Bloom Filters achieve a similar EFPR to Learned Bloom Filters while maintaining a 70x throughput advantage and beating both traditional and learned filters in overall performance on typical workloads.

Workload Distribution. The performance of Stacked Filters depends on ψ, the probability that an arbitrary negative query is in the known negative set, and on the ratio of positives to known negatives. We control both variables using a Zipf distribution and adjust the Zipf parameter, η, to show the performance under different levels of skew. As η increases, the skew of the query distribution increases, and when η is 0, the distribution is uniform. The range of η values used in our experiments matches with values of η found by empirical studies, wherein studies of web resource requests ($\eta \in [.5, .9]$), usage of English words ($\eta \approx 1$), and computer passwords ($\eta \in [.5, .95]$) were all well modeled with Zipf distribution models [3, 39, 40]. The other implicit parameter in the Zipf distribution is the number of elements being modeled, $|N|$. As $|N|$ increases, the frequency of the most queried elements decreases very slowly.

Filter Implementations. The hash function used by each filters is CityHash [29]. The Bloom Filter implementation utilizes double hashing in order to speed up computation [22]. The Counting Quotient Filter (CQF) implementation is the one provided by the authors of the original paper [28]. However in the original paper, they constrain the size of the filter to a power of two in order to allow for special operations such as resizing and merging filters. This is not relevant to our testing, so we removed this restriction in our implementation to allow for greater flexibility and better memory efficiency. The Cuckoo Filter (CF) implementation is again the one provided by the authors of the original paper [15]. Unlike CQFs, CFs require that the number of hash signatures stored be a power of two in order to perform the basic operations of insert and query. Further, the implementation provided by the authors in Fan et al. only supports signature lengths of 2, 4, 8, 12, and 16, so we implemented a fix which allowed for all integer signature lengths [15]. For the Learned Filter, we use a 16 dimensional character-level GRU similar to the one used in [23] as the basis of the filter.

Experimental Infrastructure. All experiments are run on a machine with an Intel Core-i7 i7-9750H (2.60GHz with 12 cores), 32 GB of RAM, and a Nvidia GeForce GTX 1660 Ti graphics card. To reduce the effects of noise, each experimental number reported is the average of 25 runs. For the computational performance of Learned Filters, in addition to our own results, we also use the higher empirical query throughput achieved in [33] with a more powerful GPU.

5.1 URL Blacklisting

We start by showing that Stacked Filters and Learned Filters outperform traditional filters in terms of false positive rate on the URL blacklisting dataset. After, we compare computational costs, showing that hash based filters are orders of magnitude faster than Learned Filters. Translating the two numbers into total system throughput, we show that the overall throughput using Stacked Filters is better than either traditional or Learned Filters.

Dataset: Blacklisted URLs. For URL blacklisting, because the dataset in [23] is not publicly available, we use an open-source database of blacklisted URLs, Shalla's Blacklists, originally used by Singhal and Weiss [1, 36]. This dataset contains approximately 2,800,000 URLs. To create a semantically meaningful positive set, we used all URLs blacklisted for inappropriate content as the positive set and URLs blacklisted for any other reason as the negative set. We use 140 thousand positive URLs and the full set of 1.4 million negative URLs in order to mimic the disparity between the number of blacklisted and valid URLs on the internet. For Learned Filters, we train them using the full positive set and the 280, 000 most frequently queried negatives (under the target workload). Similarly, for Stacked Filters we cap the number of known negatives to 280, 000. For negative queries, the queries follow a zipf distribution with the zipf parameter varying.

Workload Knowledge Improves Filters. Figure 4a shows that across all filter sizes, both the Learned Filter and the Stacked Filter achieve lower false positive rates than the Bloom Filter, with improvements ranging from 1.24×-40×. Thus, by utilizing workload knowledge both techniques produce substantial benefits in the compression of filters. For smaller filters and lower workload skews, the Learned Filter has a better false positive rate than the Stacked Filter by up to 4.5×; however, for bigger filters and higher workload skews, the Stacked Filter is dramatically better and provides up to a 23× improvement over the Learned Filter.

For Stacked Filters, the higher workload skew means it can find a small set of negative elements that contains a large portion of the target query workload. As the set is small, the negative filters can be small in size while pruning a large portion of the positive set for positive layers lower in the stack, making these positive layers also small in size. Since the set of negative elements is also heavily queried, and Stacked Filters reduce the expected false positive rate for the known negatives to near 0, the overall false positive rate for the workload is quite low. Alternatively, Stacked Filters can achieve the same false positive rate as traditional filters at a significantly reduced size.

For Learned Filters, they attempt to learn a generalizable representation of what it means to be a negative or positive element for this workload, or in this case, what textual features make up the domain names for illicit websites. This type of information holds regardless of workload skew, and so Learned Filters do well for no to low skew workloads. However, Learned Filters have a tough time moving to lower false positive rates.

To explain this, it is necessary to understand Learned Filters in more depth. First, Learned Filters train a binary classifier f using log-loss, with f a function producing outputs in $[0, 1]$ which tries to output a number close to 1 for x positive and close to 0 for x negative. After, to control the false positive rate, a threshold τ is set such that every example x which has $f(x) > \tau$ is considered a positive (whether x is positive or not) and $f(x) < \tau$ could be a negative or a positive (no false negatives are allowed) and so checks against the backup filter. Like Stacked Filters, the size of the backup filter can be quite small if most positive examples have a value of f over τ. However, to achieve lower and lower false positive rates, it becomes necessary to make τ higher and higher so that the learned model doesn't create false positives. As a result, more and more positive examples have $f < \tau$ and need to be inserted into the backup filter. Thus, as a result, the Learned Filter approaches a traditional filter in size as the desired false positive rate gets lower.

