Stacked Filters: Learning to Filter by Structure

Kyle Deeds*
Harvard University
kdeeds@harvard.edu

Brian Hentschel*
Harvard University
bhentschel@g.harvard.edu

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

ABSTRACT

We present Stacked Filters, a new probabilistic filter which is fast and robust similar to query-agnostic filters (such as Bloom and Cuckoo filters), and at the same time brings low false positive rates and sizes similar to classifier-based filters (such as Learned Filters). The core idea is that Stacked Filters incorporate workload knowledge about frequently queried non-existing values. Instead of learning, they structurally incorporate that knowledge using hashing and several sequenced filter layers, indexing both data and frequent negatives. Stacked Filters can also gather workload knowledge on-the-fly and adaptively build the filter. We show experimentally that for a given memory budget, Stacked Filters achieve end-to-end query throughput up to 130x better than the best alternative for a workload, either query-agnostic or classifier-based filters, and depending on where data is (SSD or HDD).

PVLDB Reference Format:
Kyle Deeds, Brian Hentschel, Stratos Idreos. Stacked Filters: Learning to Filter by Structure. PVLDB, 14(4): 600 - 612, 2021.
doi:10.14778/3436905.3436919

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://bitbucket.org/HarvardDASlab/stackedfilters.

1 LEARNING FILTERS BY STRUCTURE

Filters are Everywhere. The storage and retrieval of values in a data set is one of the most fundamental operations in computer science. For large data sets, the raw data is usually stored over a slow medium (e.g., on disk) or distributed across the nodes of a network. Because of this, it is critical for performance to limit accesses to the full data set. That is, applications should be able to avoid accessing slow disk or remote nodes when querying for values that are not present in the data. This is the exact utility of approximate membership query (AMQ) structures, also referred to as filters. Filters have tunably small sizes, so that they fit in memory, and provide probabilistic answers to whether a queried value exists in the data set with no false negatives and a limited number of false positives. For all filters, the probability of returning a false positive, known as the false positive rate (FPR), and the space used are competing goals. Filters are used in a large number of diverse applications such as web indexing [24], web caching [21], prefix matching [18], deduping data [17], DNA classification [42], detecting flooding attacks [22], cryptocurrency transaction verification [23], distributed joins [38, 40], and LSM-tree based key-value stores [14], amongst many others [43].

Robust Query-Agnostic Designs. Query-agnostic filters utilize only the data set during construction. For example, a Bloom Filter [5] starts with an array of bits set to 0 and using multiple hash functions per value, hashes each value to various positions, setting those bits to 1. A query for a value x probes the filter by hashing x using the same hash functions and returns positive if all positions x hashes to are set to 1. When querying a value that exists in the data set, these bits were set to 1 during construction and so there are no false negatives; however, when querying a value not in the data set, a false positive can occur if the value is hashed entirely to positions which were set to 1 during construction.

Other query-agnostic filters such as Cuckoo [20], Quotient [34], and Xor [25] filters work similarly to Bloom Filters. These filters generally work by hashing data values and storing the resulting fingerprints in a hash table losslessly (for instance, using cuckoo hashing for Cuckoo Filters or linear probing for Quotient filters).

If the hash functions are truly random, then every query-able value not in the data set has the same false positive chance. This makes query-agnostic filters robust, as they have the same expected performance across all workloads, and easy to deploy, as they require no workload knowledge. However, at the same time, this limits the performance of query-agnostic filters, as they are required to work for any query distribution. Additionally, the possibility for further improvements in the trade-off between space and false positive rate are limited, as current query-agnostic filters are close to their theoretical lower bound in size [7, 11].

Learning from Queries for Reduced Filter Size. Classifier based filters such as Weighted Bloom Filters [8, 45], Ada-BF [13], and Learned Bloom Filters [28, 39] utilize workload knowledge, making it possible to move beyond the theoretical limits of query-agnostic filters. Such filters need as input a sample of past queries and using that they train a classifier to model how likely every possible value is to 1) be queried, and 2) exist in the data set. The classifier is then used in one of two ways.

In the first [28, 39], it acts as a module which accepts values that have a high weighted probability of being in the data. It cannot reject values as the stochasticity of the classifier might cause false negatives. Thus, a query-agnostic filter is also built using the (few) values in the data set for which the classifier returns a false negative. Queries are first evaluated by the classifier, and if rejected, then probe the query-agnostic filter. In the second [8, 13, 45], the classifier uses the weighted probability of being in the set to control the number of hash functions used by a Bloom filter for each value. For values with a high likelihood of being in the set, few hash functions are used, setting fewer bits in the filter but also checking fewer bits, and therefore providing fewer chances to catch a false positive. For values not likely to be in the set, this is reversed.

*Both authors contributed equally to the paper

Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436919

Classifier based filters can reduce the required memory to achieve a given false positive rate when keys and non-keys have clear semantic differences. URL Blacklisting is such a use case [28], wherein browsers such as Google Chrome maintain a list of dangerous websites and alert users before visiting one. Browsers store a filter containing the dangerous websites at the client. For each web request, the client checks the filter; if the filter rejects the query, the website is safe and can be visited. If the filter accepts the query, an expensive full check to a remote list of dangerous websites is done. Because there are no false negatives, every dangerous website is caught, and that there are only a few false positives means most safe websites do not need to perform extra work.

However, classifier based filters offer a host of new problems. First, the classifier has to be accurate, which can be hard: for some workloads the data value and its probability of existing in the data set are loosely correlated. For other types of data that appear in practice such as hashed IDs, there is no correlation. Additionally, even when data patterns exist, often data such as textual keys have complex decision boundaries which require complex models such as neural networks for accurate classification. As a result, the computational expense of the classifier is often orders of magnitude more expensive than hashing. Finally, the classifier is trained on a specific sample workload, and thus if the workload shifts, we need to go through the expensive process of gathering sample queries and retraining the classifier to maintain good performance.

Stacked Filters: Encapsulating workload information structurally. We introduce a new class of filters, Stacked Filters, which structurally encapsulate knowledge about frequently queried non-existing values. The key intuition is as follows: for non-keys which are queried often, find a structural way to run them through multiple filter checks. Stacked Filters achieve that through several layers of query-agnostic filters which alternate between representing values in the data and frequently queried non-existing values.

All frequently queried non-existing values need to pass multiple membership checks to be false positives, and so they incur exponentially smaller false positive rates. At the same time, each additional layer is exponentially smaller in size, and thus the total size of a Stacked Filter is comparable to the first filter in its stack. A similar pattern holds for computational costs. Both size and computational costs rise like a geometric series with the number of layers, and thus have values close to that of a single filter, while an entire set of non-existing values has their FPR decrease arbitrarily close to zero. The overall result is that for workloads with any frequently queried non-existing values, Stacked Filters provide a superior tradeoff between false positive rate, filter size, and computation than either of classifier-based filters or query-agnostic filters.

While the idea of checking frequently queried non-existing values multiple times is intuitive, it comes with significant challenges. How many layers are needed for good filter performance? How should the memory budget be spread across the layers? How much workload knowledge is enough for good filter performance? Can we build Stacked Filters without any workload knowledge?

Contributions. The contributions of this paper are as follows:

- *Data Structure Formalization*: We introduce a new way to design workload-aware filters as multi-layer filter structures which index both positives and frequent negatives.
- *Generalization*: We show that Stacked Filters work for all query-agnostic filters including Bloom, Cuckoo, and Quotient Filters.
- *Better trade-off of FPR and size*: We derive the metric equations of Stacked Filters for size, computation, and false positive rate. Using these equations, we provide theoretical results showing Stacked Filters are strictly better in terms of FPR vs. size than query-agnostic filters on the majority of workloads, and quantify the expected benefit.
- *Optimization*: We show that the optimization problem of tuning the number of layers and layer sizes is non-convex. Still, we provide ϵ-approximation algorithms, running in the order of milliseconds, which automatically tune the number of layers and the individual sizes of each layer so that performance is arbitrarily close to optimal.
- *Adaptivity*: We show that the benefits of Stacked Filters can be extended to Stacked Filters built adaptively. Here, Stacked Filters start with a rough knowledge of how skewed a workload is, but not which values are frequently queried, and build their structure incrementally during normal query execution.
- *Experiments*: Using URL blacklisting, a networking benchmark, and synthetic experiments we show that Stacked Filters 1) provide improvements in FPR of up to 100× over the best alternative query-agnostic or classifier-based filter for the same memory budget, while retaining good robustness properties, 2) provide a superior tradeoff between false positive rate, size, and computation than all other filters, resulting in up to 130× better end-to-end query throughput than the best alternative, and 3) for scenarios where learning is not easy, Stacked Filters can still utilize workload knowledge and offer throughput up to 1000× better than classifier-based filters.

