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:

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.
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 $\varepsilon$-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 1000× 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 $E_D[F_D[\mathbb{P}(F(x) = 1|x \notin P)]]$. 


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 $\epsilon$, 

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

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 $\alpha$ 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 $c_{LS}$. 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 $L_2$ 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 $N_f$, which includes both $P$ and $N \setminus N_f$, which we denote by $N_1$ and call the infrequently queried negative set. Since $P \cup N_f$ 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 $L_3$ reach $L_3$ since they are in the construction set for $L_3$, 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 100x 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).
Algorithm 1 ConstructStackedFilter(Sp, W, C)
Input: Sp, W, C: data set, workload, constraint
1: Snf := Optimized(FW,C)  // In Sec. 5
2: // Construct the layers in the filter sequentially.
3: for i = 1 to T do
4:     Si := {}  // layer positive
5:     for xp ∈ Sp do
6:         Lc.insert(xp)  // construct Lc with space s(αi) * |Sp|
7:     for xn ∈ Snf do
8:         Li.insert(xn)  // layer negative
9:     if Li(xn) = 1 then
10:        Si = Si ∪ {xn}
11:     Snf = Sr
12:     else
13:         allocate Li w/ space s(αi) * |Snf|  // layer negative
14:     for xn ∈ Snf do
15:         Li.insert(xn)
16:     if Li(xp) = 1 then
17:        Sr = Sr ∪ {xp}
18:     Sp = Sr
19: 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 Nf, and 2) the number of layers T and false positive rate of each layer α1,...,αT.

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 xp 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 xp follows the same path. Either xp 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 p1.

Querying a Negative. For an infrequently queried negative element x, it is in neither P nor Nf 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 L1, and the majority of false positives occur from elements rejected at L2. 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 n4 and n5 as well as frequently queried negative n2.

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 FPR and size.

Notation. Stacked Filters metrics are written as a function of (α1,...,αT) plus an additional dummy α0 = 1 which is used for
To calculate the total EFPR for a Stacked Filter, we introduce a new variable $\psi$ which captures the probability that a negative element from query distribution $D$ is in $N_F$, i.e. $\psi = P(x \in N_F | x \in N)$.

**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

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

**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 $\prod_{j=i}^{2i-1} \alpha_j \alpha_i$, 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

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

**Expected False Positive Rate.** Since $N_I$ and $N_F$ partition $N$, the EFPR of a Stacked Filter is

$$\psi \prod_{i=1}^{T_I} \alpha_i + (1 - \psi) \left( \prod_{i=1}^{(T_L-1)/2} \alpha_{2i+1} \right) + (1 - \psi) \left( \prod_{i=1}^{(T_L-1)/2} \alpha_{2i+1} \right)$$

If all $\alpha$ values are equal, then this is equal to

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

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 $\alpha_i$ 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 $\alpha$ values are the same, we can use a geometric series bound on both arguments above, giving

$$s'(\tilde{\alpha}) \leq s(\alpha) \cdot \frac{1}{1 - \alpha}$$

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,

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

where $\sigma_{\text{min}}, \sigma_{\text{max}}$ 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 $N_I$ is either in $N_I$ or $N_F$, and so its probability of being a false positive is either $P(F(x) = 1 | x \in N_I)$ or $P(F(x) = 1 | x \in N_F)$. Since elements of $N_I$ have a higher chance of being a false positive, the EFPR of a Stacked Filter is $P(F(x) = 1 | x \in N_I)$. 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 EFPR 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 $\psi$, which changes to $D'$ and corresponding $\psi'$, the change in EFPR from $D$ to $D'$ depends only on the change in $\psi$ to $\psi'$. In particular, the change in EFPR is

$$(\psi' - \psi) \cdot P(F(x) = 1 | x \in N_F) - P(F(x) = 1 | x \in N_I)$$

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 $a_i$ equal to $a$. The derivations for exact equations with general $a_i$ 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

$$\sum |P|(c_i + c_q) \frac{1}{1 - a} + |N_f|c_q + |N_f|/(c_i + c_q) \frac{a}{1 - a}$$

Like in the previous subsections, filters after the first add only negligible costs to construction to Stacked Filters when $a$ 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:

$$c_{q,P} \leq \frac{2}{1 - a} c_q \quad c_{q,N_f} \leq \frac{1 + 2a}{1 - a} c_q \quad c_{q,N_i} \leq \frac{1}{1 - a} c_q$$

(3)

For small $a$, the cost of querying negative elements is essentially identical to querying a single filter. The cost of querying positive elements is about $2x$ 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 $Q$th query value is not present in the set of $Q - 1$ queries. If we denote this value by $\ell$, then for each $x_i \in N_{samp}$, its query frequency is estimated as $(1 - \ell) \cdot 2^{Q \sum_{j=1}^{Q} 1_{q_j = x_i}}$. 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 $\{a_i\}_{i=1}^{T_L}$. 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 $a$, which is a good approximation for filters such as Bloom Filters. In the second, the possible $a$ 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 $s(a) = \frac{-\log f(\alpha)}{f}$ 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 EFPR 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 $\psi$, 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 $\epsilon$ requires $O(1/\epsilon)$ 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 $\epsilon$ lower than the checked values, and runs the optimization with the new fixed $\psi$ 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 $\psi$ 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 $a$ 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 $s'(\alpha) = s(\alpha)$.
By using \( \eta \) with Varying FPRs Across Layers. Table 2 under "Bloom, fixed" shows two layers at a time: one positive and one negative. At each

