Efficient Filtering: Stacking and Hash Functions as Data

Efficient Filtering: Stacking and Hash Functions as Data

Kyle Deeds

April 3, 2020

Abstract

Approximate filtering is central to a wide variety of high performance systems in computer science. Data structures like Bloom Filters which perform this filtering answer set membership queries while guaranteeing zero false negatives and some small rate of false positives. Traditionally, these filters represent a set of elements, referred to as positives, which require expensive processing, and they protect the system from having to perform that processing on elements outside of that set, referred to as negatives. To improve overall system performance, the overhead of evaluating the filter must be less than the saved processing time, so filters must fit in memory while still producing very few false positives.

Worryingly, there is a strong lower bound on the size of a filter for a given false positive rate, and state-of-the-art filters are quickly approaching it. However, this lower bound requires strong assumptions about the available workload knowledge which do not hold in many real world contexts. Seeking to break this lower bound, several papers have suggested ways to incorporate additional workload knowledge to create better filters[12][14][5]. However, current suggestions are generally either computationally infeasible for most filtering applications or do not take full advantage of the available workload knowledge.

In this thesis, two original methods are presented for reducing the size of filters by incorporating workload knowledge. The first method, Stacked Filters (SFs), can adapt any existing filter design to drastically reduce the false positive rate. The utility of stacking is demonstrated using URL blacklisting, IP filtering, and synthetic data, showing that SFs achieve FPRs 10x lower than traditional filters on real world datasets. Then, a theoretical analysis is presented showing the memory footprint, computational cost, and robustness of SFs. Lastly, a method for incorporating false positives in an online manner is presented alongside experimental results demonstrating its efficacy in increasing overall performance.

The second method, Variable Hash Filters (VHFs), introduces an entirely new filter design based on a data structure from the networking literature [28]. VHFs are more space-efficient than state-of-the-art traditional filters for low memory budgets even in the absence of workload knowledge, and they very efficiently incorporate workload knowledge. Initial experiments and theoretical results are presented which show the FPR and size of the filters as compared to traditional methods.

Acknowledgements

I would like to thank Brian Hentschel and Professor Stratos Idreos for all that they have done to set me on the path I'm on today. For their contributions to this work, I am grateful. For their kind mentorship and encouragement, I am forever indebted.

Thank you as well to Professor Michael Mitzenmacher who gave me the tools to explore the most interesting problems in CS and provided a model for the kind of researcher that I hope to be one day.

To my friends and family, who never wished to learn about Bloom Filters but listened to my excited rambling anyways, thank you and I love you.

Contents

1 Introduction

As data sizes continue to grow, having concise means of summarizing that data is increasingly important for the efficient operation of systems that rely on it. These summaries can be stored in memory and provide fast, if approximate, answers to questions about that data even when the data itself is too large to store in memory. One of the most fundamental questions in the context of databases and networking is the membership query, "is element X a member of set S?", and many means of summary have been proposed to answer this question.

Generally referred to as filters, summary structures which answer this question have existed since 1970 when BH Bloom suggested the first and most popular one, the Bloom Filter[4]. Using only 10 bits of storage per element in the set S, a Bloom Filter can provide answers to membership queries on S while guaranteeing no false negatives and a false positive rate as low as 1%. This remarkable space efficiency led to their use in almost every sub-field of computer science.

1.1 Motivation

Because of this widespread use, new structures are presented every year which improve various aspects of these filters making them faster and more versatile. However, a lower bound on the space efficiency of filters was proved in 1978 which stated that, under the classic problem formulation, a filter must use at least $-\log(\varepsilon)$ bits per element to provide a false positive rate of ε, and modern filters are approaching that bound with state-of-the-art filters using only $-\log(\varepsilon)+2.125$ bits per element[6][18]. At first glance, this is a fairly damning result for the study of filters implying that future work in more space efficient filters will yield meager performance gains.

Fortunately, the classic problem formulation is far more restrictive than the typical use case for these filters. In particular, the 1978 lower bound assumes 1) a very large universe of elements that could be queried against the filter, 2) equal query frequency for each of those elements, and 3) a represented set chosen randomly from that universe of possible elements. Real world applications routinely break these assumptions in ways that a filter can take advantage of to obtain even further space efficiency.

The third assumption implies that there is no learn-able classification between the elements in the represented set and the rest of the universe. As this is frequently not true, recent work has found success in using ML classifiers to boost the performance of traditional filters beyond the lower bound discussed above. Future work in this area is promising, but there remain serious concerns about the computational and engineering cost of placing ML in the center of these high performance, low effort data structures.

For this reason, this thesis focuses on hash-based methods which take advantage of violations of the first two assumptions to break through the traditional lower bound while preserving the fundamental qualities of traditional filters. The new designs presented in this paper, Stacked Filters and Variable Hash Filters, benefit from skewed negative query distributions and relatively small universes. In these settings, they boast a significantly reduced false positive rate and the hallmark low query latency of hash-based filters.

1.2 Uses for Filters

In the most general sense, filters are used to guard against expensive operations in cases where only a subset of elements will provide useful results. This subset of elements is generally referred to as the positive set.

A classic example of this is looking up a key-value pair from a database stored on disk. If the key is present in the database, then the system must retrieve the associated value from disk in order to maintain the basic guarantees of a database. If the key is not present in the database, then the system should avoid the expensive disk read if possible. Because of this, state-of-the-art key-value stores maintain filters in main memory representing the set of stored keys and perform a filter check before accessing disk. By doing this, they avoid approximately 99% of the unnecessary disk accesses.

In networking contexts, filters are often used to concisely represent the resources available in other nodes of the network. For example, distributed web caching systems such as Summary Cache often have each node maintain a bloom filter for each other node connected in the network which represents the web pages cached at that node. This same paradigm is applied to peer to peer networks by Druschel and Rowstron to manage distributed hash tables. [8] [24] In other cases, filters are able to speed up distributed computation such as approximate set intersection and bitcoin transactions. [23][10]

In database contexts, filters generally serve a similar purpose to the key-value store example above where they summarize the data present on a slower medium, e.g. HDD, SDD, or networked storage. They are also used to speed up distributed join computations by summarizing the list of join keys efficiently.[22]

1.3 Formally Filtering

A filter can be thought of as a function, F, which maps elements from some potentially infinite set U to an element in $\{0,1\}$. For some arbitrary set $P \subset U$ known at construction, this function guarantees that

Symbol	Definition
F	A function representing the filter
U	Set of all query-able elements
P	Set of all positive elements
N	Set of all negative elements
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: Formal notation for filtering

every element is sent to 1. In other words, it does not permit false negatives,

$$F(x) = 1 \ \forall \, x \in P$$

In order to be useful, filters must also rarely accepting negative elements. However, this property can be defined in two semantically different ways. Traditional filters provide an expected false positive bound, EFPB. This guarantees that the probability of any particular element in the negative set, defined as $N = U \setminus P$, being a false positive is less than the EFPB, i.e.,

$$\mathbb{P}(F(x) = 1) \leq \text{EFPB} \ \ \forall x \in N$$

This is a useful property because it upper bounds the expected FPR regardless of the query distribution. However, there are strong theoretical bounds of the EFPB which can be achieved in a given amount of space. Because of this, modern work on filtering has produced filters which are tailored to a particular query distribution in order to produce greater performance. To examine the performance of these filters, we consider another metric, EFPR, i.e. the expected false positive rate under a particular negative query distribution, D.

$$\mathbb{E}_D(F(X)) = \sum_{x \in N} F(x)\mathbb{P}_D(X = x) = \text{EFPR}$$

1.4 Current Filter Designs

Existing traditional filter designs can be classified in two broad categories: bit-array filters and fingerprint filters. Non-traditional filters, those which take advantage of broken assumptions in the lower bound discussed above, are either ML-based or hash-based,

Bit-Array Filters: The original Bloom Filter and many variants of it rely on an array of bits which begin set to 0. As elements are added, they are sent to a set of k locations determined by a set of k hash functions, and the bits in those locations are set to 1. When queried, the same k hash functions are used to send an element to k locations, and it is accepted only if the bits at those locations are all set to 1. This results in a false positive rate of approximately 2^{-k} when the filter is properly loaded because the bits in the array are equal parts 1's and 0's. Optimizations on these filters primarily focus on ways of making sure all of the k locations for an element fall within the same cache line or reducing the number of expensive hash functions that need to be computed[21][11].