2 NOTATION AND METRICS

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

Notation. Let U be the universe of possible data values, such as the domain of strings or integers. Let P be a data set, which we will refer to as the positive set, and let $N = U - P$ be the set of negative values. From now on we will refer to data values as well as to queried values as elements, which is the traditional terminology in the filters literature. We will denote filter structures by F, which we treat as a function from $U \rightarrow \{0, 1\}$, and we say that x is accepted by F if $F(x) = 1$ and that x is rejected by F if $F(x) = 0$.

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

Expected False Positive Bound (EFPB). A traditional guarantee for a filter F is to bound $E_F[\mathbb{P}(F(x) = 1 | x \notin P)]$ for any x chosen independently of the creation of F. We call this bound the *expected false positive bound.*

Expected False Positive Rate (EFPR). Given a distribution D over U which captures the query probabilities for elements in U, the *expected false positive rate* is $E_{x \sim D}[E_{F|D}[\mathbb{P}(F(x) = 1 | x \notin P)]]$.

Notation	Definition
P	Set of all positive elements
N	Set of all negative elements
N_f	Negatives used to construct a Stacked Filter
N_i	The complement of N_f, i.e. $N \setminus N_f$
s	Size of a filter in bits/element
$c_{i,S}$	Cost of inserting an element from set S
$c_{q,S}$	Cost of querying a filter for an element in S
Metric	**Definition**
EFPR	Expected false positive rate of a filter structure given a specific query distribution
EFPB	Upper bound on the EFPR of a filter structure for queries chosen independently of the filter

Table 1: Notation used throughout the paper

Optimization via Expected False Positive Rate. Query throughput most directly relies on the EFPR and since the EFPR depends on the query distribution D, the goal of workload-aware filters is to capture and utilize D to improve the EFPR. Namely, for $x \in N$ with higher chance of being queried, workload-aware filters should lower the probability that x is a false positive.

Robustness via Bounding False Positive Probability. Optimizing the EFPR helps system throughput but brings concerns about workload shift. Query-agnostic filters can act as a safeguard against such a shift. To see this, note that F and D are independent by assumption and so for a query-agnostic filter with FPR ϵ,

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

regardless of what D is. Thus, filters which provide an expected false positive bound provide an upper bound on the expected false positive rate for any workload D chosen independently of the filter.

Memory - False Positive Tradeoff. For all filter structures, their EFPR and EFPB can be made arbitrarily close to 0 with enough memory, and there exists a tradeoff between the memory required and the false positive rate provided. Thus for purposes of comparison, we always report EFPR and EFPB with respect to a space budget. Space budgets in practice tend to be between 6 and 14 bits per element, and are significantly smaller than the elements they represent (which can be anywhere from 4 bytes to several megabytes).

For query-agnostic filters, the EFPR and the EFPB are equal, and is just called the false positive rate. Additionally, for all query-agnostic filters, the false positive rate α and size in bits per element s are 1-1 functions of each other. When going from one to the other, we denote the quantities by $s(\alpha)$ and $\alpha(s)$, which denote the size for a given FPR and the FPR for a given size respectively.

Computational Performance. Filter structures desire computational performance much faster than the cost to access the data they protect. We denote the cost to insert into a filter an element of set S by $c_{i,S}$. We also denote the cost to query for an element of set S by $c_{q,S}$. If no set is denoted, then $S = U$.

3 STACKED FILTERS

The traditional view of filters is that they are built on a set S, and return no false negatives for S. An alternative view of a filter is that it returns that an element is certainly in $\overline{S}$ (the complement of S), or that an element's set membership is unknown. In Stacked Filters, we use this way of thinking about filters, with S for different layers of the stack alternating between subsets of P and subsets of N, to iteratively prune the set of elements in U whose set membership is undecided.

Stacked Filters by Example. We start with an example of a 3 layer Stacked Filter using Figure 1. The filter is given the data set P and a set of frequently queried negatives N_f. The first filter in the stack, L_1, is constructed using P similarly to a traditional filter except with fewer bits per element so as to reserve space for subsequent 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. If the Stacked Filter ended here after a single filter, as is the case for all query-agnostic filters, all undecided elements would be accepted by the Stacked Filter.

Instead, Stacked Filters construction continues by probing L_1 for each element in N_f. Using all elements of N_f accepted by L_1, and which therefore normally would become false positives, Stacked Filters build a second layer with another query-agnostic filter. During a query, values which are still undecided after L_1 are passed to L_2. If L_2 rejects the value, the value is definitely in $\overline{N_f}$, which includes both P and $N \setminus N_f$, which we denote by N_i and call the infrequently queried negative set. Since $P \cup N_i$ contains both positives and negatives, the overall Stacked Filter accepts all the rejected elements of L_2 in order to maintain a zero false negative rate. If the element is instead accepted by L_2, then its set membership is still undecided and so it continues down the stack.

Construction then continues by querying L_2 for all elements in P and building a third layer with a query-agnostic filter. This layer uses as input all elements of P whose set membership is undecided after querying L_2. At query time, L_3 performs the same operations as L_1; elements rejected by L_3 are certainly in $\overline{P} = N$ and so are rejected by the Stacked Filter.

Workload-aware Design. L_2 and L_3 are how Stacked Filters structurally incorporate workload knowledge. They collaborate to filter out frequent negatives to minimize FPR. All frequent negatives that are false positives on L_1 reach L_3 since they are in the construction set for L_2, and so such frequent negatives need to pass an extra membership check to be false positives for the full Stacked Filter. To make deeper Stacked Filters, and thus perform more checks on frequently queried negatives, we recursively perform this process, adding more paired sets of layers.

An intuitive understanding of the effectiveness of Stacked Filters comes from the interplay between the extra size of additional layers vs. their benefit for FPR. Compare the simple 3 layer example above, assuming each layer has an FPR of 0.01, with a single traditional filter using FPR 0.01. If $|N_f| = |P|$, then L_2 and L_3 are on average $1/100$ the size of L_1, and the Stacked Filter has 2% higher space costs than a traditional filter. But for every element in N_f, the extra membership check makes their *FPR* a full 100× lower; thus if N_f contains any significant portion of the query distribution, the Stacked Filter has a much lower EFPR.

General Stacked Filters Construction. Algorithm 1 shows the full construction algorithm. Like both query-agnostic and classifier based filters, Stacked Filters need two inputs 1) the data set, and 2) a constraint (memory budget or a desired maximum EFPR).

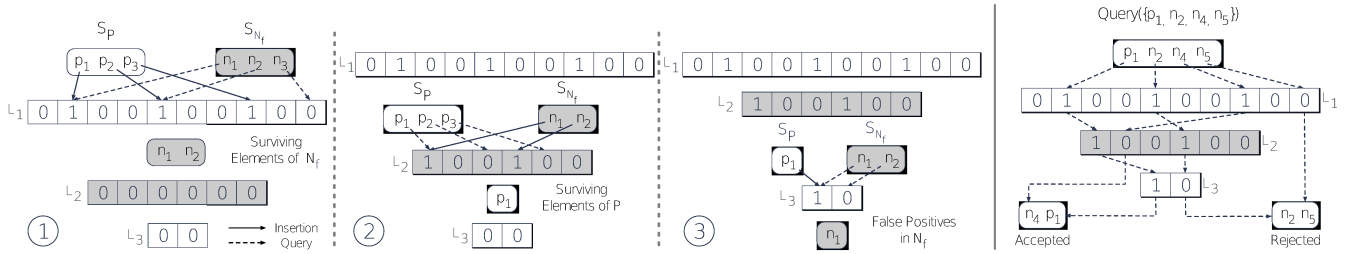


Figure 1: Stacked Filters are built in layer order, with each layer containing either positives or negatives. The set of elements to be encoded at each layer decreases as construction progresses down the stack. For queries, the layers are queried in order and the element is accepted or rejected based on whether the first layer to return not present is negative or positive.

Algorithm 1 ConstructStackedFilter(S_P, W, C)

Input: S_P, W, C: data set, workload, constraint
1: S_{N_f}, $\{\alpha_i\}$ = *OptimizeSF*(W,C) // In Sec. 5
2: // Construct the layers in the filter sequentially.
3: **for** $i = 1$ to T_L **do**
4: $S_r = \{\}$
5: **if** i mod 2 = 1 **then** // layer positive
6: construct L_i w/ space $s(\alpha_i) * |S_p|$
7: **for** $x_p \in S_P$ **do**
8: L_i.INSERT(x_p)
9: **for** $x_n \in S_{N_f}$ **do**
10: **if** $L_i(x_n) = 1$ **then**
11: $S_r = S_r \cup \{x_n\}$
12: $S_{N_f} = S_r$
13: **else** // layer negative
14: allocate L_i w/ space $s(\alpha_i) * |S_{N_f}|$
15: **for** $x_n \in S_{N_f}$ **do**
16: L_i.INSERT(x_n)
17: **for** $x_p \in S_p$ **do**
18: **if** $L_i(x_p) = 1$ **then**
19: $S_r = S_r \cup \{x_p\}$
20: $S_P = S_r$
21: return F

Stacked Filters also need workload knowledge similarly to classifier-based filters. Later on we describe in detail how much knowledge is needed, and how to gather it (even adaptively). For now, we denote workload knowledge abstractly as W, and feed this into an optimization algorithm which returns 1) a set of frequently queried negatives, denoted by N_f, and 2) the number of layers T_L and false positive rate of each layer $\alpha_1, \ldots, \alpha_{T_L}$.