Hash-Based Filters dominate Learned Filters computationally. Figure 4b depicts the computational performance of traditional, Stacked, and Learned Filters. For queries on negative elements, the Bloom Filter and Stacked Filter have around the same computational performance, and for queries on positive elements, the Stacked Filter is about 1.5× the cost of the Bloom Filter, even though it has about 2× the number of filter probes. This is because the second layer has good cache locality as it is quite small, thus it tends to be resident in the higher levels of the cache and returns faster than probes to the much larger first level (or a probe to an even larger single filter). Learned Filters achieve a substantially lower performance due to the massive number of computations needed, with the difference being between $45 - 80\times$ using a top of the line GPU (reported computational numbers taken from [33]). On our own GPU, this number grows to 450−800×; to be more than fair, we use the higher throughput numbers from their paper throughout.

Stacked Filters Maximize Overall Performance. Filters are used to protect systems from unnecessary data accesses, so the overall performance of a filter can be broken down into two pieces, 1) the overhead incurred by the filter checks and 2) the cost of the data accesses incurred by false positives. While Figure 4b gives the overhead incurred by filter checks for each technique, the cost of data accesses incurred by false positives depends both on Figure 4a and the speed at which base data accesses occur. Figures 4c and 4d show the results when using two different kinds of hardware to store the base data while the filters reside in memory: 1) a RAID array of hard disks capable of 1000 IOPS, and 2) a modern SSD capable of 40000 IOPS.

In both setups, Stacked Filters provide the best overall performance (total latency), striking the right balance between decreased false positive rates and affordable computational speeds for filter probes. Compared to traditional filters, both have fast hash-based computational performance but Stacked Filters have significantly fewer false positives; as a result, they provide total workload costs which are frequently less than half that of the traditional filter. In comparison to Learned Filters, Stacked Filters often have both better false positive rates and better computational performance; however, regardless of false positive rate, Learned Filters tend to have computational costs which make them prohibitive to use when compared to either traditional or Stacked Filters. No caching is used in these experiments but caching benefits all approaches in exactly the same way.

Naturally, the overall behavior depends fully on the device where the base data resides. As memory devices get faster, the utility of Learned Filters drops as computational costs become more critical (e.g., as shown when comparing Figure 4c to 4d). Future research on hardware accelerated neural networks can help in this direction. In addition, Learned Filters outperform both Stacked and traditional filters when only a tight storage budget is available (e.g., see left side of Figure 4a) which makes them a great candidate for edge devices with little memory (and perhaps hardware accelerators).

5.2 Stacking Improves Diverse Filter Types

In the next set of experiments, we showcase the generalized nature of Stacked Filters by demonstrating that the positive properties we saw in the previous section with Stacked Bloom Filters also hold for Stacked Quotient Filters and

