PAPER TITLE HERE

Justin Athill, Angela Hillsman, and Daniel Wood

Abstract

In this paper, we present BLANK, a heavy hitter detection algorithm that identifies the k most frequent senders along an ISP backbone link for the purpose of DoS detection. We address the advantages and disadvantages of sampling techniques and counter based methods, and propose an improved algorithm that increases accuracy per space used by combining elements of both. Furthermore, we design BLANK to run on emerging programmable switches. BLANK is prototyped in p4 and evaluated using 3 separate CAIDA datasets from an ISP backbone link. For top-k, we experimentally identify over 98 percent of the top 300 hitters using 4500 counters on a trace containing nearly 3 million packets.

1 Introduction

The Heavy Hitter problem refers to the objective of identifying the heaviest flows in a stream of data. In one variant of the problem, heavy flows are classified as those with a frequency above a threshold t. In this paper, we address a second variant of the problem– "top-k." In this variant, the heavy hitters are the top k flows by frequency.

There are many potential flows that can be analyzed in the context of the Heavy Hitters problem, including source IP addresses, destination IP addresses, transport port numbers, or five-tuples. Depending on the application of the algorithm, a different conception of flow may be appropriate. In this case, we employ our Heavy Hitters algorithm for DoS detection by identifying hosts that are responsible for sending the most traffic through an ISP link. Due to the nature of Internet traffic, a relatively few number of hosts are responsible for sending the majority of packets through a network. In fact, the top 3 percent of hosts may account for over half of the packets traveling through an ISP link in a given time period. Figure 1, a graph of cumulative traffic addressed to/from the top k sources/destinations, reveals two important conclusions about the distribution of traffic. First, there are approximately twice as many destinations as sources, and as a result, these sources account for more traffic on average. Second, for both source addresses and destination addresses, the heaviest hitters are responsible for a highly disproportionate amount of network traffic. Through the rest of this paper, we consider the frequency of packets identified by source IP address as our measure of heavy hitters.

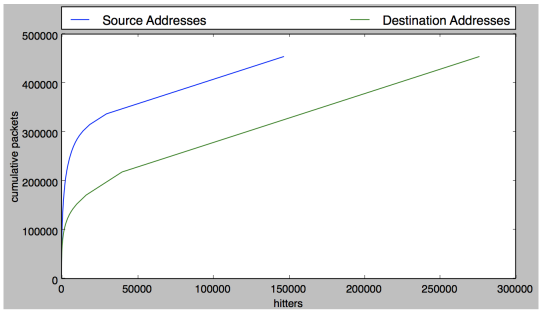

Figure 1: Graph of the cumulative traffic addressed to/from the top k sources/destinations, captured from an ISP backbone link

2 Related Work

2.1 Sampling Algorithms

Rather than count the frequency of each flow, one family of heavy hitter algorithms is based on infrequent sampling. While this approach improves scalability and drastically reduces memory usage, sampling comes at the cost of decreased accuracy. The algorithms Sampled NetFlow [5] and Sample and Hold [3] follow this approach by sampling each packet with some very small probability, such as 0.1 percent or even 0.01 percent. Sample and Hold improves upon Netflow, since once a flow is sampled, a corresponding counter is held in a hash table in flow memory until the end of the measurement interval. The entry for a flow is updated for every subsequent packet belonging to the flow. According to this scheme, the error is proportional to $1/M$, as opposed to $1/M$ for a classical sampling algorithm, where M is the available memory.