5.2.3 Inner Optimization: Fingerprint Based Filters.

For fingerprint polishing as optional in the optimization of continuous FPR filters. Upon the equal FPR per layer solution. Thus we view this final search space for the FPRs at each layer and using the optimization in Table 2. Additionally, an alternative strategy of discretizing the FPR across layers. We then polish the resulting filter by using the \( N \) by taking second derivatives. To perform optimization, at each \( T \) objective and constraint, even when fixing \( \epsilon \) change across layers, we are faced with a non-convex optimization equation is either an exponentially decreasing function of \( \epsilon \). The difference is below \( \epsilon \).

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 \( \alpha \). Otherwise, we use the gradient of size with respect to \( \alpha \) 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 \( T_0 \) and an increasing finite value of \( T_0 \) and stop when the difference is below \( \epsilon \) for all metric equations. Because each metric equation is either an exponentially decreasing function of \( T_0 \) or a geometric series in \( T_0 \), the convergence to the infinite layer values is on the order of \( O(a^{T_0/2}) \), and so usually 5 or 7 layers suffice.

Algorithm Analysis. By using \( \frac{1}{2} \) in both the outer loop over \( N_f \) and both steps 1 and 2, the overall algorithm is an \( \epsilon \) approximation to the best possible EFPR for a Stacked Filter with \( \alpha \) 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 \( \alpha \)". 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 \( \epsilon \) is small, the runtime is approximately linear in \( |N_{samp}| \). As \( \epsilon \) grows, it becomes the primary cost of algorithmic runtime and the runtime is linear in \( \epsilon^{-1} \).

Varying FPRs Across Layers. When allowing the \( a_i \) values to change across layers, we are faced with a non-convex optimization objective and constraint, even when fixing \( T_0 \) (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 \( \xi [36] \) 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 \( \epsilon \). The full algorithm, its explanation, and proofs of its theoretical properties are given in the Technical Report \( [16] \).

Theoretically, the algorithm is guaranteed to return a filter with \( \text{EFPR} \leq \text{EFPR}^* + \epsilon \), where \( \text{EFPR}^* \) 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 \( O(\epsilon^{-3}) \). Thus, the algorithmic run time is both tunable and efficient, as can be seen in Table 2 for Cuckoo Filters.

### Table 2: Optimization is efficient and tunably optimal

<table>
<thead>
<tr>
<th>( \epsilon )</th>
<th>Bloom, fixed ( \alpha )</th>
<th>Bloom, varied ( \alpha )</th>
<th>Cuckoo</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10^{-5} )</td>
<td>0.00175</td>
<td>0.00173</td>
<td>0.00172</td>
</tr>
<tr>
<td>( 10^{-4} )</td>
<td>0.00102</td>
<td>0.00101</td>
<td>0.00102</td>
</tr>
<tr>
<td>( 10^{-3} )</td>
<td>0.00007</td>
<td>0.00006</td>
<td>0.00007</td>
</tr>
</tbody>
</table>

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, 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 \( \epsilon \). The full algorithm, its explanation, and proofs of its theoretical properties are given in the Technical Report \( [16] \).

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 not the rank-frequency distribution; the layers after the first of the performance and initiates a rebuild whenever the FPR differs by more than 5%. 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 $\alpha$ 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 $\alpha$ be a desired expected false positive rate. If there exists any set $N_f$, $\psi = P_D(x \in N_f | x \in N)$, and $0 \leq k \leq \psi$ such that

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

then a Stacked Filter (optimized using Section 5 and given access to any $N_f$ satisfying the constraint) achieves the EFPR $\alpha$ 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 $\alpha$ 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...
We now experimentally demonstrate that Stacked Filters offer better false positive rates at a smaller size. This is the point where \( \alpha \) keeps the minimal \(|\mathcal{F}P|\) value for size. After, the Stacked Filter can choose larger false positive rates at \( \alpha \). This is the point where \( \alpha \) is less than a query-agnostic filter:

\[
s(\alpha) = \min_{a_L} \left( \frac{1}{1 - \alpha} \frac{|N_f|}{|P|} + \frac{1}{1 - a_L} \right)
\]

\[
s.t. \quad a_L \in (0, 1 - \psi]
\]

Figure 2c uses this to graph how a combination of \( \frac{|N_f|}{|P|} \) and \( \psi \) 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 \( \psi \), 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 \( \psi \) increases. This is the point where \( \frac{a_L}{1 - \psi} \) crosses the minimal \( a_L \) 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 \( a_L \) value increases size. Instead, the Stacked Filter keeps the minimal \( a_L \) value for size and the Stacked Filter produces both a space benefit and has lower EFPR than the input \( \alpha \).

**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.

**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. **Synthetic 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 \( \eta \) taking values between 0.5 and 1.25. For all experiments and graphs where \( \eta \) is not listed, \( \eta = 0.75 \).
Workload Knowledge Improves FPRs. Figures 3a, e, and i show that Stacked Bloom Filters are the clear winner across all workloads. Learned Filters are closer for the URL Blacklisting Filter. We see that Stacked Bloom Filters is the clear winner across persistent hardware using a slower HDD (favoring lower FPRs) and while achieving the same FPRs as query-agnostic filters.