Querying a Stacked Filter. Algorithm 2 formalizes the description given in the example for querying a Stacked Filter. Querying for an element x starts with the first layer and goes through the layers in ascending order. At every layer, if the element is accepted by the layer, it continues to the next layer. If an element is rejected by a layer containing positive elements (called a positive layer), it is rejected by the Stacked Filter. Conversely, if an element is rejected by a layer containing negative elements (negative layer), it is accepted by the Stacked Filter. 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 this algorithm is correct, and explain how the algorithm differently affects positives, frequently queried negatives, and infrequently queried negatives.

Querying a Positive. Stacked Filters maintain the crucial property of having no false negatives. During construction, every positive element x_p is added into each positive layer until it either hits the end of the stack, or is rejected by a negative layer. Querying the Stacked Filter for x_p follows the same path. Either x_p makes it to the end of the stack and is accepted by the Stacked Filter, or it is rejected by some negative layer and thus accepted by the Stacked Filter. Figure 1 shows how this process works for the positive element p_1.

Algorithm 2 Query(x)

Input: 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 mod 2 = 1 **then**
5: **return** reject // Layer positive, reject x
6: **else**
7: **return** accept // Layer negative, accept x
8: **return** accept // No layer rejected, accept x

Querying a Negative. For an infrequently queried negative element $x_i \in N \backslash N_f$, it is in neither P nor N_f and so has high likelihood of being rejected by both positive and negative layers. As a result, the majority of infrequently queried negatives are rejected at L_1, and the majority of false positives occur from elements rejected at L_2. Frequently queried negatives are a false positive for the Stacked Filter only if they are a false positive for each positive layer. This drives the FPR of frequently queried negatives towards 0, as their false positive rate decreases exponentially in the number of layers. Figure 1 shows how this process works for infrequently queried negatives n_4 and n_5 as well as frequently queried negative n_2.

Item Insertion and Deletion. Stacked Filters retain the supported operations of the query-agnostic filters they use. If the underlying query-agnostic filter supports insertion, then so does the Stacked Filter. The same is true for deletion of positives.

Insertion of an element after construction follows the same path as an insertion 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. This process works even in the case that the element was previously a frequently queried negative. The deletion algorithm follows the same pattern as the insertion algorithm, except it deletes instead of inserts the element at every positive layer.

4 METRIC EQUATIONS

We now present the metric equations and derivations for Stacked Filters. These equations are then used in Section 5 to show how to tune Stacked Filters for optimal performance, and in Section 7 to show how Stacked Filters outperform state-of-the-art filters. We show that compared to query-agnostic filters, Stacked Filters are as robust, while improving drastically in EFPR and size.

Notation. Stacked Filters metrics are written as a function of $\vec{\alpha} = (\alpha_1, \ldots, \alpha_{T_L})$ plus an additional dummy $\alpha_0 = 1$ which is used for

readability. 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 $EFPR'(\vec{\alpha})$. The metrics are variations on either an exponential function or geometric series. To make this clear by immediate inspection, we will often consider that all α_i values have the same value α (this also makes α and T_L easier to optimize in Section 5).

4.1 Stacked Filters EFPR

To calculate the total EFPR for a Stacked Filter, we introduce a new variable ψ which captures the probability that a negative element from query distribution D is in N_f, i.e. $\psi = \mathbb{P}(x \in N_f | x \in N)$.

Frequently Queried Negatives. For $x \in N_f$, a Stacked Filter returns 1 if and only if it makes it to the end of the stack. This occurs only if it is a false positive on each positive layer and so the probability of this happening is

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

Infrequently Queried Negatives. For $x \in N_i$, 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 $\prod_{j=1}^{2i-1} \alpha_j \cdot (1-\alpha_{2i})$, 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

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

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

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

If all α values are equal, then this is equal to

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

Thus, the FPR for frequently queried negatives is exponential in the number of layers and goes quickly to 0, whereas infrequently queried negatives have EFPR close to the FPR of the first layer.

4.2 Stacked Filter Sizes

Size of a Stacked Filter Given the FPR at Each Layer. For every positive layer after the first, 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 in bits per positive element is

$$\sum_{i=0}^{(T_L-1)/2} s(\alpha_{2i+1}) \cdot \left(\prod_{j=0}^{i} \alpha_{2j}\right)$$

Similarly, negatives appear in a negative layer if they are false positives for every prior positive layer and so the size of all negative layers (using the traditional metric bits per positive element) is

$$\sum_{i=1}^{(T_L-1)/2} s(\alpha_{2i}) \cdot \frac{|N_f|}{|P|} \cdot \left(\prod_{j=1}^{i} \alpha_{2j-1}\right)$$

The total space for a Stacked Filter is the sum of these two equations. Because α_i values are small, the products in parenthesis go to 0 quickly in both equations and so the total size of a Stacked Filter is dominated by its first layer.

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

$$s'(\vec{\alpha}) \leq s(\alpha) \cdot \left(\frac{1}{1-\alpha} + \frac{|N_f|}{|P|}\frac{\alpha}{1-\alpha}\right) \tag{2}$$

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

Stochasticity of Size or Filter Behavior. When constructing Stacked Filters, there are two choices when it comes to space. First, all filters can have their memory allocated up front. Using this method, size is fixed but if a higher proportion of elements makes it through the initial layers of the stack, bad behavior can happen at the subsequent filters in the stack. This happens in the form of increased FPR (Bloom filters), failed construction (Cuckoo filters), or long probe times (Quotient filters). Instead, our default is to allocate size proportional to the number of items which make it to a layer in the stack (see lines 6 and 14 of Algorithm 1). This makes the size of a Stacked Filter random, however, for large sets the size of a Stacked Filter concentrates sharply around its mean. In particular,

$$\mathbb{P}(|s' - E[s']| \geq kE[s']) \leq$$
$$\frac{1}{|P|} \cdot \frac{\alpha_{max}}{k^2} \cdot \left(\frac{s(\alpha_{min})(1-\alpha_{min})}{s(\alpha_{max})(1-\alpha_{max})}\right)^2 \cdot \frac{(1+\frac{|N_f|}{|P|})}{(1+\frac{|N_f|}{|P|}\alpha_{min})^2}$$

where $\alpha_{min}, \alpha_{max}$ are the lowest, highest FPRs of any layer in the Stacked Filter. The proof can be found in Technical Report Section 11.3.1 [16]. The leading $\frac{1}{|P|}$ term ensures that for large sets the chance of deviating away from the expected set size is negligible.

4.3 Stacked Filter Robustness

The First Layer Provides Robustness. Any element in N is either in N_i or N_f, and so its probability of being a false positive is either $\mathbb{P}(F(x) = 1 | x \in N_i)$ or $\mathbb{P}(F(x) = 1 | x \in N_f)$. Since elements of N_i have a higher chance of being a false positive, the EFPB of a Stacked Filter is $\mathbb{P}(F(x) = 1 | x \in N_i)$. For a Stacked Filter, an easy bound on this is the FPR of the first layer. Since the majority of the size of a Stacked Filter is in its first layer, worst case performance is similar to a query-agnostic filter (of the same size).

Performance Change Under Workload Shift. While EFPB provides worst case bounds, the EFPR equation shows what happens under the common case of more mild workload drifts. 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 on the change in ψ to ψ'. In particular, the change in EFPR is

$$(\psi' - \psi) \cdot \big(\mathbb{P}(F(x) = 1 | x \in N_f) - \mathbb{P}(F(x) = 1 | x \in N_i)\big)$$

Thus, the performance in terms of EFPR for a Stacked Filter decreases linearly with the change in the proportion of queries aimed at frequently queried negatives.

4.4 Stacked Filter Computational Costs

Like the previous derivations, the resulting equations for query computation time and construction time are modified geometric series. We give here bounds on the resulting equations specifically

for all α_i equal to α. The derivations for exact equations with general α_i and an arbitrary number of layers are in Technical Report Section 11.2 [16]. All equations are in terms of average computational cost and given as the number of base filter operations required.

