Stacked Filters Technical Report

Anonymous Author(s)

ABSTRACT

This file contains the technical report for Stacked Filters, and is consistently being updated. As of the current moment, it is anonymized so that no personal information is conveyed.

1 INSERTION AND DELETION OF ELEMENTS

During insertion, a positive element alternates between inserting itself into every positive filter, and checking itself against every negative filter, stopping at the first negative filter which rejects the element. Pseudocode is given in Algorithm 1.

The deletion algorithm for positive elements follows the same pattern as the insertion algorithm, except it deletes instead of inserts the element at every positive layer. Pseudocode is given in Algorithm 2.

Stacked filters do not support the insertion of new known negative elements, as this changes which positives are rejected at each layer. For instance, if a positive element during insertion is rejected at layer 2, it does not add itself to layers 3+. Inserting a new known negative element might change this, so that the element would now be a positive at layer 2; then querying for the positive would turn up a false negative. Deleting known negative elements is possible, but doesn't provide much value (it does not lower the FPR or size of the filter).

Algorithm 1 Insert(x_P)

Require: L: the array of AMQ structures which makes up the Stacked Filter.
T_L: the total number of layers.
x_P: a positive element.

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

Algorithm 2 Delete(x_P)

Require: L: the array of AMQ structures which makes up the Stacked Filter.
T_L: the total number of layers.
x_P: a positive element.

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

2 SECTION 6 PROOFS

Notes to self: Any mention of numerous approximations made herein: 1) the size of the filters 2) the ability to make filters of non-integer number of bits (and bits per element in case of Cuckoo / Quotient Filters). We restate Theorem 1 here for convenience:

Theorem 2.1. *Assume the size of a base filter in bits per positive element is* $s(\alpha_i) = \frac{-\ln(\alpha_i)+f}{c}$*, where* α_i *is the input false positive rate for this specific base filter. Let the positive set have size* $|P|$*, let the distribution of our negative queries be D, and let* α *be a desired expected false positive rate. If there exists any set* N_k*,* $\psi = P(x \in N_k | x \in N)$*, and* $k \leq \psi$ *such that*

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

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

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

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

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

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

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

□

3 OPTIMIZATION

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

3.1 The Governing Equations: FPR and Size

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

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

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

3.2 Continuous Optimization for Stacked Bloom Filters

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

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

This is plugged into the inversion of the following,

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

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

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

At this point, we do a two-stage optimization over the modified EFPR and Size functions. In the first stage, we enforce that every layer's EFPR be equal and calculate the optimal EFPR/Size for all layers subject to the Size/EFPR constraint using the ISRES global optimization algorithm from the NLOPT package [1, 2]. Then, we "polish" this equal set of EFPRs for every layer by running the local optimization algorithm COBYLA where the minimization and constraint functions are the same except that the layers can all have different EFPRs.

3.3 Integral Optimization for Stacked Cuckoo and Quotient Filters

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

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

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

4 ADDITIONAL EXPERIMENTS

4.1 Computational Performance of Various Stacked Filters

The computational performance of Stacked Bloom, Cuckoo, and Quotient Filters was examined in the same experimental setup as in section 5.2 of the main paper. The zipf parameter is fixed at .75 for each of these experiments.

As seen in Figure 1, the same trends that were found in the URL blacklisting application appear here for all three filter types which were tested. Stacked Filters have nearly identical throughput for negative queries as traditional filters. For positive queries, Stacked Filters have slightly over half the throughput of traditional filters.

The only notable difference between the graphs is the reduced throughput in bloom filters as the bits per element