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 performed due to 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 that Stacked Bloom Filters are 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.

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.

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 \(N_t\) 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 5–100x, as seen in Figures 3a, e, and i. Alternatively, Stacked Filters 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.
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 \(90-190x\). In comparison, Stacked Filters have computational performance more in line with query-agnostic filters, with the difference between 1-2x. 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\(x\) 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\(x\) more expensive and can be as much as 90\(x\) 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\(x\) 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\(x\) lower total workload costs.

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 \(\eta\). We vary the workload by reducing the value of \(\psi\) 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 \(\eta\) 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\(x\). 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.
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 taken to process the queries, and the colors inside each bar show 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 1.6x 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 3x more performant than the query-agnostic filter. Finally, we see that compared to a static Stacked Filter on phase 1, ASF is about 1.6x 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.

Filter 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.
11 PROOFS

11.1 Proof of Size Concentration Bounds

Theorem 4. Let $s'$ be the size of a filter using Algorithm 1, and assume that a Stacked Filter of $T_L$ layers false positive rate $a_i$ for each layer $i \in \{1, \ldots, T_L\}$. Let $\alpha_{\text{max}} = \max_i a_i \leq \frac{1}{2}$ and $\alpha_{\text{min}} = \min_i a_i$. Then

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

Proof. The result is an application of Chebyshev’s inequality. Start by letting $X_{n,j}^{m,i} \in \{1, \ldots, |N_f|\}$ denote the event that item $j$ of the set $N_f$ made it to negative layer $n$. Similarly, $X_{n,j}^{m,i}$ denote the event positive item $j$ made it to positive layer $n$. Since $s'$ is in bits per positive element, we have

$$s' = \frac{1}{|P|} \sum_{i=1}^{n} \sum_{j=1}^{n} s(a_{2i-1})X_{n,j}^{m,i} + \sum_{i=1}^{n} \sum_{j=1}^{n} s(a_{2i})X_{n,j}^{m,i}.$$

By using $\alpha_{\text{max}}$ in the size equation and $\alpha_{\text{min}}$ for the conditional chance of making it to the next layer, we have $E[s'] \geq s(\alpha_{\text{max}}) \frac{1 + |N_f|}{1 - \alpha_{\text{max}}}$. The variance part, we have that $\sum_{i=1}^{n} X_{n,j}^{m,i} \leq X_{n,j}^{m,i}$ unless $C_l = C_2$ and $j = j' \neq i$ by our method of construction. Thus

$$\Pr[|s' - E[s']| \leq \frac{\alpha_{\text{min}}}{|P|} \sum_{i=1}^{n} \sum_{j=1}^{n} X_{n,j}^{m,i}] + \Pr[|s(\alpha_{\text{max}}) - E[s(\alpha_{\text{max}})]| \leq \frac{\alpha_{\text{min}}}{|P|} \sum_{i=1}^{n} \sum_{j=1}^{n} X_{n,j}^{m,i}]$$

To break down the summation, we use Eve’s law. Letting $F_{\alpha_{\text{max}}}^n$ be the filtration containing all events up to negative layer $n$, we have

$$\Pr[\sum_{i=1}^{n} X_{n,i}^{m,i}] = E[\sum_{i=1}^{n} X_{n,i}^{m,i} | F_{\alpha_{\text{max}}}^{n-1}] + \Pr[\sum_{i=1}^{n} X_{n,i}^{m,i} = 0].$$

From here, one can use Eve’s law recursively to expand the expression on the right until reaching an expression only involving $X_{n,0}$. Evaluating this gives:

$$\Pr[\sum_{i=1}^{n} X_{n,i}^{m,i}] \leq \sum_{i=1}^{n} j \cdot |P|^{j-1} \cdot (1 - \alpha_{\text{max}})(1 - \alpha_{\text{max}})$$

We note that this makes intuitive sense as it is less variance than a geometric series using success parameter $(1 - \alpha_{\text{max}})$. A similar result holds for $\Pr[\sum_{i=1}^{n} X_{n,i}^{m,i}]$, and combining these two results with the variance expression above, the expected value expression above, and using Chebyshev’s inequality gives the desired result. \hfill \Box

11.2 Derivation of Computational Costs

Construction. For both positives and frequently queried negatives, the construction algorithm is best analyzed in pairs, wherein the same number of elements are inserted at one layer and queried against the next. For positives, every element is inserted into $L_4$ and 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|a_i(c_i + c_q) + |P|a_{2a}(c_i + c_q) + \ldots.

In more concise notation, this is

$$|P|(c_i + c_q) \frac{\alpha}{1 - \alpha} \sum_{i=0}^{T_L - 1} \alpha(i + |N_f|)(c_i + c_q) \frac{\alpha}{1 - \alpha}$$

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 frequently queried negatives having the first layer unpaired instead of the last. The number of operations to insert negatives is

$$|N_f|c_q + |N_f|(c_i + c_q) \sum_{i=1}^{T_L - 1} \alpha(2i - 1)$$

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

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

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

In this case, comparing against the cost for a single base filter of $c_i \cdot |P|$, the overhead is seen to be the cost of querying a base filter for both every positive and frequently queried negative element once, plus a small overhead from stacking.