Construction. The construction cost of Stacked Filters is

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

Like in the previous subsections, filters after the first add only negligible costs to construction to Stacked Filters when α is small. The majority of the cost above comes from 1) $c_i \cdot |P|$ for constructing the first layer and 2) $c_q \cdot (|N_f| + |P|)$ for querying the first layer.

Query Costs. The costs for querying a Stacked Filter for a positive, frequently queried negative, and infrequently queried negative are:

$$c'_{q,P} \leq \frac{2}{1-\alpha}c_q, \quad c_{q,N_f} \leq \frac{1+2\alpha}{1-\alpha}c_q, \quad c'_{q,N_i} \leq \frac{1}{1-\alpha}c_q \qquad (3)$$

For small α, the cost of querying negative elements is essentially identical to querying a single filter. The cost of querying positive elements is about 2× the cost of a single filter as they make it through the first layer with certainty before being rejected at layer 2 with high probability.

5 OPTIMIZING STACKED FILTERS

Sections 3 and 4 introduced Stacked Filters given their parameters: the set of frequent negatives, the FPRs of each layer, and the number of layers. We now complete the picture of how Stacked Filters are constructed. This is done in two stages: first we go from a sample of past queries to a workload model, and then we go from a model of the workload to the choice of Stacked Filters parameters.

5.1 Modeling the Workload

Like classifier-based filters, Stacked Filters require workload knowledge in the form of a sample of past queries. This set of past queries can have multiple sources depending on the application. For instance, it can be: 1) publicly available, as in the case of URL blacklisting [28] with popular non-spam websites and their query frequencies collected by OpenPageRank [2], 2) collected by the application by default, as is the case for web indexing and document search [24], where query term frequencies are collected and stored, or 3) can be collected by the system by choice, as is the case for most data systems including key-value stores [41].

After collecting the set of sample queries, Stacked Filters create a model to identify frequently queried elements. They do this by creating a smoothed histogram of the empirical query frequencies. More specifically, each element observed in the set of sample queries is put into a set N_{samp}. Then, the proportion of queries at elements outside N_{samp} is estimated by looping over all possible subsets of $Q-1$ queries from the set of Q sample queries, and seeing for what proportion of subsets the Qth query value is not present in the set of $Q-1$ queries. If we denote this value by ℓ, then for each $x_i \in N_{samp}$, its query frequency is estimated as $\frac{(1-\ell)}{Q} \cdot \sum_{j=1}^{Q} \mathbb{1}_{q_j = x_i}$. Our optimization algorithms below then choose some of the values in N_{samp} to be in N_f, creating the frequent negatives set.

5.2 Optimization Algorithms

The final step in constructing Stacked Filters is to use the workload model to choose N_f, T_L, and $\{\alpha_i\}_{i=1}^{T_L}$. The optimization algorithms which do so depend on the base filter being used and fall into two categories. In the first, the query-agnostic filter can take on any value of α, which is a good approximation for filters such as Bloom Filters. In the second, the possible α values are of the form 2^{-k} for $k \in \mathbb{N}$, which is true or a very close approximation for fingerprint based filters such as Cuckoo and Quotient Filters. For both methods, we assume that the base filter has a size equation of the form $s(\alpha) = \frac{-\log_2(\alpha)+c}{f}$ with $c \geq 0, f \geq 1$. This holds true or is a very close approximation for all major filters in practice including Bloom, Cuckoo, and Quotient filters, and additionally covers the equation for the theoretical lower bound on size for query-agnostic filters. Throughout the section, optimization is given in terms of minimizing EFPR with respect to a constraint on size. Optimization of size with respect to a bound on EFPR is similar. Additional constraints on the EFPB or expected number of filter checks may be added by only minor modifications.

5.2.1 Outer Loop: Sweeping over N_f. For both continuous and discrete FPR filters, there is an outer loop which chooses sets of N_f to optimize and an inner optimization which optimizes the Stacked Filter given N_f. The best performing value of N_f is then used.

To choose N_f, we make use of the workload model and choose N_f to be a subset of N_{samp}. The $\psi(N_f)$ value is the sum of the estimated query frequencies of each value chosen to be in N_f. Because EFPR is a monotonically decreasing function of ψ, for a fixed size N_f it is optimal to greedily choose the negative elements queried most. Thus we can order N_{samp} by the element's query frequencies and then sweep over various sizes for N_f, always using the most frequently queried elements of N_{samp} to be in N_f. The following theorem shows we can choose the size of N_f efficiently. Its proof, and the proof of all other theorems in this paper, is given in the Technical Report [16].

Theorem 1. *Given an oracle returning the optimal EFPR for a given set N_f, finding the optimal EFPR across all values of $|N_f|$ to within ϵ requires $O(\frac{1}{\epsilon})$ calls to the oracle.*

The core idea of the theorem is that values of $|N_f|$ that are close together have solutions with optimal EFPR close to each other. Using the theorem, our algorithm starts with a "current" N_f of size 0. It then increases $|N_f|$ to a strategically chosen larger value, making sure the skipped values of $|N_f|$ could have EFPR no more than ϵ lower than the checked values, and runs the optimization with the new fixed ψ and $|N_f|$. It continues to do so until $|N_f| = |N_{samp}|$, and returns the setup giving the best observed EFPR.

5.2.2 Inner Optimization: Continuous FPR Filters. The inner optimization loop for continuous filters has N_f and ψ given and works in two steps. First, we assume that all layers have the same FPR and optimize the filter as if it had infinitely many layers. Second, we truncate the infinite layer Stacked Filter to a small finite layered one that is close in performance to the infinite layer one. An optional third step modifies the procedure to search filters with varying α values across layers, but we note that this procedure is optional as it generally does not improve the EFPR.

Step 1: Fixed N_f, Infinite Equal FPR Layers. Take the equations of Section 4 with FPR equal across layers and let $T_L \to \infty$. The equations for EFPR, size, and EFPB converge to $s'(\alpha) = s(\alpha) \cdot$

$(\frac{1}{1-\alpha} + \frac{|N_f|}{|P|}\frac{\alpha}{1-\alpha})$, $EFPR(\alpha) = (1-\psi)\frac{\alpha}{1+\alpha}$, and $EFPB = \frac{\alpha}{1-\alpha}$, and the equations for computation converge to Equation (3).

By inspection, the equations for EFPR, EFPB, and computation monotonically increase with α. Thus attempts to minimize the EFPR or satisfy EFPB or computation constraints should all lower α. For the size equation, the following theorem holds.

Theorem 2. *The function* $s'(\alpha) = \frac{-\log_2(\alpha)+c}{f} \cdot (\frac{1}{1-\alpha} + \frac{|N_f|}{|P|}\frac{\alpha}{1-\alpha})$ *is quasiconvex on (0,1) when* $c \geq 0, f > 0$.

Quasi-convex functions have unique global minima, and as a result, the size equation can be minimized via gradient descent. Specifically, we use gradient descent with backtracking line-search to choose the step size. To minimize EFPR using size as a constraint, at a given time step if we are below the size constraint we decrease α. Otherwise, we use the gradient of size with respect to α to decrease the size. If for the given N_f one or more constraints is not satisfiable, we return an exception.

Step 2: Truncating to a Finite Stack. When performing truncation, we measure the difference in each metric equation using infinite T_L and an increasing finite value of T_L and stop when the difference is below ϵ for all metric equations. Because each metric equation is either an exponentially decreasing function of T_L or a geometric series in T_L, the convergence to the infinite layer values is on the order of $O(\alpha^{T_L/2})$, and so usually 5 or 7 layers suffice.

Algorithm Analysis. By using $\frac{\epsilon}{3}$ in both the outer loop over N_f and both steps 1 and 2, the overall algorithm is an ϵ approximation to the best possible EFPR for a Stacked Filter with α fixed across layers. Its runtime is $O(\epsilon^{-1}+|N_{samp}|)$ and its empirical optimization times for Stacked Bloom Filters at 10 bits per element are listed in Table 2 under "Bloom, fixed α". The workload, described in detail in Section 8.2, is the synthetic integer dataset with a Zipf distribution with $\eta = 1$, and $|N_{samp}| = 5 \cdot 10^7$.

For both this algorithm as well as the subsequent two, we note the runtime has two regions. When ϵ is small, the runtime is approximately linear in $|N_{samp}|$. As ϵ grows, it becomes the primary cost of algorithmic runtime and the runtime is linear in ϵ^{-1}.

Varying FPRs Across Layers. When allowing the α_i values to change across layers, we are faced with a non-convex optimization objective and constraint, even when fixing T_L (this can be seen by taking second derivatives). To perform optimization, at each checked value of N_f we first run the optimization using equal FPR across layers. We then polish the resulting filter by using the gradient-free algorithm COBYLA [36] to modify the FPRs of each layer. While this method very occasionally achieves improvements over the fixed FPR per layer method, it generally does not, as seen in Table 2. Additionally, an alternative strategy of discretizing the search space for the FPRs at each layer and using the optimization routines described in Section 5.2.3 also did not in general improve upon the equal FPR per layer solution. Thus we view this final polishing as optional in the optimization of continuous FPR filters.