2.2 Sketch Algorithms

Sketch algorithms are a family of algorithms that attempt to answer questions about data streams by leveraging approximation. By sacrificing accuracy, sketch algorithms allow the heavy hitter problem to be tackled using a limited amount of memory. One of the most prominent of these algorithms is Count-Min Sketch [2], which uses a two-dimensional hash table to approximate counts. Each packet is hashed using d different hash functions, and the counter in each of the d corresponding buckets is incremented. Hash collisions will cause certain bucket counts to be incremented for different flow identifiers, so to approximate the count for a given identifier, it is hashed using the d hash functions, and the minimum of these bucket counts it used. This method is hardware-friendly, but sketch based algorithms do not track the flow identifiers associated with each count, so reporting the frequency of the top k flows is not accurate.

```
Algorithm 1: Space Saving Algorithm
Table $T$ has $m$ slots, either containing $(key_j, val_j)$
  at slot $j \in \{1,\dots,m\}$, or empty. Incoming packet
  has key $iKey$
if $\exists$ slot $j$ in $T$ with $iKey = key_j$ then
    $val_j \leftarrow val_j + 1$
else
    if $\exists$ empty slot $j$ in $T$ then
        $(key_j, val_j) \leftarrow (iKey, 1)$
    else
        $r \leftarrow \mathrm{argmin}_{j \in \{1,\dots,m\}}(val_j)$
        $(key_r, val_r) \leftarrow (iKey, val_r + 1)$
    end
end
```

2.3 Counter Based Algorithms

Counter Based Algorithms process every packet, but due to memory constraints, are only able to maintain a counter of a constant number of the heaviest flows. Therefore, these algorithms aim to retain counters of only heavy hitters while ignoring lighter flows through the use of strategic admission and eviction policies. Space Saving [4] maintains a table of the frequencies of N flows, evicting the least frequent flow each time an unmonitored flow is encountered. The newly admitted element assumes the frequency of the evicted flow. This eviction policy results in large errors for heavy-tailed workloads, where many new small flows may wrongly evict larger flows. Furthermore, depending on the hardware implementation, it may be resource intensive to find the least frequent flow in the table for every newly encountered flow. The intuition behind Randomized Admission Policy (RAP) [1] is to minimize this error by being much more conservative about the elements that are admitted into the table. Instead of evicting the minimum element in the table for every new flow encountered, RAP only admits new flows with a probability of $1/(c_m + 1)$, where c_m is the minimal counter value. By being more conservative, RAP has increased accuracy over Space Saving, but also makes it more difficult for new larger flows to gain admission. In our approach, we adopt elements of the randomized admission policy to prevent false evictions while also dampening the adverse effects of a conservative policy. HashPipe [6] is heavily inspired by Space Saving, but leverages feed-forward packet processing to divide the task of finding the minimum into small parts. The algorithm consists of d stages, each with its own hash function h_d and an associated hash table. When a packet enters the pipeline at the first stage, the hash function h_0 is used to hash its identifier to a bucket. Each bucket will contain a flow identifier and its associated count. If the identifier of an incoming packet matches the identifier stored in the bucket it hashes to, the count is incremented, and no further processing is done. Similarly, if the bucket is empty, the identifier is added with a count of 1, and processing stops. However, the rest of the pipeline comes into play if the incoming packet identifier does not match the stored packet identifier. In this case, the stored packet will be evicted, and the incoming packet will be stored in its place with a count of 1. In actual implementations of HashPipe on hardware, the identifier and count of the "evicted" packet is added to the incoming packet as metadata, and the original packet continues to flow down the pipeline with this metadata. In subsequent stages, the key-counter pair that has been evicted is hashed using the corresponding hash function h_d and it is compared to the stored value in that bucket. Instead of always evicting like in the first stage, the stored value will only be evicted if it is the minimum be-

Algorithm 2: HashPipe: Pipeline of d hash tables [6]

```
                > Insert in the first stage
$l_1 \leftarrow h_1(iKey)$|
if $key_{l_1} = iKey$ then
    $val_{l_1} \leftarrow val_{l_1} + 1$
    end processing
end
else if $l_1$ is an empty slot then
    $(key_{l_1},\ val_{l_1}) \leftarrow (iKey, 1)$
    end processing
end
else
    $(cKey,\ cVal_j) \leftarrow (key_{l_1},\ val_{l_1})$
    $(key_{l_1},\ val_{l_1}) \leftarrow (iKey,\ 1)$
end
                > Track a rolling minimum
for $i \leftarrow 2$ to $d$ do
    $l \leftarrow h_i(cKey)$
    if $key_l = cKey$ then
        $val_1 \leftarrow val_l + cVal$
        end processing
    end
    else if $l$ is an empty slot then
        $(key_l,\ val_l) \leftarrow (cKey,\ cVal)$
        end processing
    end
    else if $val_l < cVal$ then
        swap $(cKey, cVal)$ with $(key_l, val_l)$
    end
end
```