Query Costs. The analysis of querying a Stacked Filter for a positive or frequently queried 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 $L_4$ and $L_2$ with certainty, $L_3$ and $L_4$ with probability $a_2$, $L_3$ and $L_4$ with probability $a_2$, $L_3$, and so on. The resulting equations for positive and frequently queried negative elements are

$$c_i' = 2c_q + 2c_q \sum_{i=0}^{T_L - 1} \prod_{i=0}^{T_L - 1} (1 - \alpha(2i - 1))$$

$$c_q'_{N_f} = c_q + 2c_q \sum_{i=0}^{T_L - 1} \prod_{i=0}^{T_L - 1} (1 - \alpha(2i - 1))$$

For an infrequently queried 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 $1, \ldots, i - 1$

$$c_q'_{N_f} = c_q \prod_{i=0}^{T_L - 1} (1 - \alpha(2i - 1))$$

Letting the FPRs at each layer be equal and using geometric series bounds, we have:

$$c_i' \leq \frac{2}{1 - \alpha} c_q'$$

$$c_q'_{N_f} \leq \frac{1 + \alpha}{1 - \alpha} c_q$$

11.3 Proofs

11.3.1 Proof of Quasiconvexity. Note: can cite paper "Beyond Convexity: Stochastic Quasi-Convex Optimization" for theoretical convergence to local minimum in some convergence speed.

Theorem 5. The size function $s(\alpha) = \frac{-\log(\alpha)}{f} + \frac{1 + \alpha}{|P|} \frac{\alpha}{1 - \alpha}$ is quasiconvex on $(0, 1)$ for $f > 0$.

Proof. For functions of a single variable, a sufficient condition for quasiconvexity is that $s'$ starts negative, has at most one root, and is then positive after (in the case a root exists). We use this to
prove the quasiconvexity of $s$. For notational convenience, we will replace $\frac{|N_f|}{|P|}$ with $N$, and note $N > 0$.

We have
\[
s'(\alpha) = \frac{N\alpha^2 + (1 - N)\alpha - (N + 1)\alpha \ln \alpha - 1}{(1 - \alpha)^2 \alpha \ln 2}
\]
Let $f$ be the numerator of $s'$, i.e., $N\alpha^2 + (1 - N)\alpha - (N + 1)\alpha \ln \alpha - 1$, and note the denominator is positive for $\alpha \in (0, 1)$. It follows that $\text{sign}(s') = \text{sign}(f)$, and therefore it is enough to show $f$ starts negative and either has no roots, or has one root and is positive for all values after its root.

First, as $\alpha \to 0$, $f(\alpha) \to -1$, so $f$ starts negative. Second, we note the functional forms of $f$, $f'$:
\[
f'(\alpha) = 2N(\alpha - 1) - (N + 1)\ln \alpha
\]
\[
f''(\alpha) = 2N - \frac{N + 1}{\alpha}
\]
By inspection, $f''(\alpha)$ has one root in $\mathbb{R}$, and so by Rolle's Theorem, $f'(\alpha)$ has at most two roots in $\mathbb{R}$. $f'(1) = 0$, so it has at most one other root. Since $\lim_{\alpha \to 0} f'(\alpha) = \infty$, then if the other root is after 1, $f$ is monotonically increasing on $(0, 1)$ and we are done. Otherwise, $f''$ has a single root $\alpha_1 \in (0, 1)$. It follows that $f$ cannot have a root in $(\alpha_1, 1)$ since $f(1) = 0$ and $f(\alpha_2) = 0$ for $\alpha_2 \in (\alpha_1, 1)$ would imply the existence of another root for $f'$.

Since $f'(\alpha) > 0 : \forall \alpha \in (0, \alpha_1)$, it follows that if $f$ has a root in $\alpha^* \in (0, 1)$, then $f(\alpha) > 0 : \forall \alpha \in (\alpha^*, 1)$. This is what we set out to show and so $s$ is quasiconvex.

\[\square\]

**Theorem 6.** The function $s(\alpha) = \frac{-\log_2(\alpha^*+c)}{\frac{1}{\alpha^*} - \frac{|N_f|}{|P|}) + \frac{|N_f|}{|P|} \frac{\alpha}{1-\alpha}$ is quasiconvex on $(0, 1)$, when $c \geq 0$, $f > 0$.

**Proof.** As before let $N = \frac{|N_f|}{|P|}$. Take the derivative of $s$, note the denominator is positive, and form $f_c$ from the numerator of $s'$, which in this case is $N\alpha^2 + (1 - N)\alpha - (N + 1)\alpha \ln \alpha - 1 + \frac{(N + 1)\alpha \ln 2}{\alpha}$. As in the proof of Theorem 5, $\text{sign}(f_c) = \text{sign}(s')$. Additionally, note that $f_c$ is equal to $f$ from the previous problem plus an additional positive term: $(N + 1)\alpha \ln 2$.