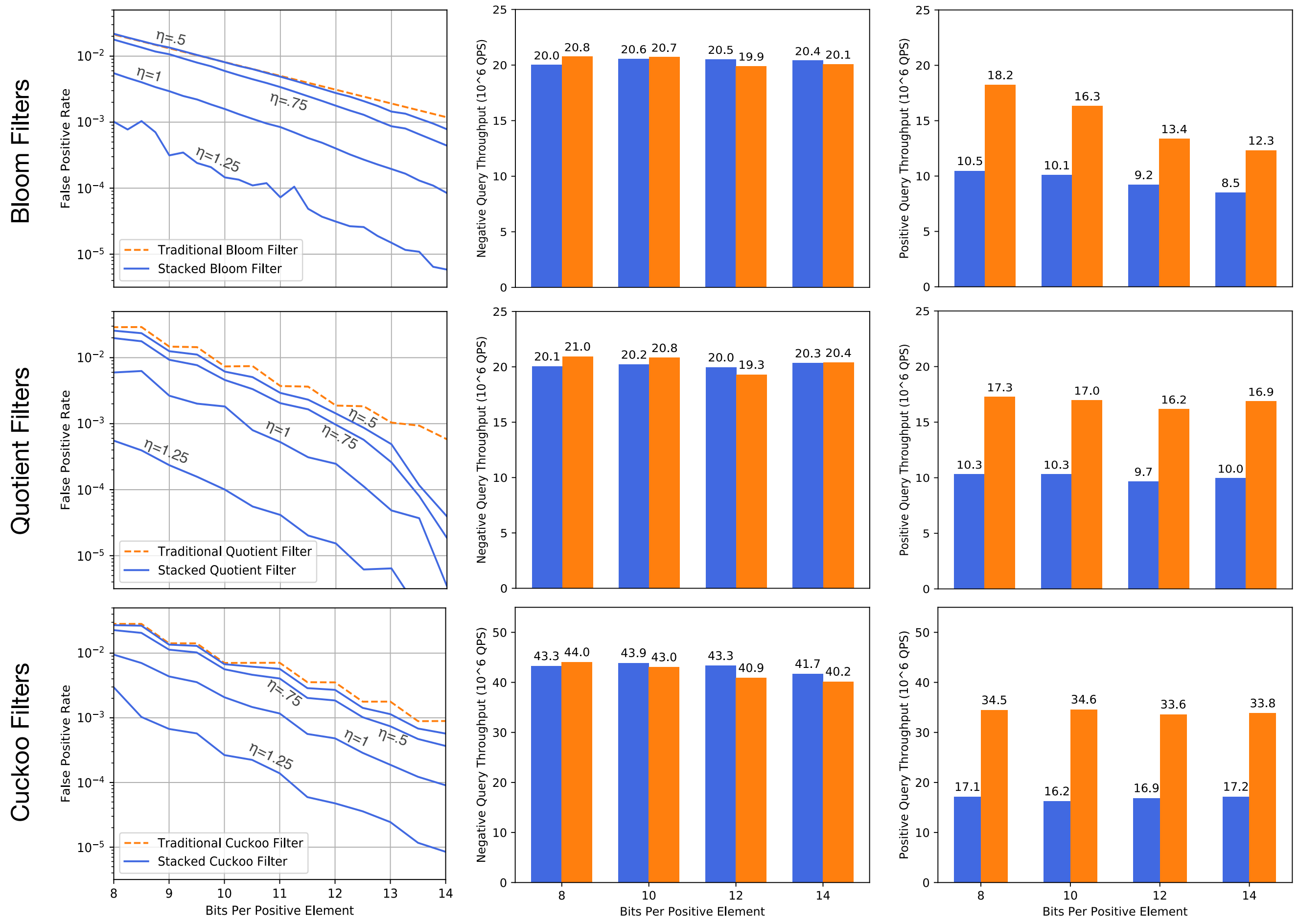

Figure 5: Stacking provides benefits across a variety of underlying filter types. Stacked Filters achieve a comparable throughput to traditional filters while achieving a drastically reduced EFPR.

Stacked Cuckoo filters. The Zipf parameter is fixed at .75 for each of the computational experiments experiments.

Dataset: Synthetic Integer Data. For this set of experiments we use random 32-bit integers to generate one million positive elements and one hundred million negative elements. As these methods are hash-based, the initial values of elements do not affect their false positive probability, and the choice of random data has no effect on performance. For this dataset, Stacked Filters choose an optimal number of most popular negative elements as described in Section 3.5.

Stacked Bloom Filters. Like we have seen for the URL data, Figure 5a shows that Stacked Bloom Filters achieve a lower EFPR than traditional Bloom Filters with a more dramatic difference as the bits per element increases and/or the data becomes more skewed. Computational performance remains competitive and so Stacked Filters can materialize big benefits in protecting from expensive data accesses. The only notable difference in computation between the Stacked Bloom Filters and other Stacked Filters is the reduced throughput in bloom filters as the bits per element increases. This is because the number of hash functions used, and corresponding memory accesses incurred, increases linearly with the bits per element.

Stacked Counting Quotient Filters. As mentioned in Section 3.5, Quotient Filters including the used variant Counting Quotient Filters (CQF), use integer length fingerprint signatures. As a result, CQFs have a small discrete set of tuning parameters which they can use. In contrast, Stacked CQFs have more parameters to tune, as they can tune each layer's hash signature length and the number of known negatives, and so Stacked CQFs are able to take advantage of nearly all of the bits allocated to the filter. This effect can be seen in Figure 5 as the Stacked CQF smoothly reduces the EFPR when given more space while the traditional CQF sees EFPR reductions in more discrete steps. The Stacked CQF achieves a decrease in the FPR at all sizes and significant space savings when the data is moderately to highly skewed.

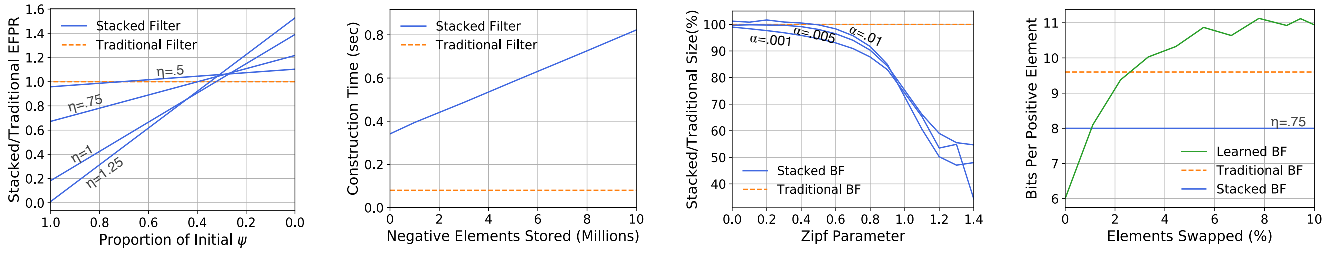

Figure 6: Stacked Filters are robust to workload shifts, can be rebuilt quickly, work across a large range of workload skews, and are immune to noisy data.

Stacked Cuckoo Filters. Figure 5 shows that Stacked Cuckoo Filters have a consistently lower EFPR than traditional Cuckoo Filters across a range of filter sizes. Similarly to CQFs, Cuckoo Filters are also tuned by choosing an integral fingerprint length, so the same smoothing effect is apparent for Stacked Cuckoo Filters as compared to traditional Cuckoo Filters.