5.2.3 Inner Optimization: Fingerprint Based Filters. For fingerprint based filters, the discrete number of fingerprint bits makes search easier. The main idea of our approach is to use breadth first search expanding the number of fingerprint bits used at each layer, working two layers at a time: one positive and one negative. At each pair of layers, derived bounds on which possible fingerprint lengths can lead to an optimal solution of each layer are used, constraining the number of options expanded. Eventually, each search path terminates, either because its choices already created too many false positives, it used all the available space budget, or the number of queries which would reach the current layer of the chosen stack is less than ϵ. The full algorithm, its explanation, and proofs of its theoretical properties are given in the Technical Report [16].

	Bloom, fixed α		Bloom, varied α		Cuckoo	
ϵ	EFPR	Time	EFPR	Time	EFPR	Time
10^{-2}	0.00175	775μs	0.00175	47 ms	0.00203	12 ms
10^{-3}	0.00173	781μs	0.00173	328 ms	0.00190	13 ms
10^{-4}	0.00172	1.01 ms	0.00172	2.9 s	0.00184	44 ms
10^{-5}	0.00172	3.7 ms	0.00172	26.7 s	0.00184	367 ms

Table 2: Optimization is efficient and tunably optimal

Theoretically, the algorithm is guaranteed to return a filter with $EFPR \leq EFPR^* + \epsilon$, where $EFPR^*$ is the EFPR of the best possible filter satisfying all constraints. We can bound the runtime of the algorithm theoretically by $O(|N_{samp}|+\epsilon^{-3})$. Additionally, the proof of the runtime bound does not rely on several key optimizations of the algorithm, and the experimental run time of the algorithm behaves more like $O(\epsilon^{-1})$. Thus, the algorithmic run time is both tunable and efficient, as can be seen in Table 2 for Cuckoo Filters.

6 INCREMENTAL CONSTRUCTION AND ADAPTIVITY

So far we assumed that workload knowledge can be collected and that workloads are static or drift slowly. While this holds for many filter use cases such as URL Blacklisting [28] and Web Indexing [24], there are many other applications where workloads change quickly, in which case continuously gathering workload knowledge is expensive. To address these use cases, we introduce Adaptive Stacked Filters (ASFs) which require knowledge about workload shape (such as how skewed they are), but do not require the gathering of a set of negative queries. Crucial to the design of ASFs is the idea of incremental construction, which allows ASFs to process queries immediately, learn frequent negatives during query evaluation, and gain benefits from stacking before finishing construction. Mirroring how we described Stacked Filters, we explain first the structure of ASFs given their parameters, then how to collect workload knowledge, and finally how to optimize their parameters.

Incremental Construction. ASFs start by constructing L_1 and use this to answer incoming queries until more layers are constructed. They also allocate an empty L_2. During query processing, when a false positive occurs, it is added to L_2. Then, when L_2 is full (in that it either hits its load factor for Cuckoo and Quotient filters or has half its bits set for Bloom filters), the ASF brings in the positive set, queries it against L_2 and adds the false positives on L_2 to a new L_3. Processing then continues using the layers up until L_3, gaining the benefits in terms of EFPR that come with extra layers. Additionally, construction on layers L_4 and L_5 can begin (if they exist), and uses the same procedure. Since N_f is captured during query processing, it does not need to be gathered before the construction of the ASF. This is the primary benefit of ASFs over Stacked Filters: they require only the workload shape (to figure out how big each layer should be) but not which values are important.

Rank-Frequency Workload Knowledge. To create ASFs, we need a rank-frequency distribution of the negative query workload. This is akin to the workload knowledge of Section 5.1, but instead of describing the query frequencies of actual values, the distribution describes the query frequencies of the value at each rank, where rank is itself defined by ordering elements' query frequencies. A classic example of this type of distribution is the Zipf distribution, which models how often the first and second most popular values occur without reference to the actual values.

The rank-frequency distribution can be calculated in many different ways. We do so using a set of past queries to make a smoothed histogram as described in Section 5.1. ASFs assume that the workload shape is relatively static even if the values are not, and under this case, ASFs rebuild themselves without performing a new analysis of the workload. For instance, YouTube video queries and other periodic workloads are a good example of a case where this holds: queries for popular videos on a given week follow consistent patterns even if which videos are popular changes each week [12]. In this case, an ASF being rebuilt and adapting to new frequent values knows what shape to take before it knows the new frequent values.

Optimizing Collection vs Exploitation. We optimize ASFs in two different ways, depending on the nature of the workload. The first uses the optimization procedures of base Stacked Filters assuming we pick the most frequently queried negatives and allocates the exact same filter. It then creates the filter incrementally during query processing instead of all at once.

The above process builds the best eventual ASF but can face many queries before achieving a fully built ASF. For this reason, we additionally create a second approach which assumes that ASFs are 3 layers and focuses on building a fully built ASF very quickly. This form of optimization takes as input an estimate of how long the filter will last and then chooses a number of queries to observe when building the second layer, denoted by N_o. The value of N_o determines the expected values of ψ and $|N_f|$:

$$E[\psi] = \sum_{x \in N_{samp}} f(x)(1 - (1 - f(x))^{N_o})$$

$$E[N_f] = (\ell \cdot N_o) + \sum_{x \in N_{samp}} (1 - (1 - f(x))^{N_o})$$

where here we recall that ℓ is the estimate of what portion of queries fall on values outside our sample. The optimization then weights the EFPR using just L_1 for N_o queries vs. the EFPR of the ASF using all 3 layers on the rest of the queries. To choose the best configuration, we perform grid search on N_o. At each value of N_o, we either calculate or estimate ψ and $|N_f|$, depending on the size of N_{samp}, and perform optimization of the three layers using discrete search for both continuous FPR filters and integer-length fingerprint filters (see Technical Report Sec. 11.5 for details [16]).

Monitoring and Adapting. To maintain robust performance, if the elements in N_f become less frequently queried over time, this needs to be rectified. To address this, the ASF monitors its performance and initiates a rebuild whenever the FPR differs by more than 50% from its expected FPR. The ASF initially tries a rebuild assuming that the popularity of particular values has changed but not the rank-frequency distribution; the layers after the first of the ASF are dropped and the procedure for construction starts from L_1. If this does not fix performance, a remodeling of the workload happens, and the filter is re-optimized and rebuilt from scratch.

Positive Set Adaptivity. ASFs address the common use case where the positive set is static but the frequent negatives are changing. This is common for read-only datasets such as the levels of an LSM tree, and it is common in general for filters because filters are not easily adaptive to changes in data size. However, in cases where new items are frequently seen, filters need to be able to adapt. We explore several preliminary strategies for this in Technical Report Section 11.6 [16].

7 BETTER SIZE-FPR TRADEOFFS

With Stacked Filters and their adaptive counterparts described fully, we ask the following critical question: when are Stacked Filters better than query-agnostic filters and by how much? In terms of their trade-off between EFPR and size, the following theorems answer this question using only summary properties of the workload: namely a choice of $|N_f|$ and $\psi(N_f)$. Along with each theorem, we present a visualization of its results, which can be used by systems designers to estimate the benefits of Stacked Filters on their workload a priori to spending the time to gather workload knowledge.

Each theorem holds exactly in the case that α is a continuous parameter for the base query-agnostic filter, and we discuss how the theorems apply to integer length fingerprint filters at the end of this section. The first theorem shows when a Stacked Filter is strictly better than a query-agnostic filter, as opposed to when a 1-layer filter is best.

Theorem 3. *Let the positive set have size $|P|$, let the distribution of our negative queries be D, and let α be a desired expected false positive rate. If there exists any set N_f, $\psi = \mathbb{P}_D(x \in N_f | x \in N)$, and $0 \le k \le \psi$ such that*

$$\frac{|N_f|}{|P|} \le \frac{\ln \frac{1}{1-k}}{\ln \frac{1-k}{\alpha} + c} \cdot \frac{1-k-\alpha}{\alpha} - 1$$

then a Stacked Filter (optimized using Section 5 and given access to any N_f satisfying the constraint) achieves the EFPR α using fewer bits than a query-agnostic filter.