In the case that $f$ is an increasing function on $(0, 1)$, then so is $f_c$ and we are done. Otherwise, let $f'$ have a root $\alpha_1$. Then on $(0, \alpha_1)$ both $f$ and $f_c$ are increasing functions and so have at most one root. On $(\alpha_1, 1)$, $f > 0$ so $f_c > 0$; thus both have no roots in $(\alpha_1, 1)$. Finally, we note that $\lim_{\alpha \to 0} f_c(\alpha) = -1$ and $f_c(1) = (N + 1) + c \ln 2 > 0$. Thus $f_c$ starts negative, has exactly one root, and is then positive. The same is therefore true of $s'$ and so $s(\alpha)$ is quasiconvex.

\[\square\]

11.3.2 Sweep over $N$ Proof.

**Theorem 7.** Given an oracle returning the optimal EFPR for a given set $N_f$ satisfying all constraints, finding the optimal EFPR across all values of $\frac{|N_f|}{|P|}$ within $e$ requires $O\left(\frac{1}{\epsilon}\right)$ calls to the oracle.

**Proof.** For $i \in [1, 2]$, let $N_i$ be the set of $n_i$ most heavily queried elements, let $\psi_i = P(x \in N_i | x \in N)$, let $\text{EFPR}_i^R$ be the best EFPR using $N_i$ satisfying all constraints, and let $P(x)$ be shorthand for the chance $x$ is a false positive using the Stacked Filter giving $\text{EFPR}_i^R$.

Then
\[
\text{EFPR}_i^R = \psi_1 P(x \in N_f) + (1 - \psi_1) P_2(x \in N_u)
\]

\[
= \psi_1 P(x \in N_f) + (1 - \psi_1) P_2(x \in N_u)
\]

\[+(\psi_2 - \psi_1)(P_2(x \in N_f) - P_2(x \in N_u))\]

\[\geq \text{EFPR}_i^R - (\psi_2 - \psi_1)\]

As a result, we can start at $\frac{|N_f|}{|P|} = 0$ elements and scan the most frequently queried elements in order. When more than $\epsilon$ difference exists in the $\psi$ values between the last call to the oracle and the current $N_f$, then we call the oracle again. By the above statements, this produces an EFPR within an $\epsilon$ amount of the best EFPR possible across all possible sets for $N_f$.

\[\square\]

Both Algorithm 3 and the algorithm for optimizing continuous FP PR Stacked Filters use less optimization subroutines than the proof above, although the number is still $O\left(\frac{1}{\epsilon}\right)$. The change comes modifying the final line of the proof, using the fact that if $\text{EFPR}_i^R \leq \alpha$, where $\alpha$ is the FP of a 1-layer Stacked Filter, then $P_2(x \mid x \in N_u) \leq \epsilon \frac{\psi_2 - \psi_1}{\psi_1 - \psi_2}$.

11.3.3 Proof of Theorem 3. We restate Theorem 3 here for convenience:

**Theorem 8.** Assume the equation for the size of a base filter in bits per positive element is of the form $s(\alpha) = \frac{-\log_2(\alpha^*+c)}{\frac{1}{\alpha^*} - \frac{|N_f|}{|P|}}$. Let the positive set have size $|P|$, let the distribution of our negative queries be $D$, and let $\alpha$ be a desired EFPR. If there exists any set $N_f$, $\psi = P_D(x \in N_f | x \in N)$, and $k \leq \psi$ such that
\[
\frac{|N_f|}{|P|} \leq \frac{\ln \frac{1}{1-k}}{\alpha} + \frac{1 - k - \alpha}{\alpha} - 1
\]
then a Stacked Filter (optimized using Section 5 and given access to any $N_f$ satisfying the constraint) achieves the EFPR $\alpha$ using fewer bits than a query-agnostic filter.

**Proof.** We limit ourselves to Stacked Filters which have an equal FPR $\alpha_k$ at each layer. In this case, we have that the EFPR of the filter is
\[
\psi_L^R_{\frac{1}{\alpha_k} - 1/2} + (1 - \psi)(1 - \alpha) \leq (\alpha_1 + \frac{\alpha}{2} + \cdots + \frac{\alpha}{L-2} + \frac{\alpha}{L})
\]

For some $L$, $T_L \geq 20$ implies this is less than $(1 - \psi)\alpha_k$. Thus, a Stacked Filter of length $N$ and $\alpha_k \leq \frac{\alpha}{L}$ has an EFPR less than $\alpha$.

Our goal is then to show that there exists a Stacked Filter with $\alpha_k \leq \frac{\alpha}{L} - \frac{\alpha}{L}$, which has size less than $\frac{-\ln(\alpha^*+\epsilon)}{\frac{1}{\alpha^*} - \frac{|N_f|}{|P|}}$. From Section 4, we have that the size of a Stacked Filter is bounded above by
\[
s(\alpha_k)\left(\frac{\frac{1}{\alpha_k} - \frac{|N_f|}{|P|}}{\frac{|N_f|}{|P|} - 1 - \alpha}ight)
\]

Plugging in our equations for the size of a base filter, a Stacked Filter is better than a traditional filter if $0 \leq \alpha_k \leq \frac{\alpha}{L} - \frac{\alpha}{L}$ and
\[
\frac{-\ln \alpha + c}{\ln 2} \leq \frac{\frac{1}{\alpha_k} - \frac{|N_f|}{|P|}}{\frac{|N_f|}{|P|} - 1 - \alpha}
\]

We now let $\alpha_k = \frac{\alpha}{L}$, and our EFPR equation gives the restraint $0 \leq k \leq \psi$. We further break apart our equation for size into two parts: our goal will be to show that $\frac{1}{\alpha_k} + \frac{|N_f|}{|P|} - \frac{1}{\alpha_k}$ $\leq b + 1$, and that $(1 + b) - \frac{-\ln \alpha + c}{\ln 2} \leq \frac{-\ln \alpha + c}{\ln 2}$. If both equations are satisfied, then (4) is satisfied as well.