Fingerprint Filters: The other branch of filter structures focus on generating a hash fingerprint of an element then storing it in a hash table in such a way as to reduce the number of comparisons possible during a lookup. The false positive rate in this case is approximately equal to $2^{-q} * c$ where q is the length of the fingerprint and c is the number of comparisons required. The two most popular filters of this variety are Cuckoo Filters and Counting Quotient Filters[7][18]. The first takes advantage of cuckoo hashing to maintain up to 95% load while checking at most 4 hashes on average. The second achieves the same load while checking only a single hash, but it uses an additional two bits per element of metadata.

A thorough comparison of these designs recently demonstrated that the tradeoffs between the throughput and false positive rate of these filters results in a situation where no filter is optimal for every application[13]. In particular, throughput-optimized Blocked Bloom Filters are performance optimal in cases where the operation being guarded against is relatively cheap while Cuckoo Filters were preferable for their lower false positive rate when guarding expensive operations.

Learned Filters: Learned Filters are ML-based and work 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 [12][16]. 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. It also means that the EFPR of the Learned Filter is the sum of it's classifier's EFPR and the backup filter's EFPR.

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%.

These benefits are however balanced by a new set of challenges introduced by Learned Filters. By placing an ML model at their core, Learned Filters require the collection of a representative dataset, the evaluation of an expensive operation within the filter, and the technical knowledge to run an ML system. These additional requirements are far from damning, but they do remain fundamental barriers to the adoption of Learned Filters today.

Complement Filters and Yes-No Filters: Complement and Yes-No Filters are recently proposed methods which use information about the negative set to decrease the EFPR of filters [14][5].

The Complement Filter assumes full knowledge of the negative set and builds a filter for both the negatives and the positives. By checking both filters during a query, the membership can typically be determined. In cases where both filters register a positive, a backup hash table is consulted which stores all of the elements in exact form.

The Yes-No Filter does not assume full knowledge of the negative set, but it is very restrictive in which false positives it can mitigate. It stores additional bloom filters referred to as "no-filters" which hold information about a false positive. However, these filters must be constructed such that a positive element is not a false positive for the no-filters. Because of this, it is often not possible to mitigate a false positive under this scheme.

These two designs both incorporate the concept of mitigating false negatives and make strides to do so, but further progress is necessary to make this a usable strategy.

Adaptive Cuckoo Filters: Recent work on Adaptive Cuckoo Filters has provided a method for modifying a traditional filter structure, Cuckoo Filters, to nullify some number of false positives as they arise[17]. In that work, when a false positive is recognized the hash function applied to the cell which caused the false positive is changed, and a backup cuckoo hash table which stores the elements in the set is consulted to generate the new fingerprint. This technique produces significant reductions in the false positive rate as compared to a traditional cuckoo filter depending on the size and skew of the negative queries. This design provides perhaps the most actionable method of nullifying false positives to date. However, the most efficient form of Adaptive Cuckoo Filter is limited in the number of false positives which it is able to nullify. A more detailed performance comparison with this method would be interesting but must unfortunately be left to future work due to time constraints.

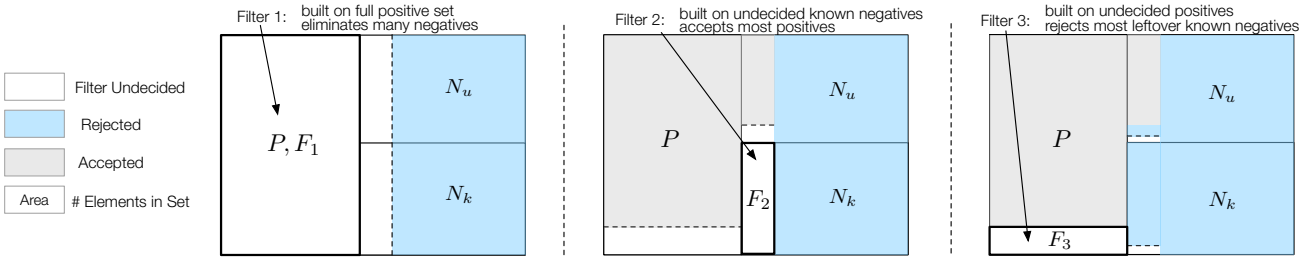

Figure 1: A conceptual diagram of the space of elements each filter uses during construction, and of the sets they accept/reject. As we descend down the stack (from left to right in the figure), filters become exponentially smaller.

2 Stacked Filters

Acknowledgements and Co-Authorship: This chapter of my thesis was created in a year-long collaboration with Brian Hentschel and Stratos Idreos. It began with me presenting the initial idea for Stacked Filters to them in the fall of 2018 and will hopefully end with it finding a home in a conference proceedings this Spring. Over the course of that year, they patiently edited and rewrote the following text and contributed their own genius and work time and again to this project.

Chapter Overview: This chapter presents Stacked Filters (SFs), a new method for adapting traditional filtering structures to take advantage of additional workload knowledge. By doing so, SFs achieve EFPRs significantly lower than traditional filters while preserving their incredible computational efficiency. This allows them to produce significantly higher overall performance in real world datasets.

We begin the chapter by describing SFs' algorithms in detail before moving on to a theoretical analysis of their EFPR, size, computational cost, and robustness. Then, we perform a thorough experimental analysis of SFs in relation to the current state-of-the-art solutions on a variety of real-world and synthetic workloads.

2.1 Algorithm Overview

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

2.1.1 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 $\mathscr{U}$ 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 1.

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 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 1. 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 1; 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 1.

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 1), 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 1. 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 1 would become accepts. This is similar to a traditional filter, which can be seen by comparing to the left side of Figure 1.

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 1 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 1, 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.

2.1.2 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 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

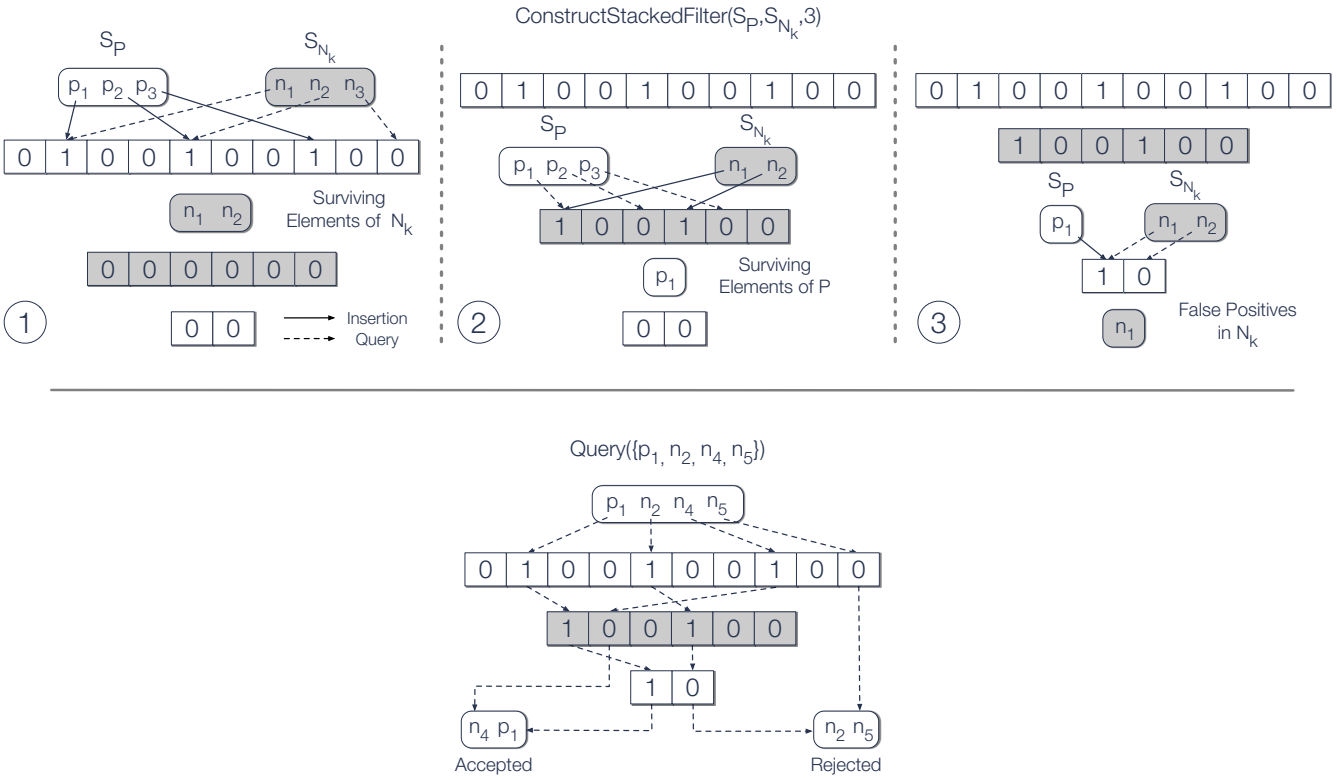

Figure 2: Stacked Filters are built in layer order, with each layer containing either positives or negatives. The set of elements to be encoded at each layer decreases as construction progresses down the stack. For queries, the layers are queried in order and the element is accepted or rejected based on whether the first layer to return not present is negative or positive.

