Big Data Methods for Efficient Network Monitoring

Ran Ben Basat
Big Data Methods for Efficient Network Monitoring

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Ran Ben Basat

Submitted to the Senate of the Technion — Israel Institute of Technology
Tamuz 5778 Haifa July 2018
This research was carried out under the supervision of Prof. Roy Friedman, in the Faculty of Computer Science. Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:


Acknowledgments

The generous financial help of the Hasso Plattner Institute and the Technion are gratefully acknowledged.
# Contents

| List of Figures | iii |
| List of Tables | v |

## 1 Introduction

1.1 Background ........................................ 3
1.2 Related Work ....................................... 6
   1.2.1 Sliding Windows .................................. 6
   1.2.2 Frequency Estimation and Heavy Hitters .......... 7
   1.2.3 Hierarchical Heavy Hitters ...................... 9
1.3 Thesis Outline ..................................... 9

## 2 Preliminaries ...................................... 11

## 3 Summing over Sliding Windows ...................... 13

3.1 Introduction ....................................... 13
3.2 Related Work ....................................... 16
3.3 Definitions ........................................ 17
3.4 Lower Bounds ...................................... 17
3.5 Upper Bounds ...................................... 22
   3.5.1 Large $\epsilon$ values ......................... 22
   3.5.2 Small $\epsilon$ Values ......................... 27
3.6 Discussion ......................................... 30

## 4 Randomized Admission Policies ..................... 31

4.1 Introduction ....................................... 31
4.1.1 Background ...................................... 31
4.2 Related Work ....................................... 33
   4.2.1 Counter based algorithms ...................... 33
   4.2.2 Sketch based algorithms ...................... 33
   4.2.3 Network monitoring architectures ............. 34
4.3 Randomized Admission Policy (RAP) ................ 35
   4.3.1 Analysis ...................................... 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Hardware Friendliness</td>
<td>37</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Cache Memory Organization</td>
<td>38</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Cache Policy</td>
<td>38</td>
</tr>
<tr>
<td>4.4.3</td>
<td>dW-RAP as a Cache Policy</td>
<td>39</td>
</tr>
<tr>
<td>4.5</td>
<td>Evaluation</td>
<td>40</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Datasets</td>
<td>40</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Metrics</td>
<td>40</td>
</tr>
<tr>
<td>4.5.3</td>
<td>On Arrival Evaluation</td>
<td>41</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Top-k Identification</td>
<td>44</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Limited Associativity Impact</td>
<td>47</td>
</tr>
<tr>
<td>4.6</td>
<td>Theoretical Guarantees for Top-k Identification Using Randomized Admission Policy</td>
<td>48</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Conditions for Successful Space Saving Top-k Identification</td>
<td>49</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Conditions for Successful RAP’ Top-k Identification</td>
<td>51</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Performance Comparison for Zipf Distributed Streams</td>
<td>53</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Section Summary</td>
<td>56</td>
</tr>
<tr>
<td>4.7</td>
<td>Conclusion and Discussion</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>Fast Flow Volume Estimation</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>59</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Background</td>
<td>59</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Contributions</td>
<td>60</td>
</tr>
<tr>
<td>5.2</td>
<td>Related Work</td>
<td>61</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Streams</td>
<td>61</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Sliding Windows</td>
<td>62</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Hierarchical Heavy Hitters</td>
<td>62</td>
</tr>
<tr>
<td>5.3</td>
<td>Preliminaries</td>
<td>63</td>
</tr>
<tr>
<td>5.4</td>
<td>Frequent items Algorithm with a Semi-structured Table (FAST)</td>
<td>64</td>
</tr>
<tr>
<td>5.4.1</td>
<td>FAST - Accurate Description</td>
<td>65</td>
</tr>
<tr>
<td>5.4.2</td>
<td>FAST Analysis</td>
<td>65</td>
</tr>
<tr>
<td>5.5</td>
<td>Windowed FAST (WFAST)</td>
<td>70</td>
</tr>
<tr>
<td>5.5.1</td>
<td>WFAST Analysis</td>
<td>71</td>
</tr>
<tr>
<td>5.6</td>
<td>Hierarchical Heavy Hitters</td>
<td>75</td>
</tr>
<tr>
<td>5.7</td>
<td>Evaluation</td>
<td>79</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Datasets</td>
<td>79</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Effect of γ on Runtime</td>
<td>80</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Speed vs. Space Tradeoff</td>
<td>80</td>
</tr>
<tr>
<td>5.7.4</td>
<td>Operation Speed Comparison</td>
<td>81</td>
</tr>
<tr>
<td>5.7.5</td>
<td>Sliding Window</td>
<td>81</td>
</tr>
<tr>
<td>5.7.6</td>
<td>Hierarchical Heavy Hitters</td>
<td>82</td>
</tr>
</tbody>
</table>
## Hierarchical Heavy Hitters