tween the two counter values. If the stored value is less, the key-counter pairs are swapped, and the key previously in the table becomes the carried key. This same process is repeated at each stage until one pseudo-minimum value is carried off the end.

Instead of attempting to find a true minimum, as is the case in Space Saving, HashPipe settles for a probabilistic minimum, obtained by comparing only one value per stage. The main idea behind the pipeline is that heavy flows will be retained and lighter flows will be evicted over time. HashPipe is useful because for each packet, there is only one read per table. This allows for efficient stream processing and gets around hardware constraints that do not allow multiple reads to the same table. The downside of this scheme is that it does not prevent duplicate keys across different tables. When a count is stored for the same keys in different stages, this reduces the space available to hold onto heavier flows. However, duplicates have been shown to account for only 5-10 percent of table space, and have a limited impact on accuracy [6]. With a fixed amount of memory available to the hardware, the number of stages d can be tuned: a greater number of stages increases heavy flow retention because more slots are sampled to pick a minimum, but it will increase the number of duplicates.

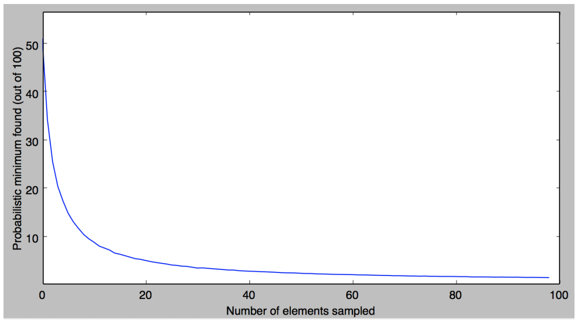

Figure 2: The minimum of a randomly selected set of elements approaches the true minimum

3 Design

3.1 Probabilistic Minimum

Starting with the Space Saving algorithm as a baseline, we first settled on probabilistic sampling as a way to make finding the minimum element more efficient, rather than linearly searching the entire table. As long as a flow with a low enough frequency is evicted when a new flow is encountered, larger flows will still be preserved in the table. As the table below shows, sampling as few as 4 elements probabilistically ensures that only elements in the lowest quintile will be evicted. Drawing from HashPipe, we implemented this optimization using a hash table pipeline, as the number of stages equals the number of elements probabilistically sampled. If there are four stages in the pipeline, the minimum of the four flows encountered (one per stage) will be a candidate for eviction.

3.2 Randomized Assignment

HashPipe, as the name suggests, features a consistent hashing scheme whereby each stage uses an independent hash function and subsequent repeated flows consistently hash to the same location within each stage. This scheme is intended to reduce the number of duplicate flows occupying space in the pipeline and consolidate their counts. However, we experimentally determined that using a random function to determine flow assignment within the first stage resulted in improvements in accuracy of nearly 50 percent.

3.3 Logarithmic Admission Policy

One of the major flaws with HashPipe is the fact that, as with Space Saving, every flow will always be admitted into the pipeline, even if it will never reoccur. In streams with many small flows, accuracy suffers as heavy hitters near the minimum threshold are evicted by small newly encountered flows. Therefore, we introduced a modified admission policy inspired by RAP, but instead of making the probability of admission inversely proportional to the frequency of the minimum counter, we used a log function to dampen extremely low admission probabilities. We introduced a log function to counteract one of the systematic errors with RAP: denying admission to new heavy hitters. This way, flows with high frequencies are still protected, but smaller flows have a larger probability of being evicted. We tested two potential solutions to this problem, both of which restrict admission to the table.