Figures 2a and 2b use this theorem to create visualizations of which workloads Stacked Filters are certainly better than query-agnostic filters. Figure 2a shows this for a desired EFPR of $\alpha = 0.03$; any workload with a set N_f such that $|N_f|, \psi(N_f)$ is above the line has a Stacked Filter which is strictly better than a Bloom filter. Figure 2b shows this trend for more alpha values. Even at a high desired α of 0.05, Stacked Filters cover a sizeable number of workloads; many workloads contain a negative set half the size of their positive set, and which contain 25% of all negative queries. As the desired EFPR decreases, Stacked Filters cover almost all real workloads; for instance at a desired EFPR of $\alpha = 0.01$, if $|N_f| = |P|$, then only 7% of negative queries need to be at values in N_f for the Stacked Filter to be more space efficient. At even lower values, the amount needed becomes negligible and almost any workload sees improvements.

Estimating Space Savings from Stacked Filters. Using the optimization routines of the prior sections and given values for $|N_f|$ and $\psi(N_f)$, it becomes possible to estimate the space savings of using a Stacked Filter as compared to a query-agnostic filter for any desired EFPR. Namely, we can optimize the size of a Stacked

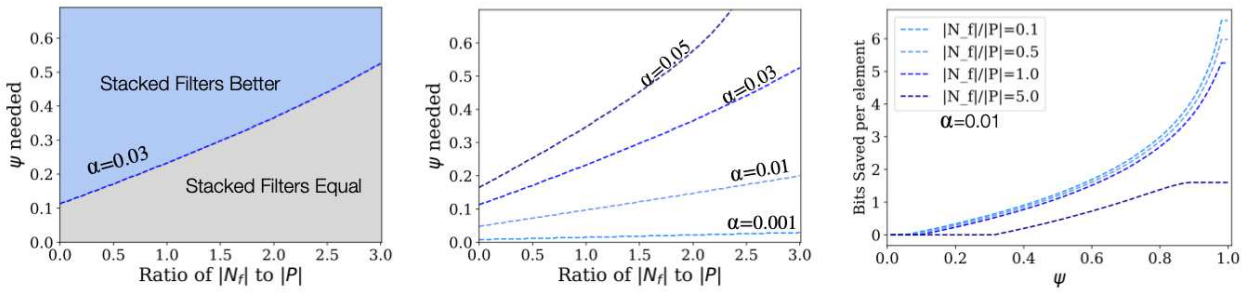


Figure 2: Analytical equations can predict both when Stacked Filters are better, and by how much.

Filter with equal FPRs across layers while requiring that its EFPR is less than a query-agnostic filter:

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

Figure 2c uses this to graph how a combination of $\frac{|N_f|}{|P|}$ and ψ produces a reduction in filter size using Stacked Bloom Filters at a desired EFPR of $\alpha = 0.01$. For each fixed value of $\frac{|N_f|}{P}$, the graph contains three parts: in the first part, it is not advantageous to build Stacked Filters and a query-agnostic filter is built. For all values of $\frac{|N_f|}{|P|}$, this is a small area and covers workloads with no frequently queried negative elements. In the second part, Stacked Filters have superlinear improvement in ψ, with improvement starting at 0 bits per element saved and going up to 7 bits per element saved. At the tail end of the graphs, the improvement stops even as ψ increases. This is the point where $\frac{\alpha}{1-\psi}$ crosses the minimal α_L value for size. After, the Stacked Filter can choose larger false positive rates at each layer while having the same EFPR as the query-agnostic filter, but this larger α_L value increases size. Instead, the Stacked Filter keeps the minimal α_L value for size and the Stacked Filter produces both a space benefit and has lower EFPR than the input α.

Integer Length Fingerprint Filters. The above equations and theory assumed that all FPRs were possible at each layer. For integer length fingerprint filters, this is not the case and the theory does not hold exactly; however, the general trajectory remains the same. Additionally, experimentally the results for integer length fingerprint filters are often better than the continuous FPR approximations suggest. This is because query-agnostic filters also suffer from limitations on the FPRs they can choose; often given more size as a budget there isn't enough space to add a full bit for every positive element. In these cases, Stacked Filters can often make use of this space to build layers deeper in the stack, and the added flexiblity of being able to use space on any layer in the stack provides additional improvements over the theory above.

8 EXPERIMENTAL ANALYSIS

We now experimentally demonstrate that Stacked Filters offer better false positive rates compared to query-agnostic Filters for the same size, or they offer the same false positive rate at a smaller size. We also show that Stacked Filters are more computationally efficient, robust, and are more generally applicable than classifier-based filters while offering similar false positive rates and sizes.

Filter Implementations. All filters use CityHash as the hash function [35]. The Counting Quotient Filter (CQF) and Cuckoo Filter (CF) implementations are taken from the original papers [20, 34]. In the original implementation, CQF is constrained to have the filter size be a power of two to allow for operations such as resizing and merging. This is not relevant to our testing, so we removed this restriction. For CF, the implementation provided only supports certain signature lengths, so we implemented a fix to allow all integer signature lengths. For classifier-based filters, we use Learned Bloom Filters [28] with text data using a 16 dimensional character-level GRU as in the original paper, and integer data using a shallow feed-forward neural network. Additionally, we compare with Sandwiched Learned Filters (SLF) [32], which uses the same model as the Learned Filter but has a query-agnostic pre-filter as well as a backup filter (Bloom filters). For Stacked Filter layers we use the same implementations as in the query agnostic filters. For most experiments we use Bloom Filters and we refer to the filter as Stacked Bloom Filter but we also show results with other filters.

Experimental Infrastructure. All experiments are run on a machine with an Intel Core-i7 i7-9750H (2.60GHz with 6 cores), 32 GB of RAM, and a Nvidia GeForce GTX 1660 Ti graphics card. Each experimental number reported is the average of 25 runs.

Datasets. We use three diverse datasets:

(1) *URL Blacklisting*: The URL Blacklisting application was used to introduce Learned Filters [28]. As the dataset in [28] is not publicly available, we instead use two open-source databases, Shalla's Blacklists [1] as a positive set of dangerous URLs, and the top 10 million websites from the Open Page Rank Initiative[2] as a negative set of safe URLs, with the probability of querying a safe URL proportional to its PageRank.

(2) *Packet Filtering*: Packet filtering is a common application for filters and was used to evaluate Counting Quotient Filters [34]. Following their lead, we use the benchmark Firehose [3] which simulates an environment where some subset of packets are labeled suspicious and need to be filtered. The benchmark is run under its default settings.

(3) *Synethetic Integers*: To more finely control experimental settings, we also use synthetic data. The data set consists of 1 million positive elements using randomly generated integer keys. Negative queries on the dataset come from a set of 100 million negative elements, also with randomly generated keys, and follow a Zipfian distribution. The skew of the negative query distribution is a controlled parameter η taking values between 0.5 and 1.25. For all experiments and graphs where η is not listed, $\eta = 0.75$.

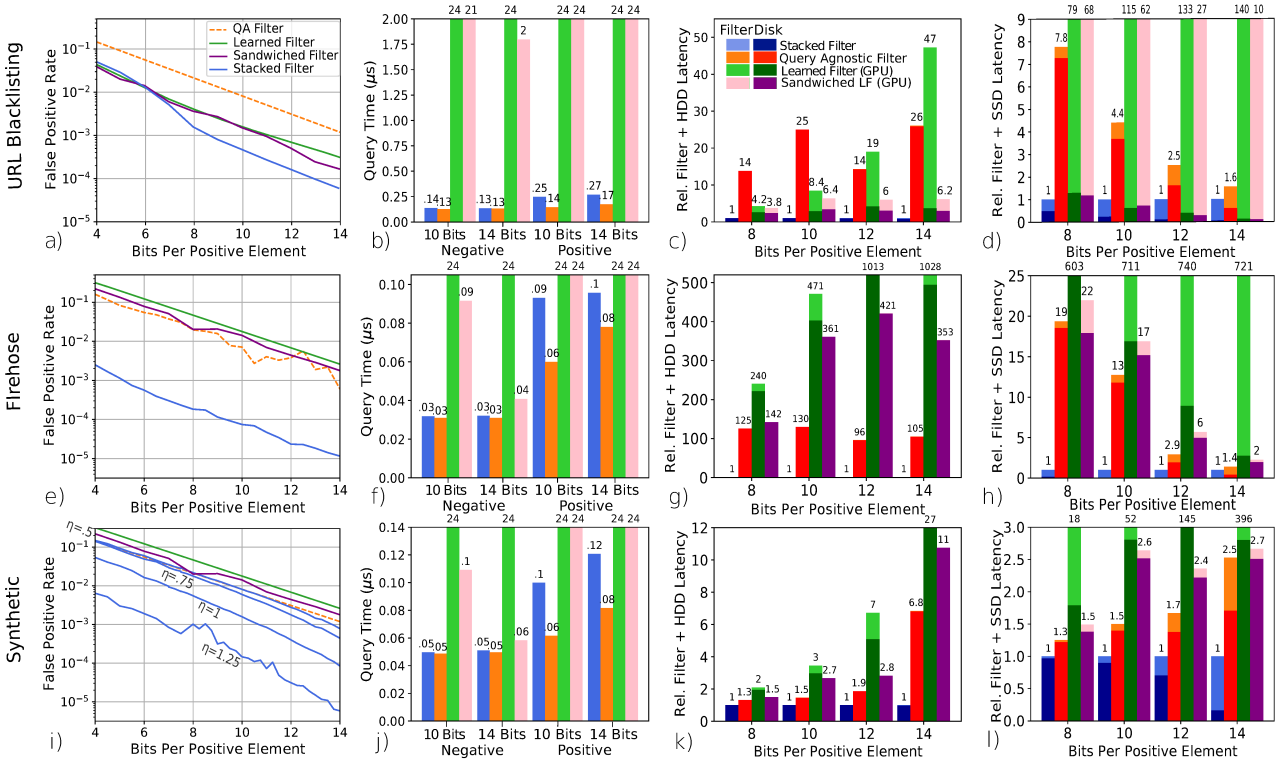


Figure 3: Stacked Bloom Filters achieve a similar EFPR to Learned Bloom Filters while maintaining a 170x throughput advantage and beat both query-agnostic and learned filters in overall performance on typical workloads.