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 2 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 2 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 $1/100$th of the negative elements would reach the second layer, and similarly only $1/100$th 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.

2.1.3 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.

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 2, which shows an example of querying for 4 elements from the filter constructed in Figure 2. 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

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

Require: S_P, S_{N_k}: sets of positive, known negative elements

// Construct the layers in the filter sequentially.
for $i = 1$ to T_L **do**
  if $i \bmod 2 = 1$ **then** {layer positive}
    // Insert positive set, Filter negative set
    $S_r = \{\}$
    for $x_p \in S_P$ **do**
      L.INSERT(x_p)
    end for
    for $x_n \in S_{N_k}$ **do**
      if L.LOOKUP(x_n) $= 1$ **then**
        $S_r = S_r \cup \{x_n\}$
      end if
    end for
    $S_{N_k} = S_r$
  else
    // Insert negative set, Filter positive set
    $S_r = \{\}$
    for $x_n \in S_{N_k}$ **do**
      L.INSERT(x_n)
    end for
    for $x_p \in S_p$ **do**
      if L.LOOKUP(x_p) $= 1$ **then**
        $S_r = S_r \cup \{x_p\}$
      end if
    end for
    $S_P = S_r$
  end if
end for
return L

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 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 2, 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 2 shows how this process works for unknown negatives n_4 and n_5 as well as known negative n_2.

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: **end if**
9: **end if**
10: **end for**
11: **return** accept {No layer rejected, accept x}

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**
8: **end if**
9: **end if**
10: **end for**

2.1.4 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 querying

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.

```
// Delete x_P from positive layers, until a negative layer rejects.
for i = 0 to i = T_L − 1 do
    if L[i] is a positive layer then {i mod 2 = 0}
        L[i].delete(x_P)
    else
        if L[i].lookup(x_P) = 0 then
            return
        end if
    end if
end for
```

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).

2.1.5 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, 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 [9, 25]. Afterwards, we perform local search using the COBYLA algorithm without the constraint that each layer have equal FPR [20].

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.

2.2 Theoretical Analysis

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.

Stacked Filters 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.

2.2.1 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

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

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.

2.2.2 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

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

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.

2.2.3 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 $|N_k|c_q + |N_k|(c_i + c_q)\alpha_1 + |N_k|(c_i + c_q)\alpha_1\alpha_3 + \cdot$, 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:

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

2.2.4 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 2.1.5 can be adapted to assign sizes to each layer using a weighted combination of the expected performance and robustness or a constraint on robustness.

2.3 Experimental Evaluation

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 [12] 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.

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. In experiments without an explicit distribution, we control both variables by defining the negative query distribution as a Zipf distribution and adjusting 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, 29, 27]. 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 [19]. The Bloom Filter implementation utilizes double hashing in order to speed up computation [11]. The Counting Quotient Filter (CQF) implementation is the one provided by the authors of the original paper [18]. 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 [7]. 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 [7]. For the Learned Filter, we use a 16 dimensional character-level GRU similar to the one used in [12] 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

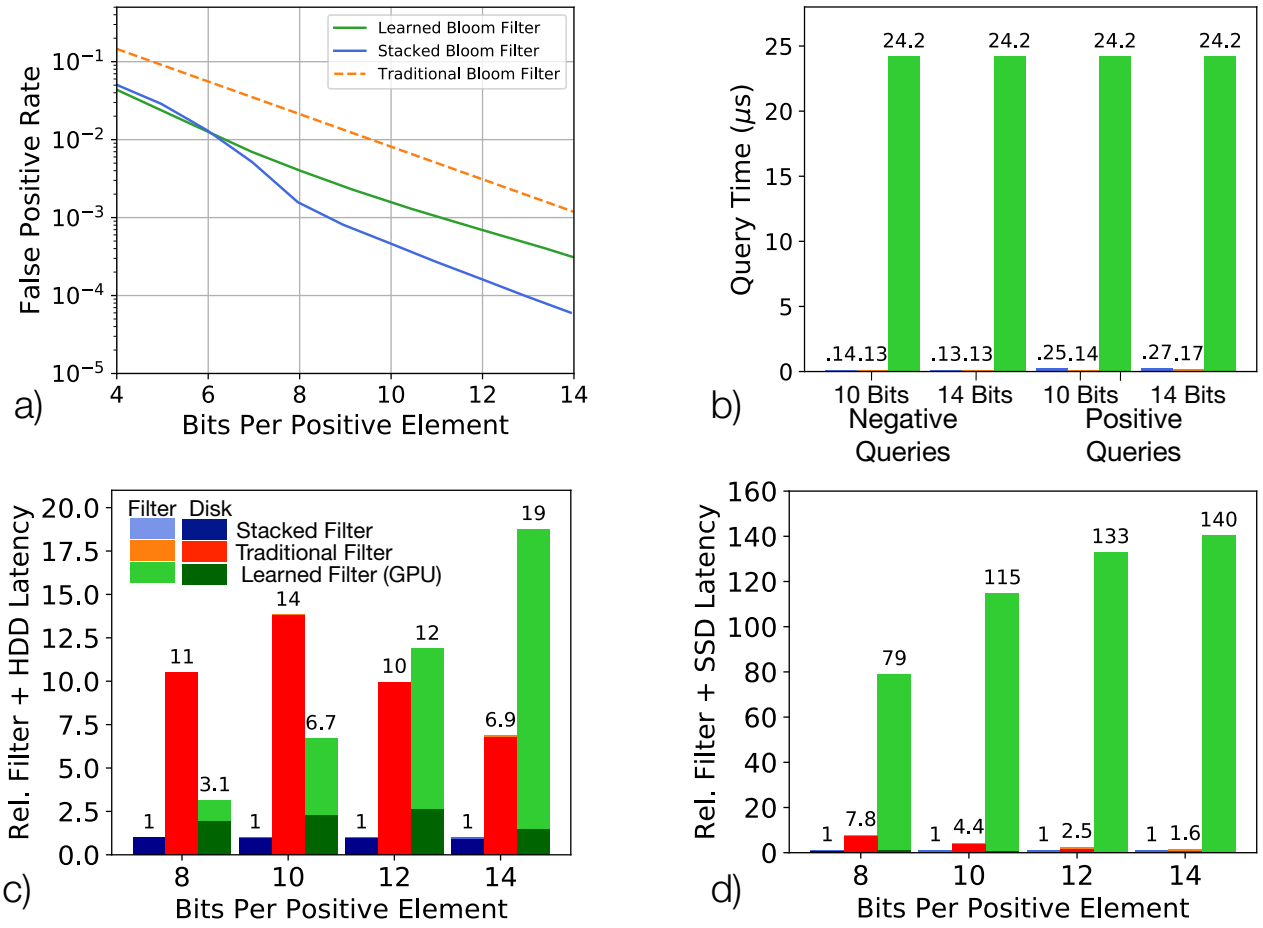

Figure 3: Stacked Bloom Filters achieve an equivalent or superior EFPR to Learned Bloom Filters while maintaining a 70x throughput advantage and beating both traditional and learned filters in overall performance on typical workloads.

the effects of noise, each experimental number reported is the average of 25 runs. For the performance comparisons using different storage mediums, we use a Seagate BarraCuda Pro ST1000 HDD and a Samsung SSD 970 EVO Plus.

2.3.1 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 [12] is not publicly available, we use two open-source databases of URLs, Shalla's Blacklists originally used by Singhal and Weiss [1, 26] and the top 10 million websites from OpenPageRankInitiative[2]. The blacklist contains approximately 2,800,000 URLs which we use for the positive set while we use the 10,000,000 websites from OpenPageRankInitiative as the negative set. We provide Learned and Stacked Filters with the higher frequency half of the negative set for training to simulate having incomplete information about the query distribution. Lastly, the negative query distribution is proportional to the pagerank of the websites.