Part 1: $(1 + b) - \frac{-\ln \alpha + c}{\ln 2} \leq \frac{-\ln \alpha + c}{\ln 2}$. Plugging in $\alpha_k = \frac{\alpha}{L}$ and solving this inequality gives: $b \leq \frac{-\ln \frac{1}{1-k} - \ln \frac{1}{k}}{\ln \frac{1}{k} - \ln \frac{1}{k}}$.

Part 2: $\frac{1}{\alpha_k} + \frac{|N_f|}{|P|} - \frac{1}{\alpha_k} \leq 1 + b$. Rearranging, we get $\frac{|N_f|}{|P|} \leq \frac{\frac{1}{|P|}}{\frac{1}{k} - 1 + b}$. We now set $b = \frac{-\ln \frac{1}{1-k}}{\ln \frac{1}{k} - \ln \frac{1}{k}}$, satisfying the inequality above,
Algorithm 3 Optimization of fingerprint-based filters

**Input:** $s$: a size constraint in bits/element

**Input:** $\epsilon$: a slack from optimum

1. $\text{FPR}_1L = \text{FPR\_SINGLE\_LAYER}(s)$
2. bestFPR, bestSetup = $\text{FPR\_1\_Layer}(s, \epsilon)$
3. $\text{prop} = \frac{|N_f|}{|P|}$

4. while $|N_f| > 1$ do
   - $\text{FPR} = \text{optNFixed}(s', \text{newPsi}, \frac{|N_f|}{|P|}, \frac{1}{\text{newPsi}})$
   - if $\text{FPR} < \text{bestFPR}$ then
     - bestFPR = $\text{FPR}$
     - bestSetup = $s'$

5. return bestFPR, bestSetup

Algorithm 4 optNFixed

**Input:** $f_1, f_2$: fingerprint sizes of new positive and negative layer

**Input:** optObj: prior optimization object

1. $\alpha_1, \alpha_2 = 2^{-f_1}, 2^{-f_2}$
2. newOpt.s = $(\text{optObj}\_s - f_1 - \text{objObj}\|N_f\| \times \alpha_1 \times f_2) / \alpha_2$
3. newOpt.$\psi = \frac{\text{optObj}\_\psi + (1 - \text{objObj}\_\psi) \times \alpha_2}{|N_f| \times \alpha_2}$
4. newOpt.$\frac{|N_f|}{|P|} = \text{optObj}\_\frac{|N_f|}{|P|} \times \frac{1}{\alpha_2}$

// prop is the proportion of negative queries which reach the end of the stack
5. newOpt.prop = optObj.prop $\times \alpha_1 \times (\text{objObj}\_\psi + (1 - \text{objObj}\_\psi) \times \alpha_2)$
6. newOpt.FPR = optObj.FPR + $(1 - \text{objObj}\_\psi \times \alpha_1 \times (1 - \alpha_2))$
7. newOpt.fingerprints = optObj.fingerprints + $\{f_1, f_2\}$
8. return newOpt

and plug in $\frac{c\epsilon}{1-\epsilon}$ for $\alpha_L$. The resulting equation is

$$\frac{|N_f|}{|P|} \leq \frac{\ln \frac{1}{1-k}}{\ln \frac{1}{1-k} + c} \cdot \frac{1 - k - \epsilon}{\epsilon - 1}$$

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

To see that the constructed Stacked Filter achieves this goal, note that as $\epsilon \to 0$, the optimization schemes of Section 5 approach the optimal filter using an equal FPR at all layers.

11.4 Optimizing Integer length Fingerprint Filters

The optimization algorithms for integer length fingerprint Stacked Filters work similarly to continuous FPR filters. One routine loops through possible values of $|N_f|$, always choosing greedily the most heavily queried elements, and calls another routine to optimize a Stacked Filter for a given $N_f$ and $\Psi$. The first routine does so in a way so as to make sure that the skipped values can produce no more than an $\epsilon$ better solution, as proven in Theorem 7.

11.4.1 Breadth First Search on Stacked Filters. Algorithms 4 and 5 deal with the optimization of an integer fingerprint filter given $N_f$ and $\Psi$. The description of the algorithm is split between the pseudocode given and various steps described in the text here. For integer-length fingerprint filters, the optimization algorithm uses breadth first search on the discrete options available, and its goal is to find a solution within $\epsilon$ of the best solution possible. The breadth first search starts with an empty stack, and at each step expands upon the current stack. It does so both by 1) finishing the current stack with a positive layer of maximum size (lines 6-9 of Algorithm 4), which puts no new options onto the BFS queue and 2) looping over the possible fingerprint lengths for 1 positive and 1 negative layer (lines 11-13 of Algorithm 4), and adding each of these deeper stacks onto the BFS queue.