Summary. Stacked Filters are never worse than traditional filters because Stacked Filters are a generalization of traditional filters. When the lower layers do not provide utility, the Stacked Filter adopts a single-layer design and becomes a traditional filter. As a result, across all workloads and all traditional filter types in Figure 5, Stacked Filters produce better false positive rates. For highly skewed workloads and large filters, this difference can reach two orders of magnitude. Further, while not shown here, the computational costs of Stacked Cuckoo and Quotient Filters follow the same patterns as seen in Figures 4b-d, where they materialize benefits in terms of total system throughput (these graphs can be seen in the technical report [2]).

5.3 Robustness

We now demonstrate how Stacked Filters retain the robustness of traditional filters which brings an additional benefit over Learned Filters. For these experiments we use Stacked Bloom Filters. We focus on two facets of robustness: maintaining performance under shifting workloads and providing utility for a variety of workloads. Because false positive rates depend only on queries for negatives, we define a workload as a negative query distribution. Unless otherwise stated, the dataset used in the synthetic integer dataset from Section 5.2 and uses $\eta = 0.75$.

Stacked Filters Are Robust to Workload Shift. Figure 6a shows how Stacked Filters' performance changes with respect to workload change. In the experiment, Stacked Filters are built initially on the integer dataset from Section 5.2 and the given value of η. For higher values of η, the proportion of queries aimed at known negatives, ψ, is larger and so the corresponding Stacked Filter optimizes more for the FPR of known negatives. We then vary the workload by reducing the value of ψ from its initial value to a value of 0, and report the FPR on the changed query distribution. We see that for all Stacked Filters, drastic workload shifts are needed in order for them to become worse than a traditional filter, where traditional filters become better than Stacked Filters only after the known negative set receives less than 40% of its initial query requests. Even in the extreme case that ψ becomes 0, so that every frequently queried element when the Stacked Filter was built is no longer queried, the Stacked Filter is never more than 50% worse than a traditional filter.

Construction Scales Linearly. While Stacked Filters are effective under shifting workloads, they will need to be rebuilt to regain their best performance. Figure 6b shows how the construction varies with the size of the negative set. As the negative set grows to an extreme of 10× the positive set, Stacked Filters cost grows linearly, with a constant .25 second overhead for optimization. For even very high cardinality negative sets, construction finishes in under a second.

Stacked Filters Useful For Moderately Skewed Data. Figure 6c shows how Stacked Bloom Filters perform relative to traditional Bloom Filters at different EFPRs, where we compare the size required to reach the desired EFPR. We see that Stacked Filters are more space efficient than Traditional Filters for a wide range of skews. In particular, when the EFPR is low .1%, Stacked Filters have lower memory requirements across all skew parameters.

Hashing is more robust than Learning. We now demonstrate that Stacked Filters provide robust behavior regardless of the patterns in the data. To demonstrate this, we take the URL dataset and swap elements between the positive and negative sets, creating a pattern which is less predictable. Figure 6d) shows the results, where a Learned Filter with an EFPR of 1% sees its required space rise rapidly even when just a few percent of the positive elements are swapped with negatives. At 1% of elements swapped, it begins to require more space than a Stacked Bloom Filter, and at 3% of elements swapped it is worse than a traditional Bloom Filter. Thus, patterns which are harder to classify or for which Bayes Error is non-zero sharply reduce the performance of Learned Filters. For traditional and Stacked Filters, they are hash-based and so are unaffected by the existence or non-existence of patterns in the data. Thus, they see no change in performance.

5.4 Additional Results

Stacked Filters' Size Scales Linearly with Data. Traditional filters scale linearly in size with the number of elements keeping the bits per element equal to attain a given FPR. Because Stacked Filters are constructed from these filters and only store low constant factor of the number of elements, they also scale linearly in size with the number of positive and negative elements used in construction. The fourth graph in Figure 7 shows this as the bits per element required to achieve a given overall EFPR remains constant while the number of elements stored increases. In this experiment, the proportion of positive and negative elements is held constant as is the value of ψ.

Filters' Performance Converge for Frequent Positives. Figure 7 shows how the overall performance of Stacked, Learned and traditional filters varies when the proportion of positive queries is increased. This is shown in the context of a filter protecting RAID HDD storage, as in the fourth graph of Figure 4 in the main paper.

What this shows is that the total difference in overall performance between the filters remains similar, although the computational overhead of the filters does have more impact as the proportion of positives increases. However, the time spent on positive queries quickly becomes the bulk of the overall computational time as the percent of positive queries rises which makes the overall performance of each filter become very similar in relative terms.

3-Layer Are Often Sufficient. Figure 7 shows the performance of a 3-layer stacked filter versus a filter whose height is optimized between 1 and 7 layers. In general, the performance is very similar for all except the most extreme skewed distributions. This implies that in many situations 3-layer Stacked Filters capture the majority of the gains found by Stacked Filters.

6 DECIDING WHEN TO STACK

The analysis and experiments so far has shown that Stacked Filters are a generalization of traditional filters, and that Stacked Filters always perform at least as well as their traditional counterparts. When it comes to using Stacked Filters in real problems, though, there is one more question remaining. Should we invest in gathering workload information? This is not for free: it takes resources and time. And so if an application invests in doing so in order to use Stacked Filters but then the optimization algorithms of Stacked Filters decide that 1 layer is optimal, then the gathering of the workload was a wasted effort. As such it would be valuable if applications can make the decision to use Stacked Filters and thus go on to collect workload information in a more principled way to minimize these costs.

In this section, using summary statistics of workloads, namely the relationship between N_k and $\psi(N_k)$, and an input desired EFPR α, we provide an answer for 1) when are Stacked Filters strictly better than traditional filters and 2) by how much should I expect the Stacked Filter to improve over the traditional filter.