Workload Knowledge Improves Filters: Figure 3a shows that across all filter sizes, both the Learned Filter and the Stacked Filter achieve lower false positive rates than the Traditional Bloom Filter, with improvements ranging from $5\times$-$13\times$. By utilizing workload knowledge both techniques produce substantial benefits in the compression of filters. For smaller filters, the Learned Filter has a slightly better false positive rate than the Stacked Filter; however, for higher bits-per-element, the Stacked Filter is dramatically better and provides up to a $6\times$ improvement over the Learned Filter.

For Stacked Filters, the workload skew allows it to find a small set of negative elements that contains a large portion of the target query workload. The negative filters can therefore be small in size while still 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 results in very effective classification as compared to traditional filters. However, the efficiency of Learned Filters begins to fade away at very low 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 requires a check against the backup filter. 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 generate too many false positives.

As a result, more and more positive examples have $f < \tau$ and need to be inserted into the backup filter. 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 3b 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 $1.5\times$ the cost of the Bloom Filter, even though it has about $2\times$ 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 $90-190\times$.

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 unnecessary data accesses incurred by false positives. While Figure 3b gives the overhead incurred by filter checks for each technique, the cost of unnecessary data accesses depends both on the filter's EFPR and the speed at which base data accesses occur. Figures 3c and 3d show the results when using two different kinds of hardware to store the base data while the filters reside in memory: 1) a modern HDD, and 2) a NVMe SSD.

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 have both better false positive rates and better computational performance; further, 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. In these experiments, each false positive results in a single random disk access.

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

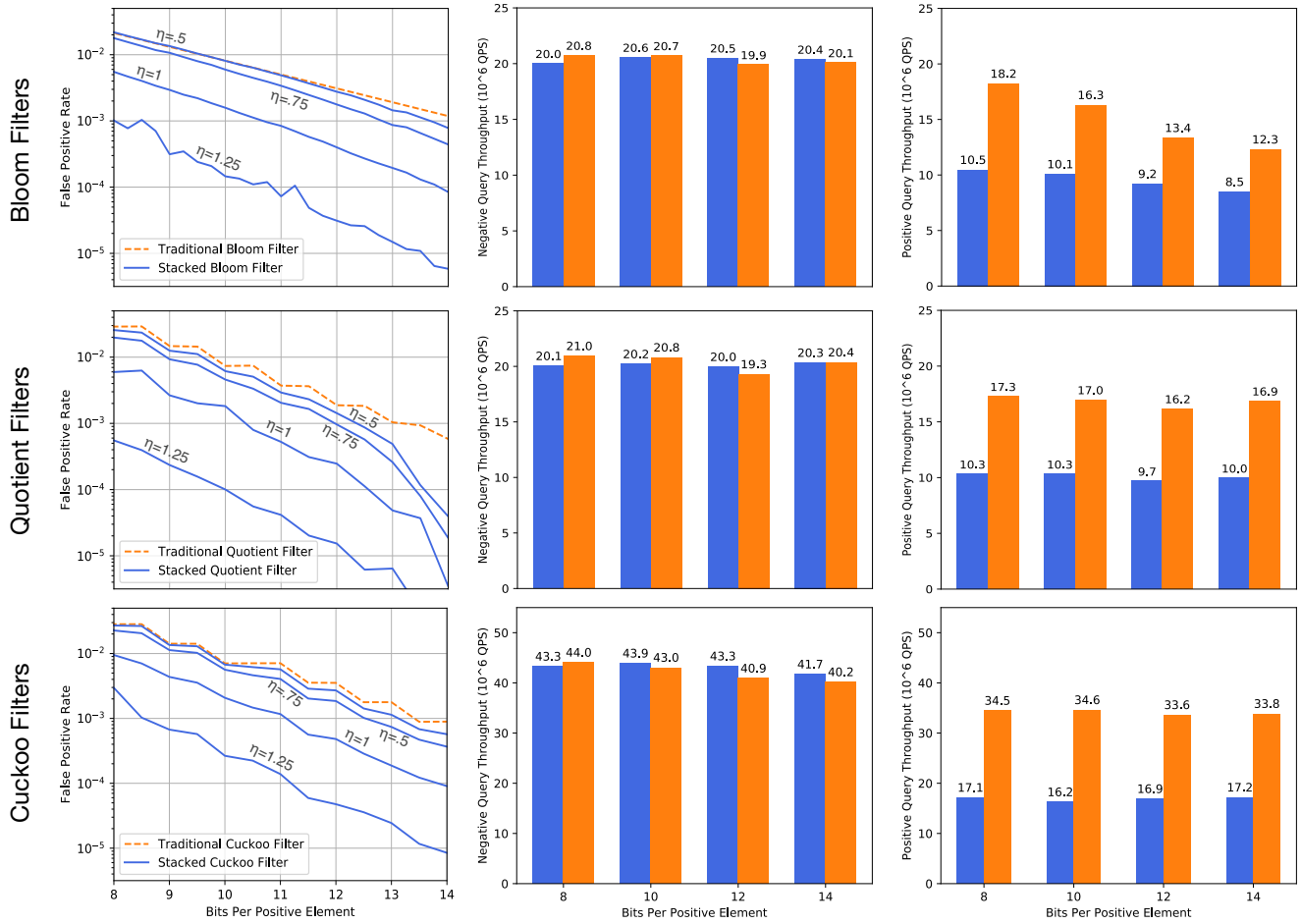

Figure 4: Stacking provides benefits across a variety of underlying filter types. Stacked Filters achieve a comparable throughput to traditional filters while achieving a drastically reduced EFPR.

only a tight storage budget is available (e.g., see left side of Figure 4a) which makes them a candidate for edge devices with little memory but access to hardware accelerators.

2.3.2 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 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

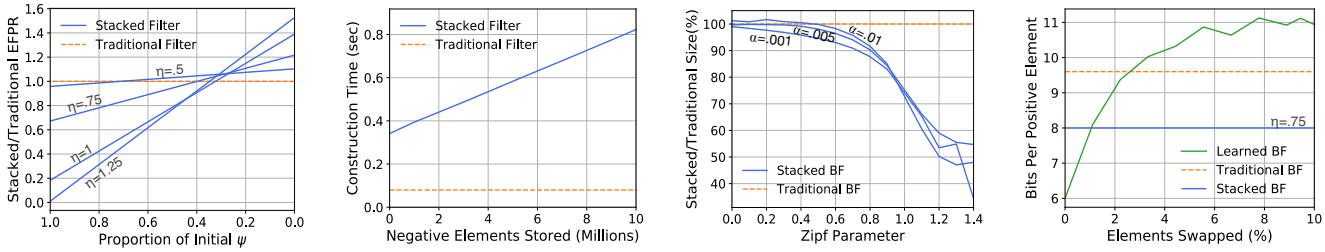

Figure 5: Stacked Filters are robust to workload shifts, can be rebuilt quickly, work across a large range of workload skews, and are immune to noisy data.

data has no effect on performance. For this dataset, Stacked Filters choose an optimal number of most popular negative elements as described in Section 2.1.5.

Stacked Bloom Filters: Like we have seen for the URL data, Figure 4a 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 2.1.5, 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 4 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.

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, the computational costs of Stacked Cuckoo and Quotient Filters follow the same patterns as seen in Figures 3b-d.

2.3.3 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 2.3.2 and uses $\eta = 0.75$.

Stacked Filters Are Robust to Workload Shift: Figure 5a 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 2.3.2 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 5b shows how the construction varies with the size of the negative set. As the negative set grows to an extreme of $10\times$ 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 5c 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.

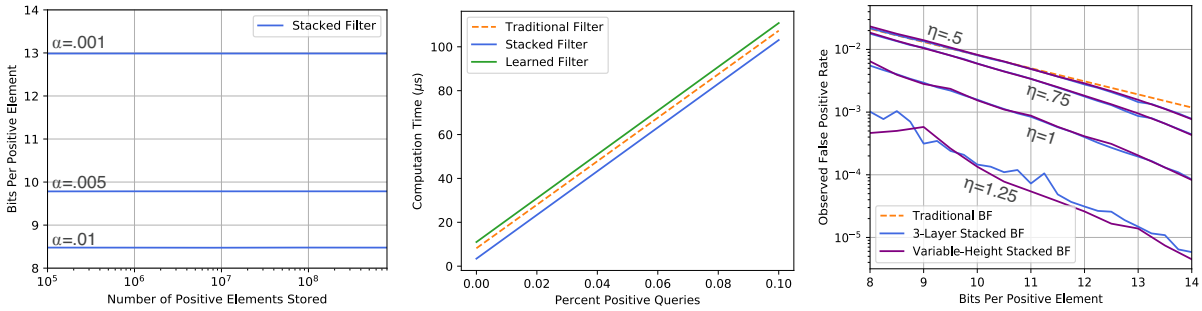

Figure 6: Stacked Filters scale linearly with data and maintain their relative gains as positive query rates increase. Most of these gains can be achieved by a simple 3-layer filter.