A Brute Force Approach. As there are a discrete number of options for fingerprints at each layer (given our bounds on $s$), the BFS will build all possible 1 layer stacks, then all possible 3 layer stacks, then all possible 5 layer stacks, etc. Taking the best FPR solution of each of these would result in the exact optimal solution of the best Stacked Filter with 1 layers, 3 layers, 5 layers, etc. However, the number of possible Stacked Filters grows exponentially with the number of layers and so this is infeasibly slow even for just 7 or 9 layers. Additionally there isn’t any guarantee about how close the found solution is to the optimal.

From Brute Force to $\epsilon$-approximation. A more efficient solution is close each search path (i.e. a partially built stack) by building a final positive layer in a way that preserves the performance of BFS. In particular, we do so when using a final positive layer produces a solution within $\epsilon$ FPR of the best possible Stacked Filter starting with the same initial layers as the current partially built stack. Then, by aggregating the best seen result across all search paths, the overall algorithm finds a Stacked Filter within $\epsilon$ FPR of the optimal.

Stopping Condition. A search path is stopped when the proportion of negative queries which reach the end of the partial stack multiplied by the FPR of a single layer built using all available space budget is less than epsilon. As an example, assume we have a four layer stack each with FPR 0.01, that the initial $\Psi = 0.5$, that
\[ \epsilon = 10^{-3}, \text{ and that using all space left for the filter on a single positive layer gives a final layer with FPR 0.01. The proportion of queries which will reach the newly built 5th layer is } 0.5 + 0.01^2 + 0.5 \cdot 0.01^4 \approx 5 \cdot 10^{-4}, \text{ where the first term is the proportion of queries from } N_f \text{ and the second is the proportion from } N_i. \text{ The best hypothetical Stacked Filter rejects all these queries, resulting in no false positives from layer 5 onwards; a single layer stack would have an approximate EFPR of } 10^{-2} \cdot 5 \cdot 10^{-4} = 5 \cdot 10^{-6} \text{ from layer 5 onwards, which is less than } \epsilon \text{ so far rejecting all queries. Thus it is suitable close to the best possible Stacked Filter starting with four layers of FPR = 0.01, and so we choose the one layer stack and terminate the search path. The stopping condition is lines 6-9 of Algorithm 4.} \]

**Expanding the Search.** To perform the BFS, we need to figure out which limited paths need to be searched to produce an optimal solution. One obvious constraint is that all search paths should be feasible; i.e. we should only choose fingerprint lengths for each layer that do not go over the allowed filter size. A second obvious constraint is that layers should have FPR less than 1, which gives that the maximum FPR a filter has will be \( \frac{1}{\epsilon} \).

A third set of constraints comes from the following bounds, which is derived from the fact that the derivative of size with respect to \( \alpha \) should never be positive (as higher \( \alpha \) values always lead to worse EFPR and size). The second equation is just a rewritten version of \( \alpha \) to \( 1 \).

The bounds on \( \alpha_1 \) and \( \alpha_2 \) then become the following bounds for fingerprint sizes, where \( f_1, f_2 \) are the integer-valued fingerprint sizes for elements in \( L_1 \) and \( L_2 \):

\[
\begin{align*}
\alpha_1 &< \frac{|P|}{|N_f|} \cdot \frac{1}{\ln 2} - \frac{1}{\ln 2} \cdot \log_2 \alpha_2 + c \\
\alpha_2 &> 2^{\frac{|P|}{|N_f|} \cdot \frac{1}{\ln 2} - \frac{1}{\ln 2} \cdot \log_2 \alpha_2 + c}
\end{align*}
\]

The bounds on \( \alpha_1 \) and \( \alpha_2 \) becomes large, as this means a single layer filter often has FPR explosions. This is exactly what happens; if \( s \) is in bits per element, so even as the algorithm uses up space, if a negative layer choice eliminates many positive elements, \( s \) can rise dramatically. This is exactly how it works; if we let \( s' \) be the next value of \( s \), then

\[
s' = (2^{f_2-c}) \cdot (s - \alpha_1 \cdot \frac{|N_f|}{|P|} \cdot f_2)
\]

For almost all values of \( s \) and \( f_1 \) (which decides \( \alpha_1 \)), this equation either has very few \( f_2 \) values which are positive or the equation grows nearly exponentially and so \( s' \) explodes. Since the FPR is of the form \( 2^{s'c} \), growths in \( s \) drive the FPRs of a 1 layer stack quickly towards 0. For instance, at \( s = 40 \), the FPR is around \( 10^{-12} \), which is a lower FPR than essentially any filter requires in practice. The result is that many search paths terminate essentially immediately, and so far fewer Stacked Filter possibilities are added to the queue than predicted by just the algorithmic analysis above.
Filter FPR
1000 0.0031
1000 0.0032
1000 0.0033
5000 0.0034
6000 0.0035

This subsection details the process of optimizing a 3-layer ASF. As input the optimization has a number of queries the filter is expected to last, denoted by \( F_{TTL} \) as well as a size budget \( s \). We start by searching over a range of values for \( N_0 \), with the search performed logarithmically over values between 1000 and \( F_{TTL} \), i.e. each value of \( N_0 \) checked is 1.4 times as big as the previous value.