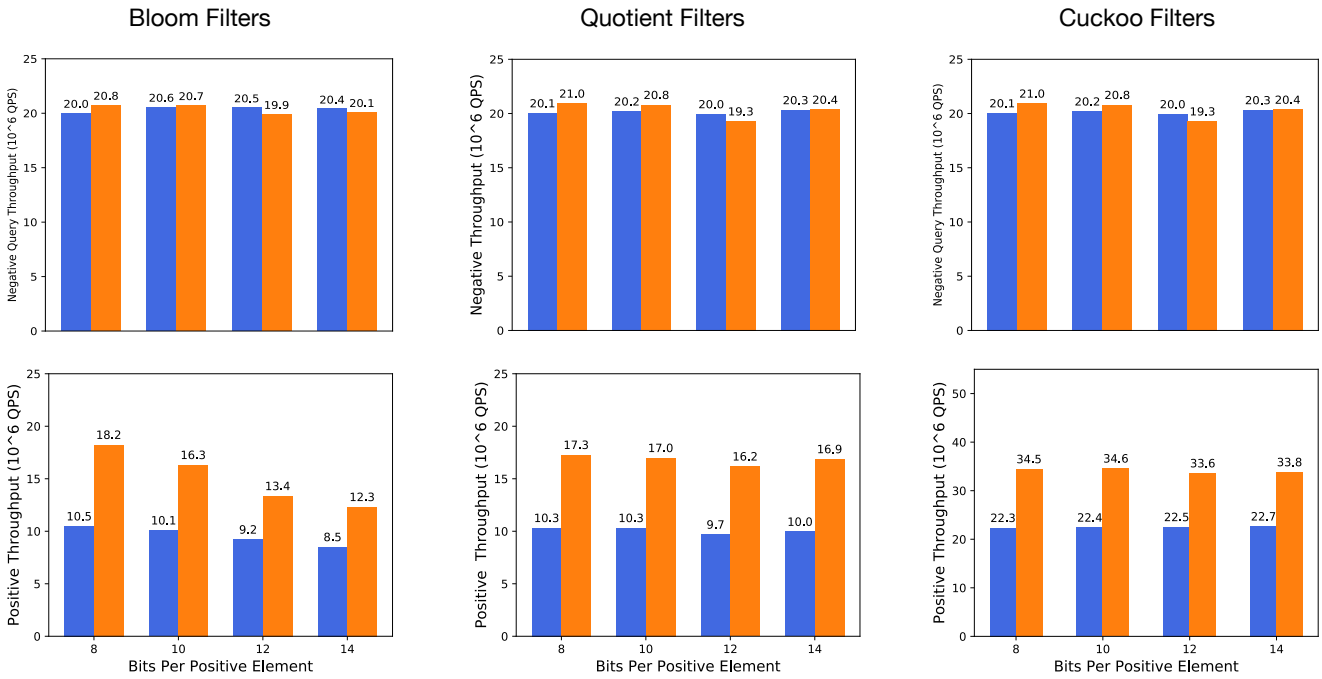

Figure 1: Computational Performance of Stacked Filters

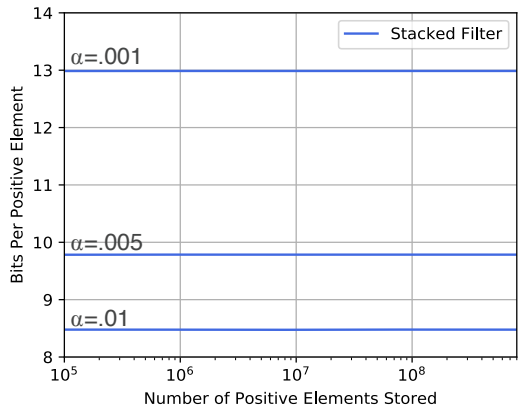

Figure 2: Stacked Filters' Bits Per Element vs Data Size

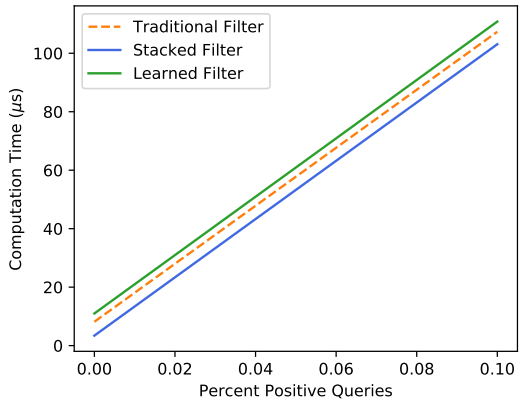

Figure 3: Overall Performance vs Percent of Positive Queries

increases. This is because the number of hash functions used, and corresponding memory accesses incurred, increases linearly with the bits per element.

4.2 Stacked Filters Scale Linearly with Stacked Filters

Traditional filters scale linearly in size with the number of elements keeping the bits per element equal to attain a given FPR. Because Stacked Filters are constructed from these filters and only store low constant factor of the number of elements, they also scale linearly in size with the number of positive and negative elements used in construction. Figure 2 shows this as the bits per element required to achieve a given overall EFPR remains constant while the number of elements stored increases. In this experiment, the proportion of positive and negative elements is held constant as is the value of ψ.

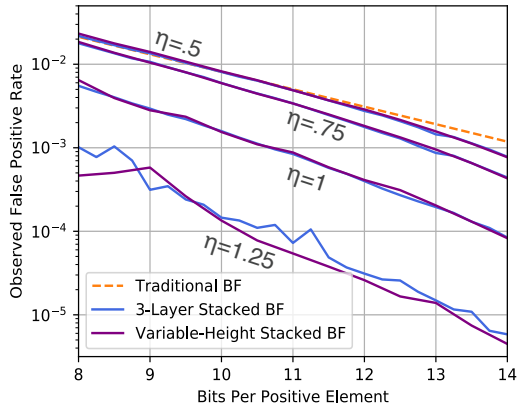

Figure 4: Performance of 3-Layer vs Variable Height Stacked Filters

4.3 Overall Performance Versus Percent of Positives

Figure 3 shows how the overall performance of Stacked, Learned and traditional filters varies when the proportion of positive queries is increased. This is shown in the context of a filter protecting RAID HDD storage, as in the fourth graph of Figure 4 in the main paper.

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

4.4 EFPR Reduction From More than 3 Layers

Figure 4 shows the performance of a 3-layer stacked filter versus a filter whose height is optimized between 1 and 7 layers. In general, the performance is very similar for all except the most extreme skewed distributions. This implies that in many situations 3-layer Stacked Filters capture the majority of the gains found by Stacked Filters.

REFERENCES

[1] S. G. Johnson. Nlopt.

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