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.

2.3.4 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 6 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 6 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.

Algorithm 5 DeclareFalsePositiveAndCheckIfFull(x)

Require: L: the array of filters in the Stacked Filter.
FPCapacity: the max number of false positives to declare.
FPCount: the number of declared false positives.

```
1: if FPCount ≥ FPCapacity then
2:     return true
3: end if
4: if L[2].LookupElement(x)==false then
5:     L[2].InsertElement(x)
6:     FPCount++
7: end if
8: return false
```

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 6 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.

2.4 Adaptive Stacked Filters

This section describes a different method for constructing a Stacked Filter which allows for the gradual accumulation of a negative set, Adaptive Stacked Filters (ASFs). With this method, the additional requirements beyond those of the traditional filter are simply general information about the negative query distribution's skewness and the ability to flag false positives as they arise. We also limit the number of layers in this version of the filter to 3 to simplify the design and reduce additional construction time. Despite this relaxation of SFs requirements ASFs still outperform traditional filters significantly on the URL blacklisting task.

2.4.1 ASF Construction

There are three stages in ASF construction; cold construction, cold lookup, and warm construction. During cold construction, the size of all three layers is determined via an adjusted optimization method, and the first layer is constructed from the positive set. At this point, the cold lookup phase begins, and the SF begins operation by simply returning the results of the first layer. The only difference from regular

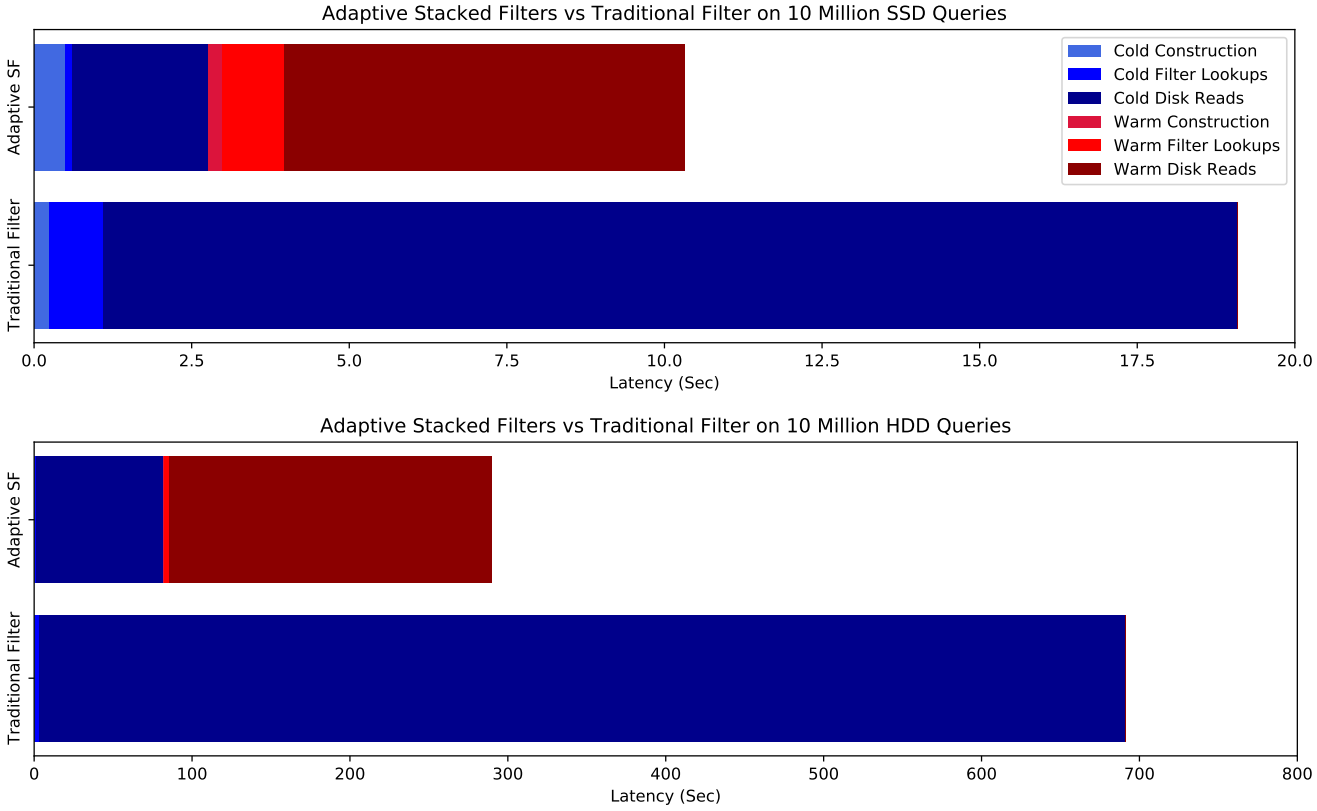

Figure 7: .

use is that when a false positive is generated the user alerts the ASF by calling Alg 5. This allows the SF to build the second filter over time as false positives arise. Once the second filter is saturated, Alg 5 begins to return "true", and the user is able to choose a convenient time to build the third layer. This warm construction simply requires reading in the positive set, testing them against the second layer, and inserting any false positives into the third layer. The lookup function of the SF is then updated to use all three layers.

2.4.2 Optimizing Sampling

To choose the optimal configuration in the adaptive setting, the SF must determine two interconnected things. First, it must decide the number of sample queries which it will witness in the "cold lookup" phase, referred to as $|N_{SQ}|$. In order to do this, a total number of queries is chosen and the SF optimizes the overall EFPR in the cold and warm periods of operation given that lifespan. A longer "cold lookup" phase results in a higher percent of false positives which are captured in the second layer, but it also results in

less of the total time spent taking advantage of the SF's lower EFPR. Second, the SF must still determine the allocation of bits to the different layers the same as previously.

The important variables which must be calculated to do this optimization are the ψ and $|N_k|$ values which result from a particular number of sample queries. However, because the sample queries are randomly drawn from the query distribution, these values become random variables themselves in the adaptive setting. Therefore, we substitute them with their expected values and calculate that instead. The equations for these values given a query PDF, f, are as follows,

$$\mathbb{E}(\psi) = \sum_{x \in N} f(x)(1 - (1 - f(x))^{|N_{SQ}|})$$

$$\mathbb{E}(N_k) = \sum_{x \in N} (1 - (1 - f(x))^{|N_{SQ}|})$$

Because these are relatively slow to compute for large $|N|$, only a relatively limited number of potential sample sizes are considered, and these values are only calculated for them.

2.4.3 Experimental Results

To test the performance of ASFs, we implemented them and compared them against traditional filters on the URL Blacklisting dataset. The workload is set at 10 million queries, and the filters are guarding against SSD and HDD reads. We include all stages of construction and querying in this performance measure including optimization, cold construction, cold lookup, warm construction, and warm lookup.

7 shows that ASFs result in significantly lower overall costs over the lifespan of a filter. This is due to the fact that the warm lookup phase spends significantly less time doing disk reads than the traditional filter does. This compensates for the additional overhead expended during the warm construction phase and the optimization time spent in the cold construction phase. The savings are greater both proportionally and in total in the HDD context because the slower disk reads of the HDD dominate the performance in that context, so the lower FPR achieved by ASFs in the warm phase has a larger impact.

3 Variable Hash Filters

Chapter Overview: This chapter presents Variable Hash Filters (VHFs), a new filter design which saves up to 2 bits per element of space as compared to modern filter designs while adapting efficiently to false positives as they arise. To do this, VHFs build on recent innovations in compact data structures for networking applications[28] and takes advantage of the increasing availability of hardware acceleration for parallelizable algorithms.

We begin by providing background on the data structure referenced above. Then, we describe VHFs in detail before performing a theoretical analysis of their FPR and computational complexity. Lastly, we present an initial set of experiments plotting their FPR and size.

3.1 SetSep: The Precursor to VHFs

A recent paper in networking proposed a data structure called SetSep which stores indices into a set of hash functions in order to separate sets[28]. This presents an interesting third way forward in the creation of filters which is neither reliant on bit arrays nor hash fingerprints.