Stacked Filters are better across a range of realistic workloads. The following theorem assumes the size of a traditional filter can be written in the form $s(\alpha) = \frac{-\ln(\alpha)+f}{c}$, where f is a constant overhead and c is a constant multiplicative factor. We note that Bloom Filters, Cuckoo Filters, and Quotient Filters can all be put in this form.

THEOREM 6.1. *Let the positive set have size $|P|$, let the distribution of our negative queries be D, and let α be a desired expected false positive rate. If there exists any set N_k, $\psi = P(x \in N_k | x \in N)$, and $k \leq \psi$ such that*

$$\frac{|N_k|}{|P|} \leq \frac{\ln \frac{1}{1-k}}{\ln \frac{1-k}{\alpha} + f} \cdot \frac{1-k-\alpha}{\alpha} - 1$$

then there exists a Stacked Filter capable of achieving the same or better EFPR as a traditional filter with fewer bits.

The proof of theorem 9.1 can be found in the appendix 9.

Figure 8a and 8b show how this theorem can be used to create a visualization of which workloads Stacked Filters are certainly better than traditional filters on, in this case for Bloom Filters. Figure 8a shows this most clearly for a desired FPR of $\alpha = 0.03$; any workload which has a set N_k such that $|N_k|, \psi(N_k)$ is above the line has a Stacked Filter which is strictly better than a traditional filter. Figure 8b shows this trend for more alpha values. Even at a high desired α value of 0.05, Stacked Filters cover a sizeable percent of workloads; it seems reasonable to suggest that a negative set half the size of one's positive set contains 25% of all negative queries. As the desired FPR decreases, Stacked Filters cover a greater portion of workloads; for instance at a desired FPR of $\alpha = 0.01$, if $|N_k| = |P|$, then only 7% of negative queries need to be at values in N_k for the Stacked Filter to be more space efficient. At even lower values, the amount needed becomes negligible and almost any workload sees improvement from Stacked Filters. Thus, the trends seen for the Zipf distribution in Section 5 hold across all distributions: at low desired FPRs, Stacked Filters are superior to traditional filters.

Stacked Filters Produce Sizable Size Improvements. To estimate the size reduction from Stacked Filters, we limit ourselves to Stacked Filters with equal FPRs at each layer; as such, the reported numbers are a lower bound on the size improvement from using Stacked Filters.

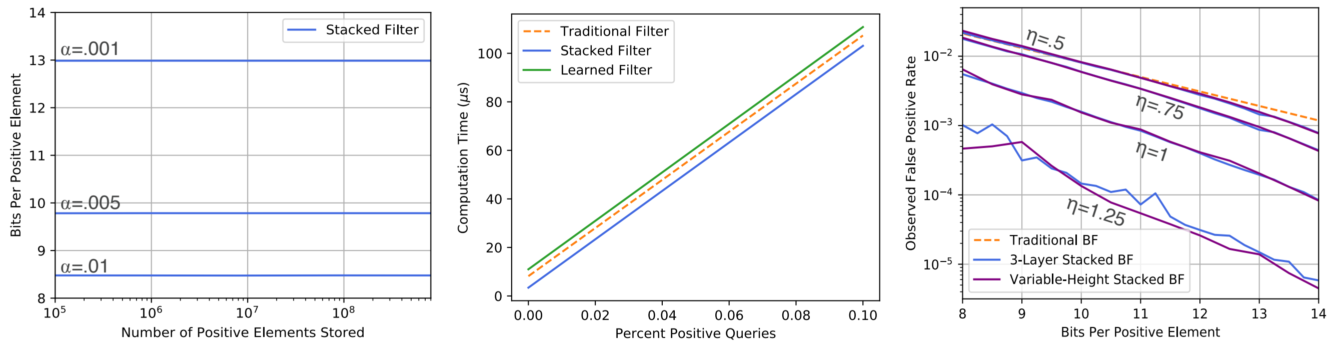

Figure 7: Stacked Filters scale linearly with data and maintain their relative gains as positive query rates increase. Most of these gains can be achieved by a simple 3-layer filter.

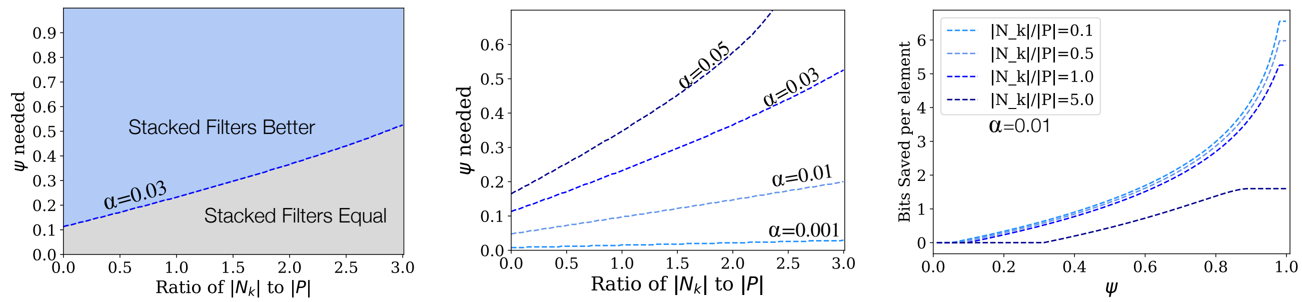

Figure 8: Analytical equations can predict both when Stacked Filters are better, and by how much.