### 6.1 Background

### 6.2 Contributions

### 6.3 Related Work

### 6.4 Randomized HHH (RHHH)

#### 6.4.1 Basic terminology

#### 6.4.2 Randomized HHH

### 6.5 Evaluation

#### 6.5.1 Accuracy and Coverage Errors

#### 6.5.2 False Positives

#### 6.5.3 Operation Speed

### 6.6 Virtual Switch Integration

#### 6.6.1 Open vSwitch Overview

#### 6.6.2 Open vSwitch Evaluation

### 6.7 Analysis

#### 6.7.1 Accuracy Analysis

#### 6.7.2 Coverage Analysis

#### 6.7.3 RHHH Properties Analysis

### 6.8 Discussion

## Additional Sliding Window Measurements

### 7.1 Background

#### 7.1.1 Contribution

#### 7.1.2 Chapter organization

### 7.2 Related work

#### 7.2.1 Set Membership and Counting

#### 7.2.2 Count Distinct

#### 7.2.3 Entropy Detection

#### 7.2.4 Preliminaries – Compact Set Multiplicity

### 7.3 SWAMP Algorithm

#### 7.3.1 Model

#### 7.3.2 Problems definitions

#### 7.3.3 SWAMP algorithm

### 7.4 Empirical Evaluation

#### 7.4.1 Overview

#### 7.4.2 Evaluation of analytical guarantees

#### 7.4.3 Set membership on sliding windows

#### 7.4.4 Per-flow counting on sliding windows

#### 7.4.5 Count distinct on sliding windows
7.4.6 Entropy estimation on sliding window ........................................ 120
7.5 Analysis of SWAMP .................................................................... 121
  7.5.1 Analysis of per flow counting .................................................. 121
  7.5.2 Analysis of count distinct functionality ..................................... 123
  7.5.3 Analysis of entropy estimation functionality ............................... 128
7.6 Discussion .................................................................................. 130

8 Conclusions .................................................................................. 133
  8.1 Summing over Sliding Windows .................................................. 133
  8.2 Randomized Admission Policies ................................................ 133
  8.3 Weighted Heavy Hitters ............................................................... 134
  8.4 Hierarchical Heavy Hitters ......................................................... 134
  8.5 All in One Sliding Bloom Filter .................................................. 134

Bibliography .................................................................................... 135

Hebrew Abstract
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Comparison of SRAM and DRAM Memory</td>
</tr>
<tr>
<td>1.2</td>
<td>Relations between the discussed problems. Each line goes from a base</td>
</tr>
<tr>
<td></td>
<td>problem to a generalized one. The gray boxes represent related works</td>
</tr>
<tr>
<td></td>
<td>while our projects focus on the green ones.</td>
</tr>
<tr>
<td>3.1</td>
<td>The setting for the proof of Theorem 3.5.1.</td>
</tr>
<tr>
<td>4.1</td>
<td>A 4-way set associative cache with 8 lines. When an item (x) arrives,</td>
</tr>
<tr>
<td></td>
<td>Set(x) = 2 is calculated and the item can be stored in any of the ways</td>
</tr>
<tr>
<td></td>
<td>of set 2.</td>
</tr>
<tr>
<td>4.2</td>
<td>A 4-way set associative cache with RAP policy. Item x has Set(x) = 2.</td>
</tr>
<tr>
<td></td>
<td>Set 2 includes r, w, c and z, while x is not in the cache. The eviction</td>
</tr>
<tr>
<td></td>
<td>policy selects c, because its frequency is the smallest. x will be</td>
</tr>
<tr>
<td></td>
<td>admitted into the cache with probability $\frac{1}{8}$. If x is</td>
</tr>
<tr>
<td></td>
<td>admitted, its counter will be incremented to 8.</td>
</tr>
<tr>
<td>4.3</td>
<td>On Arrival – Mean square error vs. number of counters. Note that CMS</td>
</tr>
<tr>
<td></td>
<td>and CS have 8 times as many counters!</td>
</tr>
<tr>
<td>4.4</td>
<td>The recall for finding the top-32 elements vs. number of counters.</td>
</tr>
<tr>
<td>4.5</td>
<td>The average recall achieved for identifying the top-512 using 1024</td>
</tr>
<tr>
<td></td>
<td>counters, compared to the size of the stream.</td>
</tr>
<tr>
<td>4.6</td>
<td>The Precision-Recall curves for identifying the top-512 with 1024</td>
</tr>
<tr>
<td></td>
<td>counters.</td>
</tr>
<tr>
<td>4.7</td>
<td>Comparison of the performance of d-Way RAP for different associativity</td>
</tr>
<tr>
<td></td>
<td>levels.</td>
</tr>
<tr>
<td>4.8</td>
<td>The counters allocated for the top-k elements are increased with rates</td>
</tr>
<tr>
<td></td>
<td>$f_1, \ldots, f_k$. The increase rate of the minimal tail counter is</td>
</tr>
<tr>
<td></td>
<td>at most $\frac{1}{m-k}$.</td>
</tr>
<tr>
<td>4.9</td>
<td>The probability of moving to a larger index state is $\lambda$ and is</td>
</tr>
<tr>
<td></td>
<td>not dependent on the process made so far or the current state index.</td>
</tr>
<tr>
<td>5.1</td>
<td>An example of how FAST utilizes the SOS structure. Here, flows are</td>
</tr>
<tr>
<td></td>
<td>partially ordered according to the third digit (100’s), and each flow</td>
</tr>
<tr>
<td></td>
<td>maintains its own remainder; e.g., the estimated volume of $D$ is $\hat{v}_D = 583$.</td>
</tr>
</tbody>
</table>
5.2 The stream is divided into intervals of size $W$ called *frames* and each frame is partitioned into $k$ equal-sized *blocks*. The window of interest is also of size $W$, and overlaps with at most 2 frames and $k+1$ blocks.