In Front Rejection (AKA The Bouncer), packets are randomly assigned a slot in the first stage. If empty, the packet is inserted and the frequency is set to 1. If the slot contains a packet with the identical source IP address, the frequency is incremented. Otherwise, the packet evicts the resident of the slot with probability $p = 1/(5 * log(c_m + 1))$, where c_m is the frequency count of the flow in the slot. This is easy to implement, and it saves a lot of computational resources further down the pipeline, since on average 90 percent of packets will be refused admission to the pipeline. However, it is not necessarily accurate because the probability of admission is determined by only the flow randomly compared to in the first stage of the pipeline. This flow is not guaranteed to the be the minimum, but it is likely to be an average flow, which turns out to be an adequate compromise.

In Back Rejection (AKA The Interview), all packets are admitted to the entrance of the pipeline and proceed through all stages of the pipeline. However, rather than always evicting the minimum of the flows encountered throughout the pipeline to make way for the new flow, a calculation is made at the end of the pipeline. With probability $p = 1/(5 * log(c_m + 1))$, the minimum flow is evicted, otherwise the minimum flow is inserted back into the first stage, retroactively denying admission to the newly encountered flow located in the first stage.

Algorithm 3: HashPipe Bouncer Admission

```
            > Randomly position in first stage
$l_1 \leftarrow rand_1(iKey)$
if $key_{l1}$ = $iKey$ then
    $val_{l1} \leftarrow val_{l1} + 1$
    end processing
end
else if $l_1$ is an empty slot then
    $(key_{l1}, val_{l1}) \leftarrow (iKey,1)$
    end processing
end
else
            > Randomized Admission Policy
    $m \leftarrow val_{l1}$
    if $random() < \frac{1}{5*\log(m+1)}$ then
            $(cKey, cVal_j) \leftarrow (key_{l1}, val_{l1})$
            $(key_{l1}, val_{l1}) \leftarrow (iKey, 1)$
    end
    else
            end processing
end
            > Track a rolling minimum
for $i \leftarrow 2$ to $d$ do
    $l \leftarrow h_i(cKey)$
    if $key_l$ = $cKey$ then
            $val_1 \leftarrow val_l + cVal$
            end processing
    end
    else if $l$ is an empty slot then
            $(key_l, val_l) \leftarrow (cKey, cVal)$
            end processing
    end
    else if $val_l < cVal$ then
            swap $(cKey,cVal)$ with $(key_l,val_l)$
    end
end
```

4 Evaluation

We evaluate the accuracy of the variants of our algorithm through a series of simulations in which we fine tune the various parameters: number of stages, memory size, hash functions vs. random assignment, and probabilistic admission coefficients. In order to simulate realistic streams of traffic, we ran testing using three different traces from the equinox chicago ISP backbone link, recorded in 2016. These anonymized traces each contain between 20 - 40 million packets, over 1 million different flows, and range from 40 minutes to 1 hour long. The data was obtained with permission from the Center for Applied Internet Data Analysis (CAIDA). We parsed the data from the CAIDA traces to isolate only the source IP addresses. Each source IP address is considered a separate flow.

4.1 Accuracy Metrics

Through measuring the false negative rate when testing our algorithms against the CAIDA data, we attempted to experimentally determine which combination of design policies yielded the best results.

