PAPER TITLE HERE

Justin Athill, Angela Hillsman, and Daniel Wood

Abstract

ABSTRACT HERE

1 Introduction

The Heavy Hitter problem refers to 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 the 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 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.

Example of incorporation citations [1].

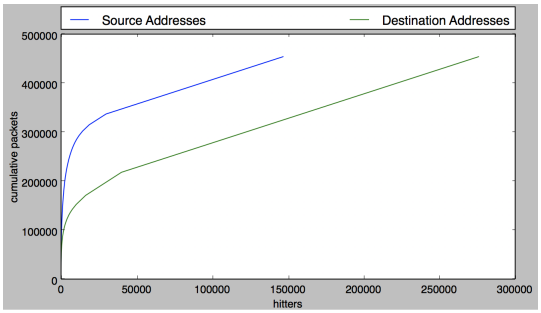

Figure 1: Graph of the cumulative traffic addressed to/from the top k sources/destinations, captured from an ISP backbone link

2 Related Work

2.1 Sampling Algorithms

Rather than count the frequency of each packet, 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 and Sample and Hold 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/Mfor 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, 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

```
1  Table T has m slots, either containing (key_j, val_j)
     at slot j ∈{1,…,m}, or empty. Incoming packet
     has key iKey
2  if ∃ slot j in T with iKey = key_j then
3    val_j ← val_j + 1
4  else
5    if ∃ empty slot j in T then
6         (key_j, val_j) ← (iKey, 1)
7    else
8         r ← argmin_{j ∈ {1,…,m}}(val_j)
9         (key_r, val_r) ← (iKey, val_r + 1)
10   end
11 end
```

2.3 Counter Based Algorithms

Counter Based Algorithms process every flow, 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 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) 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 Saver, 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 is heavily inspired by Space Saver, 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 hd and an associated hash table. When a packet enters the pipeline at the first stage, the hash function h0 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 âĂIJevictedâĂİ 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 hd 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 between 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 Saver, HashPipe settles for a probabilistic minimum, obtained by comparing only one

```
Algorithm 2: HashPipe: Pipeline of d hash tables
1                    > Insert in the first stage
2  l1 ← h1(iKey)
3  if key_l1 = iKey then
4     val_l1 ← val_l1 + 1
5     end processing
6  end
7  else if l1 is an empty slot then
8     (key_l1, val_l1) ← (iKey,1)
9     end processing
10 end
11 else
12    (cKey, cVal_j) ← (key_l1, val_l1)
13    (key_l1, val_l1) ← (iKey, 1)
14 end
15                   > Track a rolling minimum
16 for i ← 2 to d do
17    l ← h_i(cKey)
18    if key_l = cKey then
19          val1 ← val_l + cVal
20          end processing
21    end
22    else if l is an empty slot then
23          (key_l, val_l) ← (cKey, cVal)
24          end processing
25    end
26    else if val_l < cVal then
27          swap (cKey,cVal) with (key_l,val_l)
28    end
29 end
```

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 [CITATION]. 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.

3 Design

Probabilistic Minimum Starting with the Space Saver algorithm as a baseline, one of the first improvements we made was locating a probabilistic minimum using sampling, rather than linearly searching through the entire table to find the minimum. Furthermore, as long as a flow with a low

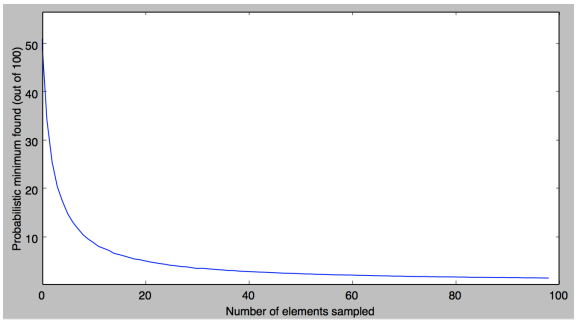

Figure 2: The minimum of a randomly selected set of elements approaches the true minimum

enough frequency is evicted, 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. This optimization is implemented in the hash table pipeline, as the number of stages equals the number of elements probabilistically sampled. If there are 4 stages in the pipeline, the minimum of the four flows encountered (one per stage) will be a candidate for eviction.

3.1 Randomized "Hashing"

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 50Logarithmic Admission Policy

One of the major flaws with HashPipe is the fact that, as with Space Saver, every flow will always be admitted into the pipeline, even if they will never occur again. In streams with many small flows (normal Internet traffic), 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 only admitted to the entrance of the pipeline with probability $p = 1/(5 * log(c_m + 1))$, where c_m is the frequency count of a randomly sampled flow in the first stage of the pipeline. This is easy to implement, and it saves a lot of computational resources since over 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 hashed 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, effectively denying permanent admission of the new flow into the pipeline.

Algorithm 3: HashPipe: Pipeline of d hash tables

```
1          > Randomly position in first stage
2  l1 ← rand1(iKey)
3  if key_l1 = iKey then
4     val_l1 ← val_l1 + 1
5     end processing
6  end
7  else if l1 is an empty slot then
8     (key_l1, val_l1) ← (iKey,1)
9     end processing
10 end
11 else
12         > Randomized Admission Policy
13    m ← val_l1
14    if random() < 1/(5*log(m+1)) then
15          (cKey, cVal_j) ← (key_l1, val_l1)
16          (key_l1, val_l1) ← (iKey, 1)
17    end
18    else
19          end processing
20 end
21         > Track a rolling minimum
22 for i ← 2 to d do
23    l ← h_i(cKey)
24    if key_l = cKey then
25          val_1 ← val_l + cVal
26          end processing
27    end
28    else if l is an empty slot then
29          (key_l, val_l) ← (cKey, cVal)
30          end processing
31    end
32    else if val_l < cVal then
33          swap (cKey,cVal) with (key_l,val_l)
34    end
35 end
```

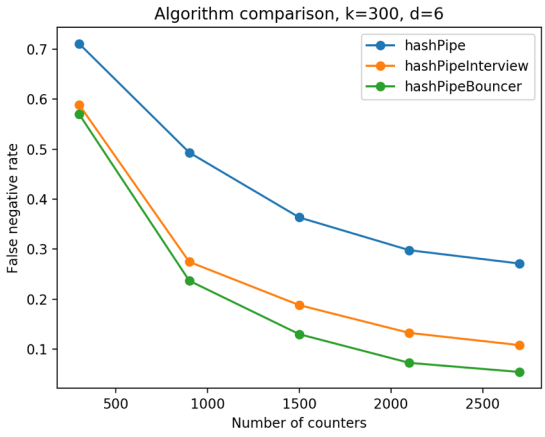

Figure 3: Comparison of false negatives of Interview and Bouncer to baseline. Both algorithm optimizations improve on the standard HashPipe algorithm over the entire tested memory range

4 Evaluation

5 Conclusions

References

[1] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.