SetSep works by first splitting the elements whose set membership must be stored into small sets via a combination of hashing and sorting buckets. Through this process they achieve fairly low variance in the size of the eventual subsets. At that point, they use a set q of hash functions which are chosen so that each one makes up a bit in a q-length string which encodes the set of the element.

For example, suppose that there are two elements x_1 and x_2, which are in sets 1 and 2 respectively, in a particular subset and that q equals 3. The first hash function would send both elements to an even number. The second would send x_1 to an even number and x_2 to an odd number, and the third would send x_1 to an odd number and x_2 to an even number. Therefore, the string generated by evaluating the three hash functions on x_1 is 001, and the one for x_2 is 010. This would allow the encoding of 2^3 potential set memberships.

Important to the use of this design for filtering is what happens to elements which were not entered into the data structure. In the initial step, they are approximately randomly sent to one of the sets of hash functions. Then, those functions are evaluated, and each hash function has an approximately equal probability of returning an even or odd number. This results in the element being sent to a uniformly random set within the 2^q available options. Suppose, however, that there were only $2^q - 1$ valid sets. This would mean that any element sent to the unclaimed number would be known with certainty to not exist in any set. This insight is the basis of how SetSep is extended to create a traditional filter.

3.2 Algorithm Overview

Variable Hash Filters are constructed very similarly to the SetSep construction. There are two phases 1) the elements are separated into smaller sets of 10-20 2) a set of k hash functions are chosen such that they all send every element to an odd number.

Phase 1 In the first phase, Variable Hash Filters diverge from SetSep by instead using a scheme more similar to perfect hashing methods to evenly spread the elements into slots. The elements are first hashed using a single hash function to a sector. At this stage, a large number of hash functions are tested and the best is chosen based on which most evenly spreads the elements within the sector to buckets. The elements within a bucket then undergo the same process where a hash function is chosen for the bucket which most evenly distributes the elements to the slots which contain the final sets of hash functions to be evaluated. Through this process, we are attempting to minimize the variance in the number of elements-per-slot which has a large impact on the performance of the filter.

By limiting the number of bits available to index the hash functions, the impact on the overall size of the filter is minimized. Further, the tables holding the sector-level and bucket-level hash indices will be very small, on the order of 1/1000 and 1/100 the size of the filter, and likely fit in the higher levels of the cache hierarchy. This reduces the number of main memory accesses necessary to evaluate the filter.

Phase 2 In the second phase, we consider what happens at the slot-level. All elements in the slot are viewed as being in the same set which is denoted by a value of all 1s. Therefore, each hash function in the set of k hash functions is chosen to send every element in the slot to an odd number. However, unlike SetSep, we bound the number of hash functions which can be tested in order to conserve space. In particular, we set the number of bits used to index each hash function to r which allows the indexing of 2^r hash functions. This will usually be set between 2 and 4 larger than the number of elements that land in a slot on average.

Further, a the zero index in the hash function space is reserved for the function which sends every element to 1. Therefore, if a hash function cannot be found which sends every element of a slot to zero, then that hash index is set to zero in order to ensure that no false negatives are generated.

3.3 Theoretical Analysis

3.3.1 Notation

Before further analysis of this data structure, we introduce some useful notation. The number of elements which will be entered into the filter is n, and the number of slots in the filter is m. As above, the number of bits per hash function is r, and the number of hash functions is k. The number of slots per bucket is s, and the number of buckets per sector is b. The false positive rate will be denoted α, and the size of the filter will be denoted M. Lastly, we denote the ith indexed hash function by H_i, and introduce the function $SAT(H_i)$ which evaluates to true if that hash function sends every element in its slot to an odd number and false otherwise.

3.3.2 False Positive Rate Under Perfect Spreading

First, we assume that the hashing of elements to slots is perfect, i.e. if there are $12m$ elements then every slot receives 12 elements. We also set $r = \frac{n}{m} + 3$. To calculate the false positive rate, we need to determine the probability that a given slot is unable to find a satisfactory hash function in the space of 2^r hash functions. Each hash function, H_i has the same probability of being satisfactory as $\frac{n}{m}$ fair coin flips being heads. Therefore,

$$P(SAT(H_i)) = 2^{-\frac{n}{m}}$$

This means that the probability that all hash functions are unsatisfactory is,

$$P(\bigcup_{i=0}^{2^r} SAT(H_i)) = (1 - 2^{-\frac{n}{m}})^{2^r}$$

When q is set as above, this expression becomes approximately .0003 for values of $\frac{n}{m}$ between 6 and 18, showing that it is highly likely that a good hash function can be found.

Now, let i be the index of the slot that a negative element x is hashed to, then the probability that it is a false positive becomes,

$$\alpha = \frac{1}{2} * (1 - P(\bigcup_{i=0}^{2^r} SAT(H_i))) + P(\bigcup_{i=0}^{2^r} SAT(H_i)) \approx .51$$

Further, the space required to do this is $\frac{r}{\frac{n}{m}}$ bits per element. If the process is repeated independently k times, the false positive rate is approximately,

$$\alpha = (.51)^k$$

Inverting this, we get,

$$k = -(\frac{1}{\log_2(.51)})\log_2(\alpha) \approx 1.02 * \log_2(\alpha)$$

Lastly, we can use the relationship between k and size to turn this into a relationship between bits per element and FPR,

$$\frac{M}{n} = -\frac{r}{\frac{n}{m}} * (\frac{1}{\log_2(.51)})\log_2(\alpha)$$

Choosing to put 14 elements in each slot, we get the following where c is the bits per element required for partitioning in phase 1,

$$\frac{M}{n} \approx -1.2 * \log_2(FPR) + c$$

For context, Bloom Filters use $-1.44\log_2(FPR)$ bits per element to achieve a given FPR.

3.3.3 The Space-Computation Trade-Off

Looking at the previous section, one equation stands out as representing a fundamental aspect of this data structure.

$$\frac{M}{n} = -\frac{r}{\frac{n}{m}} * (\frac{1}{\log_2(.51)})\log_2(\alpha)$$

The fractional coefficient $\frac{r}{\frac{n}{m}}$ represents the bits used for each hash divided by the average number of elements per slot. Importantly, the probability of finding a hash is governed not by this fraction but rather by the difference between r and $\frac{n}{m}$, i.e. if r is equal to $\frac{n}{m}$ plus 3 then the chance of finding a satisfactory hash will be high. This implies that the space efficiency of the data structure can be increased by increasing $\frac{n}{m}$ while keeping that difference constant. Doing this does increase space efficiency, however it increases the computational time exponentially. As $\frac{n}{m}$ goes to infinity, this produces a data structure that approaches the optimal size of $-\log 2(\alpha)$.

Therefore, upfront computational expenditure during construction can be traded for a more space efficient data structure.

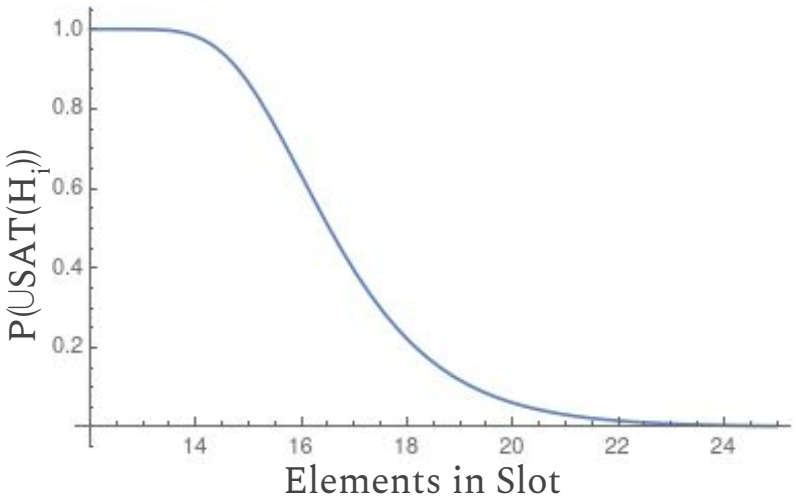

Figure 8: The probability of finding a hash function given $r = 16$ degrades smoothly as the number of elements increases.

3.3.4 Probability of Finding a Hash Under Imperfect Spreading

As shown in figure 8, the probability of finding a satisfactory hash function decreases rapidly as the number of elements in the slot grows. However, by setting the number of hash bits to several more than the average number of elements per slot, we sacrifice some space while gaining some extra room for the elements per slot to vary somewhat.

Further, this curve shows that when a slot is somewhat overfull, e.g. 5 elements more than the mean of 13, there is still a high probability that at least one of the k attempts to find a hash function succeeds. This would reduce the number of induced false positives from this slot by half, a significant gain. For context, SetSep's method for partitioning the set into slots resulted in a maximum of 21 elements in a slot when performed with an average of 16 elements per slot over a set of 16 million elements.