As a Stacked Filter grows to larger and larger depths, we have that our equation for the EFPR, (1), becomes less than $(1-\psi)\alpha_L$, where α_L is the FPR of each layer in the stacked filter. Thus, for any $\alpha_L \leq \frac{\alpha}{1-\psi}$, we have that the Stacked Filter has a lower total EFPR than α. We can then use our equation for size, (2), fix any value of $\frac{|N_k|}{|P|}$, and minimize this with respect to α_L under the constraint that $\alpha_L \in (0, \frac{\alpha}{1-\psi}]$. Doing this leads to an equation for expected reduction in space for the same false positive rate:

$$s(\alpha) - \min_{\alpha_L} s(\alpha_L)\left(\frac{1}{1-\alpha_L} + \frac{|N_k|}{|P|}\frac{\alpha_L}{1-\alpha_L}\right)$$
$$s.t. \qquad \alpha_L \in (0, \frac{\alpha}{1-\psi}]$$

We note that this equation is quasiconvex for each of Bloom, Quotient, and Cuckoo Filters and so is easily solvable via gradient descent. Doing so returns the expected savings from using Stacked Filters in comparison to traditional filters given only summary properties of the workload: note that only $|N_k|$, $\psi(N_k)$, and α are needed.

Figure 8c shows a graph of how $\frac{|N_k|}{|P|}$ and ψ produce a reduction in filter size using Bloom Filters at a desired FPR of $\alpha = 0.01$. For each fixed value of $\frac{|N_k|}{P}$, the graph contains three parts: in the first part, it is not advantageous to build Stacked Filters and a traditional filter is built. For all values of $\frac{|N_k|}{|P|}$, this is a small area and covers workloads with almost no frequently queried negative elements. In the second part of the graph, we see that Stacked Filters have superlinear improvement in ψ, with improvement starting at 0 bits per element saved and going up to 7 bits per element saved. At the tail end of the graphs, the improvement stops even as ψ goes higher. This is the point where $1-\psi$ crosses $\arg\min_{\alpha_L \in (0,1)} s(\alpha_L)(\frac{1}{1-\alpha_L} + \frac{|N_k|}{|P|}\frac{\alpha_L}{1-\alpha_L})$; here, even though the Stacked Filter can choose larger false positive rates at each layer while having the same EFPR as the traditional filter, this larger α_L value increases size. Instead, the Stacked Filter keeps the minimal α_L value for size and the Stacked Filter produces both a savings benefit and has lower EFPR than the input α.

7 RELATED WORK

Sandwiched Learned Filters. Sandwiched Learned Filters (SLFs) are an extension to Learned Bloom Filters which use a preliminary filter before querying the classifier [27]. Similarly to our findings, the use of a sequence of filters brings increased space efficiency and reduced false positive rates. However, SLFs remain centered around a computationally expensive model, whereas Stacked Filters achieve performance similar to existing filters. In addition, Stacked Filters can extend to an arbitrary height, while extending SLFs with more layers would need to take into account the difficulties arising from using a series of dependent ML models.

Filters that Use Known Negatives. There is a limited existing literature on Bloom Filters which use a known negative set in construction in order to mitigate false positives. Yes-No Filters do this by keeping a set of additional "no" filters that store false positives [8]. Unlike Stacked Filters, Yes-No filters ensure no false negatives are produced by throwing out false positives that would cause a false negative if added to the "no" filter. Doing so asymptotically increases their computational costs and fails to utilize the full workload knowledge available. Complement Bloom Filters assume access to the full negative universe and create a Bloom Filter for both the full positive and negative sets [25]. This reduces the number of false positives, but requires a vast amount of space to store all negative elements and is limited to applications where all negatives are available at construction. Retouched Bloom Filters loosen the restrictions of filtering by allowing a small number of false negatives in order to heavily reduce the false positive rate; however, by allowing false negatives they no longer operate as a traditional filter [14].

Filter Variations. Many designs have been proposed to solve the standard filtering problem include the Cuckoo Filter, Quotient Filter, Morton Filter, and Blocked Bloom Filter [4, 6, 15, 31]. These advances benefit Stacked Filters as they can be included as layers in the stack. Other designs such as the Counting Quotient Filter, BloomFlash, Buffered Bloom Filter attempt to create a filter which successfully takes advantage of fast persistent storage [7, 11, 28]. Promising work has shown significant speedups in filtering using GPU acceleration which allows these structures to take advantage of recent hardware advances [19, 20, 26]. Recent work by [24] defined a model for discussing performance optimal filtering which strongly influenced our discussion of performance.

8 CONCLUSION

Stacked Filters provide a new facet to the filter design space. By taking advantage of workload knowledge, they provide a significantly reduced false positive rate while maintaining a similar query throughput to existing filters. We showed this technique can be fruitfully applied to a range of existing filter designs, and demonstrated its utility in URL blacklisting. Further, we provided bounds which describe the kinds of workloads that benefit from multi-layer stacks, showed that Stacked Filters are robust to changing workloads, and provide utility under a variety of skewed query distributions.

REFERENCES

[1] Shalla secure services kg. http://www.shallalist.de.

[2] Stacked filters technical report. https://github.com/AnonSigmodGit/StackedFiltersTechnicalPaper/blob/master/Stacked_Filters_Technical_Report.pdf.

[3] L. A. Adamic and B. A. Huberman. Zipf's law and the internet. *Glottometrics*, 3(1):143–150, 2002.