For each dataset, we simulate having incomplete information about the query distribution by giving Learned, Sandwiched Learned, and Stacked Filters only the higher frequency half of the negative set for training. We also varied this amount from between 10% to 80% of the training data and saw the same relative results.

8.1 Evaluating Total Filter Performance

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 unnecessary operations incurred by false positives. While 1) is a characteristic of just the filter, 2) depends both on the FPR of the filter and the cost of the operations the filter protects against. Thus, we fix memory and measure FPR and probe (computation) time. We then translate these two metrics into total filter performance for the common case of protecting against base data accesses on persistent hardware using a slower HDD (favoring lower FPRs) and a faster NVMe SSD (favoring lower computational rates).

Workload Knowledge Improves FPRs. Figures 3a, e, and i show the false positive rates for all filters across all three datasets. Here we use Bloom filters for both the query-agnostic filter and the Stack Filter. We see that Stacked Bloom Filters is the clear winner across all workloads. Learned Filters are closer for the URL Blacklisting scenario which is a favorable scenario for learning but still Stacked Filters provide a better FPR for most memory budgets. For other workloads, Learned Filters give similar or slightly worse FPR than traditional query-agnostic filters while Stacked Filters provide a drastic benefit.

Across all three workloads, the negative query distribution has frequently queried elements and so Stacked Filters can find a "small" set of negative elements that contain a large portion of the query workload (we need only N_f to not be dramatically larger than P). Under these conditions, deeper stacks have only marginal overhead compared to a single layer and so Stacked Filters allocate almost all their space budget to the first layer. This way, they achieve essentially the same FPR as query-agnostic filters on infrequent negatives and at the same time, eliminate all false positives from frequently queried negatives. Stacked Filters improve upon the FPR of query-agnostic Bloom Filters by 5-100×, as seen in Figures 3a, e, and i. Alternatively, Stacked Filters can reduce their size significantly while achieving the same FPRs as query-agnostic filters.

Learned and Sandwiched Learned Filters' performance depends heavily on the dataset. With URL Blacklisting (Fig. 3a) where keys have semantic meaning and are correlated with their appearance in the positive or negative set, Learned and Stacked filters achieve similar results. When the keys have less semantic meaning such as in Firehose (Fig. 3e) or the synthetic integer data (Fig. 3i), Learned Filters' performance degrades and becomes worse than a standard Bloom Filter. Overall, the evaluation of Learned Filters depends on 1) how correlated keys are with their positive/negative set membership, and 2) how complicated their decision boundaries are for the dataset (this affects computational performance as well). Figure 3a shows that even on tasks that are considered good fit for Learned Filters, their performance can be matched by Stacked Filters.

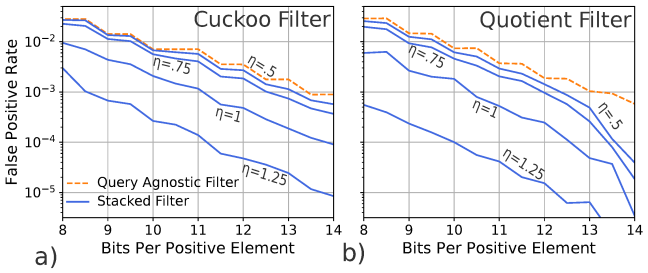


Figure 4: Stacking provides robust performance benefits across a variety of underlying filter types.

Hash-Based Filters Dominate Classifier-Based Filters Computationally. Figures 3b, f, and j depict the computational performance of each filter. Noticeably, Learned Filters have computational performance that is orders of magnitude slower than hash-based filters, with the difference between 90-190×. In comparison, Stacked Filters have computational performance more in line with query-agnostic filters, with the difference between 1-2×. For queries on negative elements, the Bloom Filter and Stacked Bloom Filter have essentially identical computational cost, and for queries on positive elements, the Stacked Filter probe is about 1.5× the cost of the Bloom Filter probe. For Sandwiched Learned Filters, their performance is a weighted combination of hash-based filters and the classifier; overall, they are at least 3× more expensive and can be as much as 90× more computationally expensive.

Stacked Filters Maximize Overall Performance. As Figures 3 c-g-k and d-h-l show, Stacked Filters strike the best balance between decreased false positive rates and affordable computational speeds, resulting in the best overall performance across both hard disk and SSD. Compared to query-agnostic 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 1.4-130× lower. In comparison to the classifier-based Filters, Stacked Filters are better or equal in terms of false positive rates and better by orders of magnitude in computational performance, resulting in 1.5-1028× lower total workload costs.

8.2 Stacking Improves Diverse Filter Types

Figures 4a, and b show that Stacked Filters benefits generalize across diverse filter types used for the stacked layers. More specifically, Figure 4 shows the performance of Stacked Cuckoo and Stacked Quotient Filters on the synthetic integer dataset. This is the same experiment as for Stacked Bloom Filters in Figure 3i. Collectively these three graphs all show the same benefits, which is that Stacked Filters achieve lower false positive rates compared to their query-agnostic counter-parts as workloads are more skewed. This is because more skewed workloads query their frequent negatives more often and because at lower FPRs the first layer in a Stacked Filter culls almost all elements, making subsequent layers in the Stack cheaper to build (as can be seen in Equation (2)). Further, while not shown here, the computational costs of Stacked Cuckoo and Stacked Quotient Filters follow the same patterns as seen in Figures 3b, f, and j, leading to similar benefits in terms of total throughput.

8.3 Stacked Filters are Workload-Robust

We now demonstrate that Stacked Filters retain the robustness of query-agnostic filters, bringing an additional benefit over classifier-based filters. We focus on two facets of robustness: maintaining performance under shifting workloads and providing utility for a variety of workloads. We use the synthetic integer dataset with size $s = 10$, and Bloom filters for both the query-agnostic and the Stacked filter.

Robust to Workload Shifts. Figure 5a shows how Stacked Filters' performance adapts to workload changes with diverse values for η. We vary the workload by reducing the value of ψ from its initial value to a value of 0. This captures scenarios where the frequently queried negative values are changing over time, and so Stacked Filters are no longer optimized for the most frequently queried negatives. For Stacked Filters, regardless of the skew of their initial distribution, drastic workload shifts are needed to become worse than query-agnostic filters, with query-agnostic filters being better only after the frequently queried negative set loses more than 60% of its initial set. Even in the extreme case that every frequently queried element from when the Stacked Filter was built is no longer queried ($\psi = 0$), the Stacked Filter is never more than 50% worse than a query-agnostic filter.

Robust to Workload Misspecification. Figure 5b shows the behavior when the sample queries used to build the filter come from a different distribution than the future workload. Here the queries come from the integer dataset with $\eta = 0.75$, however, during workload modeling, we used η values from 0.15 to 1.35. Figure 5b shows that even when the modeled workload is significantly different from the true workload, the Stacked Filter retains most of its performance and outperforms query-agnostic filters.

Robust to Workload Type. Because Stacked Filters rely on taking advantage of frequently queried negative values, uniform workloads are the most difficult ones. However, if $|N|$ is relatively small, every negative element is still frequently queried. Figure 5c shows that Stacked Bloom Filters outperform query agnostic Bloom filters when $|N|$ is a reasonable multiple of $|P|$, anywhere up to 25×. Since the uniform distribution is the worst distribution possible for Stacked Filters, this shows that Stacked Filters are better than query-agnostic filters for all small universe sizes.

Easy to Reconstruct. While Stacked Filters are effective under shifting workloads, they need to be rebuilt to regain their best performance. Figure 5d shows that the construction cost grows slowly with the number of the frequently queried negatives, is significantly faster than classifier-based filters, and is comparable to query-agnostic filters. Because Stacked Filters can be reconstructed quickly, they can handle periodic bulk updates. Such a strategy is key for handling workloads with dynamic positive data.