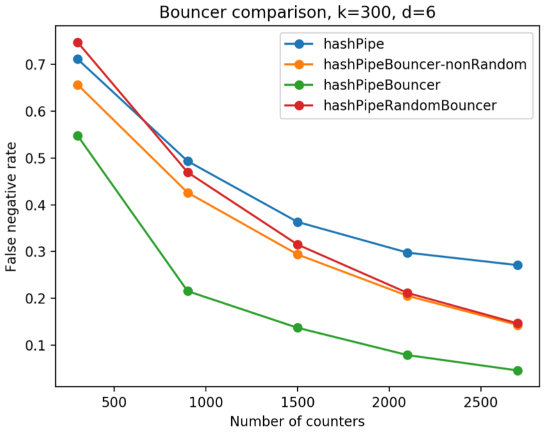

Figure 3: Comparison of Bouncer admission policy algorithm with different combinations of randomization and hashing at each stage. In hashPipeBouncer-nonRandom, consistent hash functions are used in every stage, and no randomnes is applied. The hashPipeBouncer algorithm, only applies randomness in the first stage–should an incoming packet be admitted to the table, it is placed at a random index rather than using a hash function. This variation performed the best across all tested memory sizes and is the main Bouncer algorithm we use in further comparison. The hashPipeRandomBouncer algorithm does comparisons and inserts at random indices in all table stages, and was also shown to be outperformed by limiting randomization to the first stage.

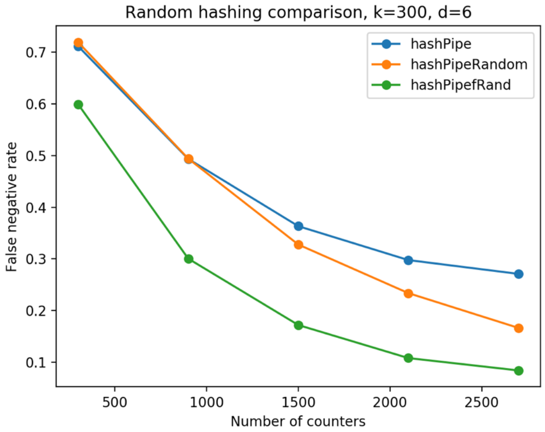

Figure 4: Impact of random assignment rather than using hash functions. The hashPipeRandom algorithm randomizing indices in all stages and provides some benefits at high memory sizes. The hashPipefRand algorithm assigns a random index in the first stage and then uses consistent hash functions in the following stages. Randomization in the first stage improves accuracy rates across the entire tested memory range, showing that randomization is useful absent any strict admission policy.

We focused on testing the two different admission policies, and applied different levels of randomized slot assignment to find an optimal algorithm. When applying front rejection, we found that uniformly randomizing the table index in the first stage only yielded the best results, and gave as much as a 50 percent accuracy improvement over simple front rejection with normal hashing (see Figure 3). While some randomization provided improvements for front rejection, our tests of back rejection showed that normal hashing at all stages was best (see Figure 4).

Why is there a discrepancy in the effectiveness of random slot assignment? Perhaps this can be explained by the increased level of eviction that occurs in the first stage when random indices are used. When slot assignments are randomized, it is less likely that the same key will end up at the same index and increase its count in the first stage. So effectively, keys are more quickly pushed to later stages where there is a greater emphasis on retaining heavy flows. Since consistent hashing is performed in later stages, the same key is able to add to its count, and can do this more frequently when it is quickly sent down the pipeline from the first stage. Randomized slot assignment more effectively treats the first stage as transient, so light flows will more quickly be evicted and heavy flows will more quickly be retained in the core of the pipeline. For this reason, randomizing indices in all stages is less effective because retention also is lower in the latter stages, when it is most important to hold onto heavy flows. Randomization in the first stage adds an accuracy boost to front rejection, but it introduces problems with the more complex back rejection policy. The goal with the Interview policy, is to deny first stage admission to the incoming key at a high rate, and replace it with the probabilistic minimum that emerges from the pipeline. However, when randomization is applied, we can't guarantee that we are swapping the minimum with the incoming key in the first stage. This renders our stricter admission policy ineffective if we can't guarantee that the incoming key is the one