3.3.5 Runtime Analysis

Construction Runtime In the context of this problem, computation will first be described in the number of hash functions are evaluated which we label C_H. In addition, we limit the number of bits available to index hash functions for the sector and bucket spreading functions at r_s and r_b bits respectively. Therefore, the number of computations per element in phase one of the algorithm is $2^{r_s} + 2^{r_b}$ because every index-able hash function is tested.

The number of computations in the second phase of construction is more variable. This is because once a satisfactory hash function is found the rest of the index-able hash functions are not attempted. Therefore, if a slot has fewer elements, it will likely find a hash function without testing nearly as many

functions and require less computation. To upper bound the computation, we can assume that each slot tests every hash function. Given this, at most $k * 2^r$ hash functions are computed per element in the second phase. From this, we get the final expression,

$$C_H = n(2^{r_s} + 2^{r_b} + k * 2^r)$$

This shows that the construction time remains linear in the size of the data which is crucial for performance given the size of data handled by filters. However, it also shows that the number of bits allocated to indexing hash functions at all levels has a very large impact on the eventual construction time. Lastly, the computation increases logarithmically with respect to the false positive rate.

While this algorithm is computationally very expensive, it is fortunately very efficient in data movement. In fact, each stage of the construction is performed on subsets of the data which can fit in the processor cache. First, calculating the sector hashes requires repeatedly running sequential computations over a vector whose length is equal to the number of elements in the sector, generally on the order of 2000. Supposing a processor cache of at least one Mb and elements of on average at most 500 bytes, this implies that all sector-level computation would take place within the processor cache. Calculating the bucket hashes does the same on a subset of those elements and is therefore even more cache efficient. Lastly, selecting hashes within a slot iterates over 10-20 elements repeatedly, making the process nearly register-resident. All three stages together require retrieving the dataset from main memory four times under reasonable constraints.

Query Runtime When querying an element, three hash functions are evaluated in phase one in order to determine the sector, bucket, and slot. Then, k further hash functions are evaluated to determine result of the slot's computation. Therefore, the query computation, which we denote Q_H, can be expressed as,

$$Q_H = k + 3$$

This is logarithmic in the false positive rate similarly to traditional bloom filters. However, the more relevant analysis once again lies in the data accesses required for a query. After a full cache flush, at least three main memory data accesses are necessary to evaluate the filter: the array of sector hashes, the array of bucket hashes, and the array of slot hashes. One caveat is that, depending on the values of k and r, a slot may not fit into a single cache line and therefore may incur one random and one sequential memory access.

When elements are being queried in bulk, the array of sector hashes and bucket hashes will be present in the processor cache depending on the size of the filter and the size of the cache. The number of sector hashes is $\approx n/2000$, and each hash takes up 4 bytes. Further, the number of bucket hashes is $\approx n/100$, and they are the same size. Therefore, the sizes of the array of sector hashes relative to the filter as a whole are,

$$\frac{n/2000}{n/2000+n/100+k*n/13} = \frac{1}{21+k*153}$$

For a standard false positive rate of 1%, this results in,

$$\frac{n/2000}{n/2000+n/100+k*n/13} \approx .001$$

Because of this, the array of sector hashes will fit in the cache if the size of the filter is less than a thousand times the size of the cache. The array of bucket hashes, on the other hand, takes up the following percentage of the filter.

$$\frac{n/100}{n/2000+n/100+k*n/13} \approx .02$$

Therefore, if the filter is less than 51 times the size of the cache, then both the arrays will fit in the processor cache. In this case, there is only one random memory access incurred by this filter for each query.

3.4 Parallelization and GPU Acceleration

The construction of VHFs lends itself to parallelization because of its structure as a tree of independent sub-problems. Each level of construction consists of finding a set of hash functions which operate on disjoint subsets of the data. Therefore, each hash function being selected can be handled by a different thread without causing any data races.

In the first phase, parallelization would take place by simply allowing one thread to handle finding the hash for each sector, then for each bucket. Depending on the size of the filter, this is likely already a point which would benefit from GPU acceleration. However, if the number of sectors is relatively small, e.g. in a filter with less than 50,000 elements, it may be preferable to use the fewer but faster threads on a cpu for selecting the sector hashes.

In the second phase, each thread of the GPU performs the full computation for one slot at a time until all slots have found hash functions or tested every possible hash function.

By offloading this computation onto a GPU, it is likely that order of magnitude improvements can be achieved. Further, GPU speeds are continuing to increase at exponential rates while CPUs are beginning to plateau which makes a GPU-native filter attractive as data sizes continue to grow[15].

3.5 Adaptivity & False Positives

One exciting element of these filters is their ability to remove false positives dynamically and cheaply. This is done in the following way. During construction, the hashes of the elements in each slot are stored on disk for latter access. When a false positive is identified, the elements in the slot that it was hashed to are pulled off of disk. Then the filter attempts to find a new hash for one of the hashes in the slot which satisfies the property of sending all the true positives to an odd number but additionally sends the false positive to an even number.

To ensure that false positives are not reintroduced cyclically, hash functions are only replaced with functions that have a higher index. Notably, while a slot holds k hashes to achieve an FPR of 2^{-k}, only a single hash would have to be updated to fix a false positive. This means that fixing a false negative has $\frac{1}{k}$ the cost of including a new positive element.

For comparison, the cost of nullifying a false positive in the Stacked Filters design is slightly higher than the cost of including a positive element because it requires storage in the second layer as well as some fraction of the storage costs from the latter layers. Given that a VHF could account for k times as many false positives as a stacked filter, the relative reduction in storage costs versus traditional filters could be significantly greater.

Further, whereas Adaptive Stacked Filters require a "cold start" where they underperform traditional filters, VHFs would be able to smoothly incorporate false positive information. This results in a more predictable and smoother increase in overall performance over the lifespan of the filter.

3.6 Experimental Results

A simple evaluation of VHFs is presented here. FPR and the probability of a hash not being found is explored, while an evaluation of the computational efficiency is left for future work.

3.6.1 False Positive Rate

The experimental false positive rate of Variable Hash Filters matches the optimal predicted FPR fairly closely. The theoretical FPR shown in figure 9 is calculated on the assumption that every slot contains the average number of elements per slot. They diverge because some slots contain too many elements

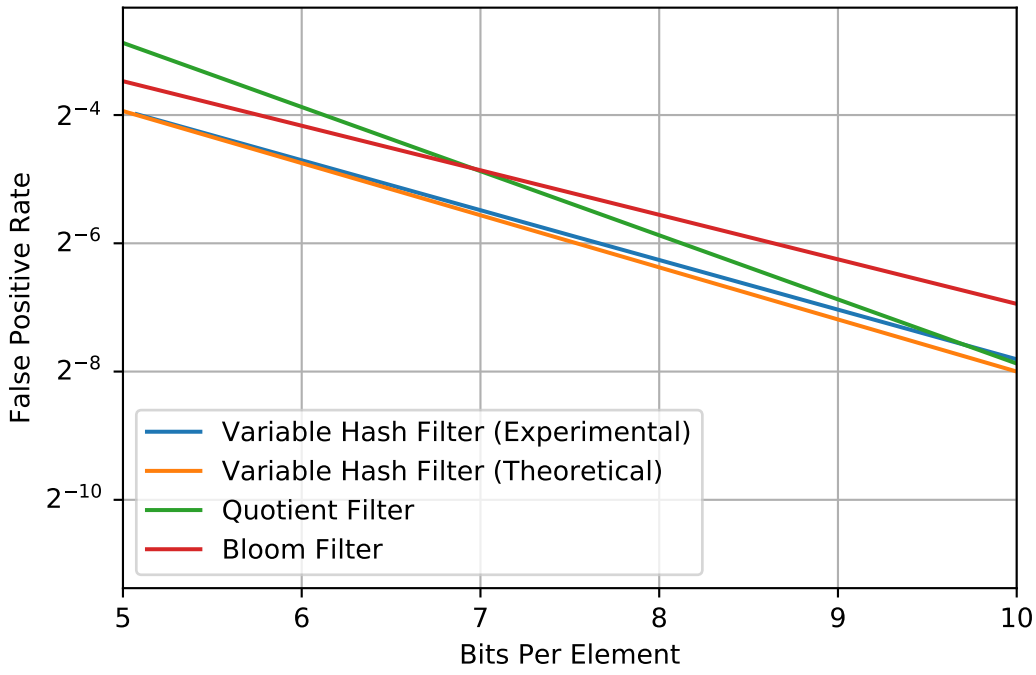

Figure 9: The false positive rate decreases exponentially in the number of bits per element, and Variable Hash Filters use fewer bits than Bloom Filters or Quotient Filters.