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

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

[6] A. D. Breslow and N. S. Jayasena. Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment*, 11(9):1041–1055, 2018.

[7] M. Canim, G. A. Mihaila, B. Bhattacharjee, C. A. Lang, and K. A. Ross. Buffered bloom filters on solid state storage. In *ADMS@ VLDB*, pages 1–8, 2010.

[8] L. Carrea, A. Vernitski, and M. Reed. Yes-no bloom filter: A way of representing sets with fewer false positives. *arXiv preprint arXiv:1603.01060*, 2016.

[9] 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.

[10] 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.

[11] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H. Du. Bloomflash: Bloom filter on flash-based storage. In *2011 31st International Conference on Distributed Computing Systems*, pages 635–644. IEEE, 2011.

[12] 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.

[13] 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.

[14] 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.

[15] 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. https://github.com/efficient/cuckoofilter.

[16] 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.

[17] 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.

[18] 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.

[19] T. Gubner, D. Tomé, H. Lang, and P. Boncz. Fluid co-processing: Gpu bloom-filters for cpu joins. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, page 9. ACM, 2019.

[20] A. Iacob, L. Itu, L. Sasu, F. Moldoveanu, and C. Suciu. Gpu accelerated information retrieval using bloom filters. In *2015 19th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 872–876. IEEE, 2015.

[21] S. G. Johnson. Nlopt.

[22] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: building a better bloom filter. In *European Symposium on Algorithms*, pages 456–467. Springer, 2006.

[23] 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.

[24] 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.

[25] H. Lim, J. Lee, and C. Yim. Complement bloom filter for identifying true positiveness of a bloom filter. *IEEE communications letters*, 19(11):1905–1908, 2015.

[26] L. Ma, R. D. Chamberlain, J. D. Buhler, and M. A. Franklin. Bloom filter performance on graphics engines. In *2011 International Conference on Parallel Processing*, pages 522–531. IEEE, 2011.

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

[28] 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. https://github.com/splatlab/cqf.

[29] G. Pike and J. Alakuijala. Cityhash, Jan 2011. https://github.com/google/cityhash/blob/master/src/city.h.

[30] 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 51–67, 1994.

[31] F. Putze, P. Sanders, and J. Singler. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms*, pages 108–121. Springer, 2007.

[32] 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.

[33] J. W. Rae, S. Bartunov, and T. P. Lillicrap. Meta-learning neural bloom filters. *arXiv preprint arXiv:1906.04304*, 2019.

[34] 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.

[35] 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):233–243, 2005.

[36] K. Singhal and P. Weiss. deepbloom2018. https://github.com/karan1149/DeepBloom.

[37] 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.

[38] 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.

[39] D. Wang, H. Cheng, P. Wang, X. Huang, and G. Jian. Zipf's law in passwords. *IEEE Transactions on Information Forensics and Security*, 12(11):2776–2791, 2017.

[40] G. K. Zipf. Selected studies of the principle of relative frequency in language. 1932.

9 APPENDIX

9.1 Proof of Theorem 1

We restate Theorem 1 here for convenience:

THEOREM 9.1. *Assume the size of a base filter in bits per positive element is* $s(\alpha_i) = \frac{-\ln(\alpha_i)+f}{c}$, *where* α_i *is the input false positive rate for this specific base filter. Let the positive set have size* $|P|$, *let the distribution of our negative queries be D, and let* α *be a desired expected false positive rate. If there exists any set* N_k, $\psi = P(x \in N_k | x \in N)$, *and* $k \leq \psi$ *such that*

$$\frac{|N_k|}{|P|} \leq \frac{\ln \frac{1}{1-k}}{\ln \frac{1-k}{\alpha} + f} \cdot \frac{1-k-\alpha}{\alpha} - 1$$

then there exists a Stacked Filter capable of achieving the same or better EFPR as a traditional filter with fewer bits.

PROOF. We limit ourselves to Stacked Filters which have an equal FPR α_L at each layer. In this case, we have that the EFPR of the filter is

$\psi \alpha_L^{T_L - 1/2} + (1-\psi)[(1-\alpha_L) * (\alpha_L + \alpha_L^3 + \alpha_L^5 + \ldots + \alpha^{T_L - 2}) + \alpha^{T_L}]$

For some N, $T_L \geq N$ implies this is less than $(1-\psi)\alpha_L$. Thus, a Stacked Filter of length N and $\alpha_L \leq \frac{\alpha}{1-\psi}$ has an EFPR of α or lower.

Our goal is then to show that there exists a Stacked Filter with $\alpha_L \leq \frac{\alpha}{1-\psi}$ which has size less than $\frac{-\ln(\alpha)+f}{c}$. From Section 4, we have that the size of a Stacked Filter is bounded above by

$$s(\alpha_L)\left(\frac{1}{1-\alpha_L} + \frac{N_k}{P}\frac{\alpha_L}{1-\alpha_L}\right)$$

Plugging in our equations for the size of a base filter, a Stacked Filter is better than a traditional filter if $0 \leq \alpha_L \leq \frac{\alpha}{1-\psi}$ and

$$\frac{-\ln \alpha_L + f}{c}\left(\frac{1}{1-\alpha_L} + \frac{N_k}{P}\frac{\alpha_L}{1-\alpha_L}\right) \leq \frac{-\ln \alpha + f}{c} \qquad (3)$$

We now let $\alpha_L = \frac{\alpha}{1-k}$, and our EFPR equation gives the restraint $0 \leq k \leq \psi$. We further break apart our equation for size into two parts: our goal will be to show that $\frac{1}{1-\alpha_L} + \frac{N_k}{P}\frac{\alpha_L}{1-\alpha_L} \leq 1 + b$, and that $(1+b)\frac{-\ln \alpha_L + f}{c} \leq \frac{-\ln \alpha + f}{c}$. If both equations are satisfied, then (3) is satisfied as well.