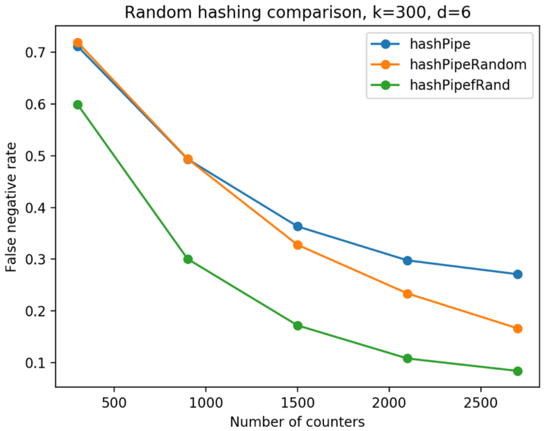

Figure 5: Impact of random slot assignment rather than using hash functions. The hashPipeRandom algorithm randomizing indices in all stages and provides some benefits at high memory sizes. The hashPipefRand algorithm only hashes to a random index in the first stage and then uses consistent hash functions in the following stages. Randomization in the first stage improves accuracy rates across the entire tested memory range, showing that randomization is useful absent any strict admission policy.

being evicted in the first stage. Evidently, randomized slot assignment in the first stage can effectively increase accuracy under the right conditions, and was even shown to be useful as a standalone improvement to HashPipe without applying front or back rejection (see Figure 5). However, front rejection does not benefit from randomized slot assignment, and the standalone version of the policy ended up with the best performance. In all further testing when randomization is applied, we run multiple trials on each dataset and report average values.

We were also able to tune our algorithm's performance by experimenting with the multiplicative factor in our logarithmic admission equation. Figure 6 shows how performance improved up to a log factor value of 5, and experienced diminishing returns thereafter. We settled on using a log factor of 5 for both front and back rejection admission policies for all further testing.

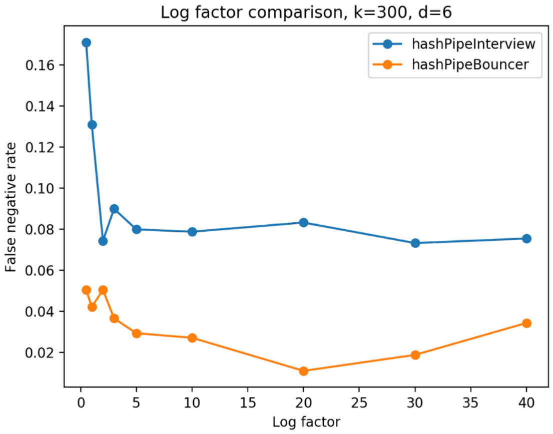

Figure 6: Impact of varying the factor f when calculating the admission threshold $p = 1/(f * log(c_m + 1))$. Results show that factors of 5 and greater experienced relatively similar accuracy rates for both front rejection and back rejection.

4.2 Comparison of HashPipe Implementations

We began algorithm comparison by varying the available memory size, which corresponds to a greater number of counters that can be stored across all hash tables. Figure 7 shows our results when searching for the top 300 flows and using 6 table stages. Both algorithms that applied an admission policy outperformed the standard HashPipe algorithm for all tested memory sizes. Accuracy rates improved by about 20 percent when the number of counters was limited to 300, equal to the number of top-k flows being identified, and increases to more than 50 percent improvement when more than 2000 counters are used. HashPipe with front rejection in particular brought its accuracy rate to more than 90 percent with 2000 counters, which is less than half the memory required by the standard HashPipe algorithm to achieve that accuracy with 300 heavy hitters and 6 table stages. In all algorithms, improvement begins to diminish after 1000 counters are available.