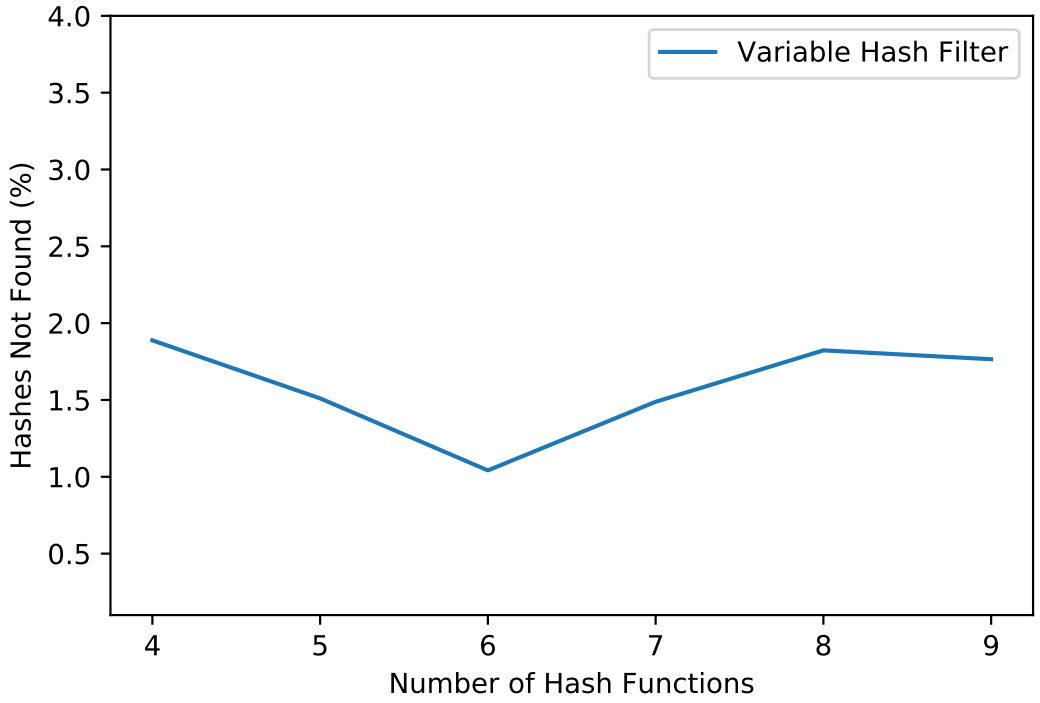

Figure 10: The proportion of the hashes within slots that are set to the trivial hash is constant at 2%.

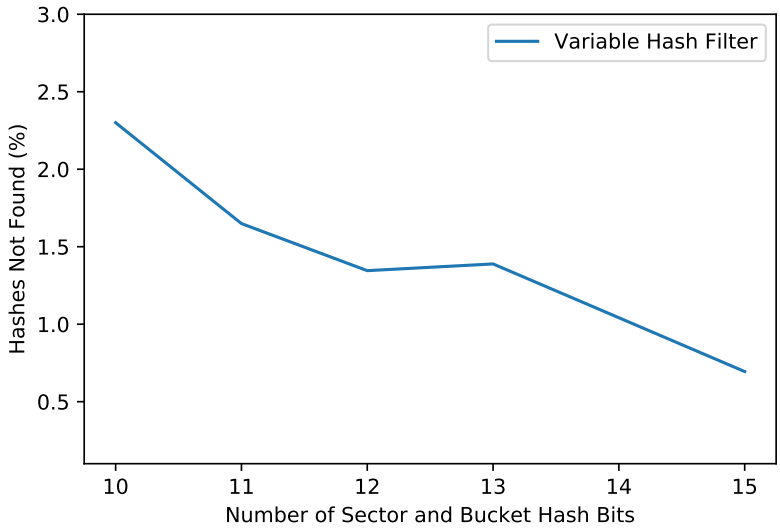

Figure 11: The probability of finding a hash function given $r = 16$ improves slowly as more computation and space is provided to finding the sector and bucket hashes.

even after the spreading in phase one which results in less than all of the hash functions being found. When negative elements are sent to a slot where a valid index-able hash hasn't been found, that hash automatically counts as a 1. If they are sent to a slot with most of its hashes not found, then they are very likely to generate a false positive. This effect is exaggerated at lower false positive rates because having even one slot with none or only a single valid hash function results in a significant portion of negative elements being directly accepted from that slot relative the false positive rate as a whole.

As seen in 9, the false positive rate achieved by VHFs is lower than the theoretical false positive rates of either Counting Quotient Filters or Bloom Filters for sizes lower than 10 bits per element.

3.6.2 Probability of Not Finding a Hash

10 shows that as the number of hash functions per slot grows, the proportion of hashes which are not found stays fairly constant. The effectiveness of the spread capability is determined by the number of bits that is provided to index the sector and bucket hashes. In this experiment, those values were both set to 12. However, as shown in figure 11, if more bits are given to selecting the hashes at the sector and bucket level, e.g. 16 bits for each, this percentage drops to .5%, and the FPR drops correspondingly. However, doing this significantly increases the computational cost.

4 Conclusion

In this thesis, two novel filtering methods were presented to overcome a theoretical barrier in space efficiency. The first, Stacked Filters, takes advantage of the large body of filtering research by adapting traditional filters to handle false positives. The second, Variable Hash Filters, presents a new filter design which natively handles false positives while simultaneously providing a lower false positive rate than existing filters for low bits-per-element.

These methods can be used to improve the overall efficiency of memory-constrained systems by producing a lower FPR and reducing the number of unnecessary operations performed. In a broader sense, this work builds on a research direction started by Complement Filters, Yes-No filters, and Adaptive Cuckoo Filters, demonstrating that the nullification of false positives can provide significant benefits even in limited-information environments and without sacrificing computational efficiency.

References

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

[2] Top 10 million websites: Openpagerank.

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

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

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

[6] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 59–65. ACM, 1978.

[7] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael 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.

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

[9] Steven G Johnson. Nlopt.

[10] Kota Kanemura, Kentaroh Toyoda, and Tomoaki Ohtsuki. Design of privacy-preserving mobile bitcoin client based on γ-deniability enabled bloom filter. In *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–6. IEEE, 2017.

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

[12] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[13] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter Boncz. Performance-optimal filtering: Bloom overtakes cuckoo at high throughput. *Proceedings of the VLDB Endowment*, 12(5):502–515, 2019.

[14] Hyesook Lim, Jungwon Lee, and Changhoon Yim. Complement bloom filter for identifying true positiveness of a bloom filter. *IEEE communications letters*, 19(11):1905–1908, 2015.

[15] Chien-Ping Lu. K3 moore's law in the era of gpu computing. In *Proceedings of 2010 International Symposium on VLSI Design, Automation and Test*, pages 5–5. IEEE, 2010.

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

[17] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 36–47. SIAM, 2018.

[18] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob 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.

[19] Geoff Pike and Jyrki Alakuijala. Cityhash, Jan 2011. https://github.com/google/cityhash/blob/master/src/city.h.

[20] 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.

[21] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *Journal of Experimental Algorithmics (JEA)*, 14:4, 2009.

[22] Sukriti Ramesh, Odysseas Papapetrou, and Wolf Siberski. Optimizing distributed joins with bloom filters. In *International Conference on Distributed Computing and Internet Technology*, pages 145–156. Springer, 2008.

[23] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 21–40. Springer-Verlag New York, Inc., 2003.

[24] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[25] T.p. 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.

[26] Karan Singhal and Philip Weiss. deepbloom2018. https://github.com/karan1149/DeepBloom.

[27] Ding Wang, Haibo Cheng, Ping Wang, Xinyi Huang, and Gaopeng Jian. Zipf's law in passwords. *IEEE Transactions on Information Forensics and Security*, 12(11):2776–2791, 2017.

[28] Dong Zhou, Bin Fan, Hyeontaek Lim, David G Andersen, Michael Kaminsky, Michael Mitzenmacher, Ren Wang, and Ajaypal Singh. Scaling up clustered network appliances with scalebricks. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 241–254. ACM, 2015.

[29] George Kingsley Zipf. Selected studies of the principle of relative frequency in language. 1932.