Part 1: $(1+b)\frac{-\ln \alpha_L + f}{c} \leq \frac{-\ln \alpha + f}{c}$: Plugging in $\alpha_L = \frac{\alpha}{1-k}$ and solving this inequaliity gives: $b \leq \frac{\ln 1-k}{\ln \frac{\alpha}{1-k}}$.

Part 2: $\frac{1}{1-\alpha_L} + \frac{N_k}{P}\frac{\alpha_L}{1-\alpha_L} \leq 1 + b$ Rearranging, we get $\frac{|N_k|}{|P|} \leq \frac{b}{\alpha_L} - 1 - b$. We now set $b = \frac{\ln 1-k}{\ln \frac{\alpha}{1-k}}$, satisfying the inequality above, and plug in $\frac{\alpha}{1-k}$ for α_L. The resulting equation is

$$\frac{|N_k|}{|P|} \leq \frac{\ln \frac{1}{1-k}}{\ln \frac{1-k}{\alpha} + f} \cdot \frac{1-k-\alpha}{\alpha} - 1$$

as desired, with the constraint that $0 \leq k \leq \psi$. □

9.2 Optimization Details

Here we describe in detail the method used in the paper to determine the EFPR of each layer in the stack.

The Governing Equations: FPR and Size. When performing the optimization in either the continuous case or the following to equations are the minimization objective and the constraint. Which is which depends on whether there is a target EFPR or a target memory budget being optimized against. The first equation is independent of the underlying filters, but the size equation relies on the function $s(\alpha)$ which is the minimum bits per element required to achieve a given EFPR for a specific filter type.

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

$$SIZE = \sum_{i=0}^{\frac{T_L-1}{2}} s(\alpha_{2i+1}) * |P| * (\prod_{j=0}^{i} \alpha_{2i}) + \sum_{i=1}^{\frac{T_L-1}{2}} s(\alpha_{2i}) * |N_k| * (\prod_{j=1}^{i} \alpha_{2i-1})$$

Continuous Optimization for Stacked Bloom Filters. When optimizing Stacked Bloom Filters, the function s is found by first calculating a number of hash functions then inverting the exact expression for the probability of a false positive. The number of hash functions, k, is found by rounding the following expression for the optimal number of hash functions for a given FPR, α,

$$k = \log_2(\alpha)$$

This is plugged into the inversion of the following,

$$\alpha = (1 - (1 - \frac{1}{m})^{kn})^k$$

$$m = \frac{1}{(1 - (1 - (\alpha)^{1/k})^{1/kn}}$$

With this, we have fully defined EFPR and Size equations which take the EFPRs of every layer as input. However, a set number of known negatives and a value of ψ is assumed in these equations. In order to optimize the number of negatives taken given a distribution like the Zipf distribution, we include the number of known negatives as a variable to be optimized and calculate the value of ψ from the distribution. For the Zipf, this simply means setting ψ equal to the cdf of the distribution at x equals the number of known negatives.

At this point, we do a two-stage optimization over the modified EFPR and Size functions. In the first stage, we enforce that every layer's EFPR be equal and calculate the optimal EFPR/Size for all layers subject to the Size/EFPR constraint using the ISRES global optimization algorithm from the NLOPT package [21, 35]. Then, we "polish" this equal set of EFPRs for every layer by running the local optimization algorithm COBYLA where the minimization and constraint functions are the same except that the layers can all have different EFPRs.

Integral Optimization for Stacked Cuckoo and Quotient Filters. Cuckoo and Quotient Filters' size and EFPR is fully determined by the integer length of the hash fingerprint stored for each element and the number of elements, and these lengths fall within a relatively small set of reasonable values, in particular between 2 and 20. Therefore, the optimization algorithm for a Stacked Cuckoo/Quotient Filter can simply check every combination of these values for each layer and take the parametrization which minimizes the EFPR/Size and fulfills the constraint on Size/EFPR. Naively, this requires calculating the size and EFPR of 18^{T_L} filter configurations. For 3 layer stacks, this is a very feasible operation as it only results in 5832 potential configurations. However, for larger stacks, it is important to begin intelligently limiting the search space.

In practice, this optimization process involves a nested for-loop with a depth equal to the depth of the stack. The outermost for-loop iterates through the potential parameters for the first layer in order from 2 to 20, then the next outermost for-loop does the same for the second layer and so forth. At each nested layer of the loop, the running size of the Stacked Filter is calculated by adding up the size of the layers whose parameters are set, i.e. after passing through the first i nested loops, the running size is calculated by summing the first i layers of the Stacked Filter. The same is done for the EFPR.

These running sums allow us to restrict the search space based on the constraint. Once the running size/EFPR has exceeded the size/EFPR constraint, no choice in parameters of the latter layers will be able to lower the size/EFPR. Therefore, any further nested for-loops, which iterate through possible parameters of latter layers, can be skipped. By skipping these configurations, the optimization is significantly sped up. Other optimizations of this process are still being analyzed.