In addition to testing the impact of varying memory, we also experimented with the number of pipeline stages available. HashPipe tends to perform best with a limited amount of stages so that the number of duplicates is limited. The optimal number of stages is about 6 with the standard Hash-

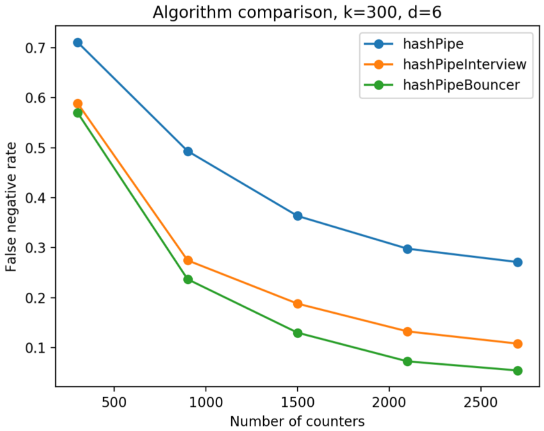

Figure 7: Comparison of false negatives of Interview and Bouncer to baseline. Both algorithm optimizations improve on the standard HashPipe algorithm over the entire tested memory range

Pipe, but Figure 8 shows that our algorithms continue to improve past this point as the number of table stages is increased. While Interview and Bouncer also experience diminishing returns after about 8 table stages, they do not see the same performance decrease present in the standard HashPipe algorithm.

4.3 Space Saving Comparison

As a final benchmark, we compare our algorithms to the Space Saving algorithm, which finds the true minimum of all of its counters. While this algorithm allows for more accuracy, hardware constraints that restrict multiple reads to the same table make implementations of this algorithm unrealistic. Figure X shows that our algorithms fall behind an idealized implementation of Space Saving, where it is no issue to compute a global minimum. Similar to the baseline implementation of HashPipe, our implementations also outperform Space Saving when a low amount of memory is available. This is because Space Saving is only guaranteed to hold onto the kth heaviest item when it is larger than average count in the table, so its full benefits are realized when memory is increased

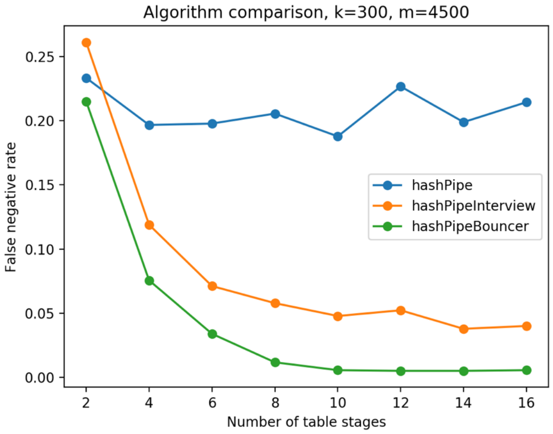

Figure 8: Comparison of algorithms when number of table stages is varied. Interview and Bouncer do not suffer from the same accuracy drawbacks as the baseline when increasing the number of table stages

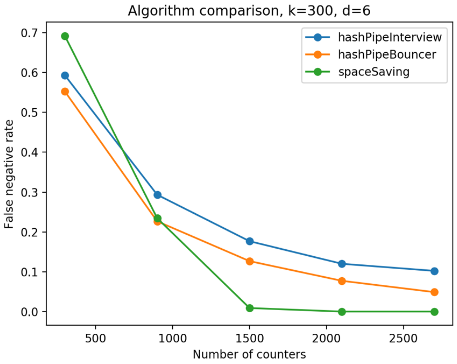

Figure 9: Comparison of Interview and Bouncer with Space Saving

5 Conclusions

References

[1] E. G. F. R. K. Y. Ben Basat, R. Randomized admission policy for efficient top-k and frequency estimation. In *IEEE INFOCOM 2017 Conference on Computer Communications*, Atlanta, GA, 2017.

[2] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 4 2003.

[3] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the

elephants, ignoring the mice. *ACM Transactions on Computer Systems*, 21(3):270–313, 8 2003.

[4] A. D. Metwally, A. and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. *International Conference on Database Theory*, 3363(1), 1 2005.

[5] C. Netflow. Netflow. https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[6] N. S. R. O. M. S. Sivaraman, V. and J. Rexford. Heavy-hitter detection entirely in the data plane. In *Proceedings of ACM Symposium on SDN Research 2017*, Santa Clara, CA, Apr. 2017.