The value of \( N_0 \) determines the expected values of \( \psi \) and \( |N_f| \), which are:

\[
E[\psi] = \sum_{x \in N_{samp}} f(x)(1 - (1 - f(x))^{N_0}) \\
E[|N_f|] = (t \cdot |N_0|) + \sum_{x \in N_{samp}} (1 - (1 - f(x))^{N_0})
\]

Recall that \( t \) is the estimate of what portion of queries fall on values outside our histogram. The equations pessimistically assume in the second equation that all elements of \( N_f \) which were not present in the histogram sample will never again be queried, i.e. they are one-off queries. If the universe of elements is not extremely large or if the query pattern of the elements not present in the histogram is skewed, then the true \( E(\psi) \) may be higher, and the true \( E(|N_f|) \) may be smaller.

If the size of \( N_{samp} \) is large, then calculating exactly \( E[\psi] \) and \( E[|N_f|] \) can be expensive. Thus, when dataset sizes are larger than 1 million elements, we sample uniformly with replacement 100,000 values of \( N_{samp} \) and estimate \( E[\psi] \) and \( E[|N_f|] \) by:

\[
E[\psi] = (1 - t) - \frac{|N_{samp}|}{100,000} \sum_{i=1}^{100,000} f(x_i) \cdot (1 - f(x_i))^{N_0} \\
E[|N_f|] = (t \cdot |N_0|) + \frac{|N_{samp}|}{100,000} \sum_{i=1}^{100,000} (1 - (1 - f(x_i))^{N_0})
\]

In the above estimations, we make use of the fact that \( \sum_{x \in N_{samp}} f(x) = 1 \), and hence we only estimate the \( -f(x) \cdot (1 - (1 - f(x))^{N_0} \) term. This considerably reduces the variance of the estimate.

The optimization goal is to optimize the weighted sum of the EFP\( R \) using only \( L_1 \) for the first \( N_0 \) queries and the EFP\( R \) of the ASF using all 3 layers on the rest of the queries. To do this, we use discrete search on the FPRs of each layer for both continuous FPR filters and integer-length fingerprint filters. For integer-length fingerprint filters we check all fingerprint sizes on each layer which are between \( \lceil c + 1 \rceil \) and \( 2^{-16} \), and return the best configuration satisfying the size constraint. For the continuous FPR filters, we similarly use size bounds which provide FPRs of \( 2^{-2} \) and \( 2^{-16} \) on each layer, and then perform grid search where we check sizes that are 0.1 bits per element apart. As before, we return the best configuration that satisfies the size constraint.

Figures 7 and 8 display the optimization times and FPRs for ASFs on the synthetic integer dataset with and without sampling. The dataset has 1 million positive values, 100 million negative values, and a zipf of 1. The number of negative values in \( N_{samp} \) is the dependent variable. In general, for small datasets the times are comparable and execute in sub-second time. As the dataset size grows, the sampling based approach becomes significantly faster while maintaining similar accuracy.

### 11.5 3-Layer Optimization of Adaptive Stacked Filters

Learned Filters perform under such a scenario; in this case the slow rebuild makes them a significantly harder to deploy option. For all, they risk either blowing up their FPR (Bloom, Quotient) or rejecting the insertion (Cuckoo) after they grow beyond their desired number of elements \( n \). However, Stacked Filters suffer from insertions more acutely than query agnostic filters. Because the under-capacity positive layers have a lower initial EFPR, many of the negative elements which will eventually be false positives once the filter is at capacity do not register as false positives initially. Therefore, the Stacked Filter is unable to take advantage of its knowledge of known negatives as effectively as would normally. For Bloom Filters, there is a fairly easy fix. In general, one can insert negatives into \( L_2, L_4 \), and so on if they are close to being false positives in that they are 1 bit away from being a false positive.

Figures 9 and 10 show the changes in FPR of a Stacked Filters from additions into the filter with and without the one-bit adjustment. In the experiment, we use the integer experiment from the paper with an initial positive set of 1 million elements, a negative universe of 100 million elements, 10 bits per positive element, and a zipf parameter of .75. To perform the experiment, we start by allocating a filter under the assumption that the filter can hold 1.25 million positive elements. We then insert half a million new positive elements and test the incremental change in the EFPR of both the Stacked Bloom Filter and query agnostic Bloom filter. The results both show that Stacked Filters can struggle with new insertions moreso than query-agnostic filters, but also that with the adjustment Stacked Filters remain more performant. For both Stacked Filters and query-agnostic filters, the difference in FPR of the filter between its low point and its high point is a full order of magnitude.

Because filters struggle with new items, a common approach is to rebuild the filter on new updates. To assess whether this strategy would be effective for Stacked Filters as well, we include the results of an additional experiment in figure 11. This experiment compares the latency of query agnostic filters and stacked filters on a workload consisting of a single bulk load followed by 10 million queries against an SSD for a range of positive set sizes. All other conditions are the same as they are in the above experiments. This shows that for reasonable positive set sizes the stacked filter provides a lower overall latency on a rebuild and query workload, implying that they can be used effectively in filtering schemes which rely on this paradigm. The graph also shows how Learned and Sandwiched Learned Filters perform under such a scenario; in this case the slow filter rebuild makes them a significantly harder to deploy option when fast rebuilds are needed.
Figure 9: Normal Stacked Filter

Figure 10: With one bit adjustment

Figure 11: Rebuild + Query Time