8.4 Incrementally Adapting to Workload Shifts

We now show that Adaptive Stacked Filters (ASFs) are capable of adapting quickly to changing workload patterns improving on the (good) worst case guarantees of base Stacked Filters. We use the synthetic integer dataset with Zipf parameter 1, let $s = 10$ bits per element, and use Bloom Filters for both the query-agnostic and Stacked Filter. The ASF is optimized using the 3-layer approach.

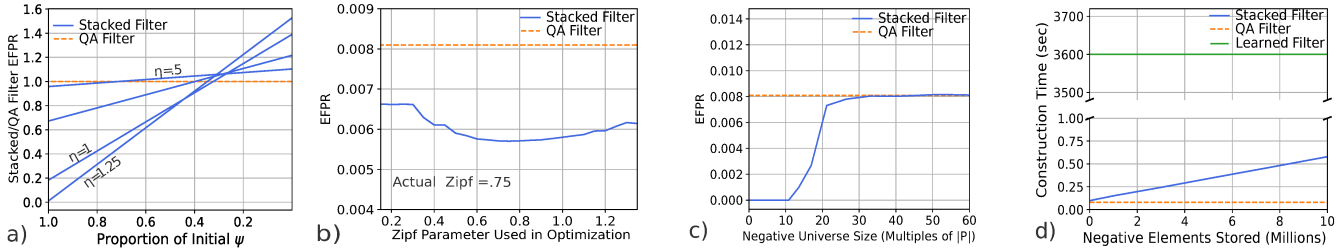
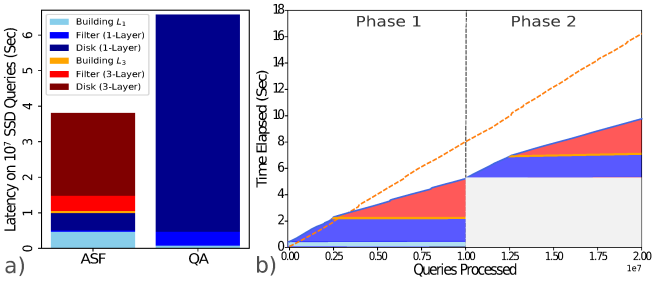


Figure 5: Stacked Filters are robust to workload shifts, skew, noisy data, and can be rebuilt quickly.



c)

	ASF 1-Layer	QA Filter	ASF 3-Layer	Static SF
FPR	.0101	.0087	.0028	.0017

Figure 6: High performance without workload knowledge.

Fast Incremental Construction. Figure 6a shows the performance of an ASF and a query-agnostic filter on a static workload of 10 million queries, with the filters protecting against data accesses to an SSD. The height of each bar shows the total time to process the queries, and the colors indicate the breakdown of how that time is spent. Overall, the ASF results in better performance over the workload compared with the query-agnostic filter, as its gains in FPR after the 3rd filter is built more than compensate for the cost to build L_3 and its slightly worse performance when using only L_1.

Adapting to Shifting Workloads. Figure 6b shows that ASFs work well for shifting workloads. Phase 1 of the experiment replicates the previous experiment; then, in phase 2, the query frequency distribution remains a Zipf with parameter $\eta = 1$ but the popularity of the elements is flipped so that the previously least popular element is now the most popular and vice versa. The figure details how long the ASF and query-agnostic filter take to process each workload. Additionally, the colored bars show what phase the ASF is in during query processing at the time of each query.

While phase 1 shows as before that the ASF performs well on static workloads, phase 2 shows that the ASF can adapt to shifts in workload. The ASF does so by recognizing at the start of phase 2 that a shift has occurred and signaling a rebuild. Then, performance continues as in a static phase: the ASF learns the new workload pattern quickly by incrementally building layer 2, then builds layer 3 and capitalizes on the reduced false positive rate once that occurs. Ultimately, across both phases, ASFs process the workload 1.6× faster than query-agnostic filters.

Breaking Down ASF Performance. Figure 6c breaks down the benefits of ASFs and compares them with static Stacked Filters. The table repeats a trend seen throughout the paper: the first layer of a Stacked Filter is nearly as performant as a query-agnostic filter. Then, as the ASF builds its 3rd layer its FPR drops to nearly 1/4 of its previous value and is 3× more performant than the query-agnostic filter. Finally, we see that compared to a static Stacked Filter on phase 1, ASF is about 1.6× worse, so if workloads are sufficiently static, then a static Stacked Filter is best.

9 RELATED WORK

Sandwiched Learned Filters. Sandwiched Learned Filters (SLFs) are an extension of Learned Bloom Filters which use a preliminary filter before querying the classifier [32]. Similarly to our findings, the use of a sequence of filters brings increased space efficiency and reduced false positive rates. However, SLFs remain centered around a computationally expensive model, whereas Stacked Filters achieve performance similar to hash-based filters. In addition, Stacked Filters extend to arbitrary heights, while extending SLFs to more layers would need to take into account the difficulties arising from using a series of dependent ML models.

Filters that Use Frequent Negatives. There is a limited existing literature on Bloom Filters which use knowledge of the negative set in construction to create better false positive rates. Most of these require different assumptions than Stacked Filters such as 1) being able to store the whole negative set [10, 30] or 2) allowing some false negatives [19]. The closest in functionality would be Adaptive Cuckoo Filters [33], which correct false positives at the cost of extra space. Compared to Stacked Filters, they more easily adapt to the workload but have a worse FPR-size tradeoff.

Filter Variations. Many designs have been proposed to solve the standard filtering problem and work to lower the FPR vs. size tradeoff of filters [4, 6, 20, 44]. Other filter designs aim at building better filters in terms of their system throughput and optimize their computation or memory accesses [9, 15, 26, 27, 29, 31, 34, 37]. This work is all complementary to Stacked Filters as increasing the efficiency of each base filter creates better Stacked Filters.

10 CONCLUSION

Stacked Filters learn to structurally take advantage of workload knowledge via hashing, as opposed to learning to separate keys from non-keys based on semantic information via a classifier. They eschew intra-key similarity in order to focus on succinct hash-based memorization of frequent negatives. This provides provably good properties in terms of computation, robustness, and false positive rate. By taking into account workload knowledge, Stacked Filters also provide better FPR/space tradeoffs than query-agnostic filters, providing a good balance of the best properties of both query-agnostic and Learned filters.

ACKNOWLEDGEMENTS: This work is partially funded by the USA Department of Energy project DE-SC0020200.

REFERENCES

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

[2] Top 10 million websites: Openpagerank. "https://www.domcop.com/openpagerank/what-is-openpagerank".

[3] K. Anderson and S. Plimpton. Firehose streaming benchmarks, 2015. "https://firehose.sandia.gov/".

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

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

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

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

[8] J. Bruck, J. Gao, and A. Jiang. Weighted bloom filter. pages 2304 – 2308, 08 2006.

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

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

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

[12] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon. Analyzing the video popularity characteristics of large-scale user generated content systems. *IEEE/ACM Transactions on networking*, 17(5):1357–1370, 2009.

[13] Z. Dai and A. Shrivastava. Adaptive learned bloom filter (ada-bf): Efficient utilization of the classifier, 2019.

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

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

[16] K. Deeds, B. Hentschel, and S. Idreos. Technical report: Stacked filters, 2020. http://daslab.seas.harvard.edu/publications/StackedFilters.pdf.

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

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

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

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

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

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

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

[24] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He. Bitfunnel: Revisiting signatures for search. In *SIGIR '17 Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2017.

[25] T. M. Graf and D. Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters, 2019.

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

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

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

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

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

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

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

[33] M. Mitzenmacher, S. Pontarelli, and P. Reviriego. Adaptive cuckoo filters. *ACM J. Exp. Algorithmics*, 25, Mar. 2020.

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

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

[36] M. J. D. Powell. *A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation*, pages 51–67. 1994.

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

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

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

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

[41] RocksDB. Rocksdb trace, replay, analyzer, and workload generation, 2020. "https://github.com/facebook/rocksdb/wiki/RocksDB-Trace,-Replay,-Analyzer,-and-Workload-Generation".

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

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

[44] M. Wang, M. Zhou, S. Shi, and C. Qian. Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters. *Proc. VLDB Endow.*, 13(2):197–210, Oct. 2019.

[45] X. Wang, Y. Ji, Z. Dang, X. Zheng, and B. Zhao. Improved weighted bloom filter and space lower bound analysis of algorithms for approximated membership querying. In M. Renz, C. Shahabi, X. Zhou, and M. A. Cheema, editors, *Database Systems for Advanced Applications*, pages 346–362, Cham, 2015.