5.3 The effect of parameter $\gamma$ on operation speed for different error guarantees ($\epsilon$). $\gamma$ influences the space requirement as the algorithm is allocated with \[ \left\lceil \frac{1+\gamma}{\epsilon} \right\rceil \] counters.

5.4 Runtime comparison for a given error guarantee ($\epsilon = 2^{-8}$). All algorithms provide the same guarantees and FAST uses different $\gamma$ values to show the speedup gained from allocating additional counters.

5.5 Runtime as a function of accuracy guarantee ($\epsilon$) provided by the algorithms.

5.6 Space overheads of WFAST compared to previous works. Note that WFAST operates in constant time while the other algorithm requires linear scanning of all counters.

5.7 WFAST with varying window sizes ($\epsilon = 2^{-8}$) and varying $\epsilon$ ($W = 2^m$).

5.8 Runtime of of HHHH algorithms as a function of their accuracy guarantee ($\epsilon$).

6.1 A high level overview of this work. Previous algorithms’ update requires $\Omega(H)$ run time, while we perform at most a single $O(1)$ update.

6.2 An example of the lattice induced by a two dimensional source/destination byte hierarchy. The top left corner (*,*) is fully general while the bottom right (s1.s2,s3.s4,d1.d2,d3.d4) is fully specified. The parents of each node are directly above it and directly to the left.

6.3 Accuracy error ratio – HHH candidates whose frequency estimation error is larger than $N\epsilon$ ($\epsilon = 0.001$).

6.4 The percentage of Coverage errors – elements $q$ such that $q \not\in P$ and $C_{q|P} \geq N\theta$ (false negatives).

6.5 False Positive Rate for different stream lengths.

6.6 Update speed comparison for different hierarchical structures and workloads.

6.7 Throughput of dataplane implementations ($\epsilon = 0.001, \delta = 0.001, 2D$ Bytes, Chicago 16).

6.8 Measured throughput in both dataplane and distributed implementations.

6.1 A high level overview of this work. Previous algorithms’ update requires $\Omega(H)$ run time, while we perform at most a single $O(1)$ update.

6.2 An example of the lattice induced by a two dimensional source/destination byte hierarchy. The top left corner (*,*) is fully general while the bottom right (s1.s2,s3.s4,d1.d2,d3.d4) is fully specified. The parents of each node are directly above it and directly to the left.

6.3 Accuracy error ratio – HHH candidates whose frequency estimation error is larger than $N\epsilon$ ($\epsilon = 0.001$).

6.4 The percentage of Coverage errors – elements $q$ such that $q \not\in P$ and $C_{q|P} \geq N\theta$ (false negatives).

6.5 False Positive Rate for different stream lengths.

6.6 Update speed comparison for different hierarchical structures and workloads.

6.7 Throughput of dataplane implementations ($\epsilon = 0.001, \delta = 0.001, 2D$ Bytes, Chicago 16).

6.8 Measured throughput in both dataplane and distributed implementations.
7.4 Space/accuracy trade-off in estimating per flow packet counts (window size: $W = 2^{16}$). ........................................... 118
7.5 Accuracy of SWAMP’s count distinct functionality compared to alternatives. .......................................................... 119
7.6 Accuracy of SWAMP’s entropy functionality compared to the theoretical guarantee. .................................................. 120
## List of Tables

3.1 Comparison of **BASIC-SUMMING Algorithms**. . . . . . . . . . . . . . . . . 15  
3.2 Summary of our results for a $RW\epsilon$-approximation of the **BASIC-SUMMING problem**. 15  
5.1 List of Symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64  
5.2 Variables used by the WFAST algorithm. . . . . . . . . . . . . . . . . . . . 71  
5.3 A summary of key characteristics of the real Internet traces used in this work. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79  
6.1 List of Symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88  
7.1 Summary of SWAMP’s accuracy guarantees. . . . . . . . . . . . . . . . . 110  
7.2 Comparison of sliding window set membership algorithms for $\epsilon = W^{-o(1)}$. 111  
7.3 List of symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
Abstract

Streaming algorithms are used nowadays for a large variety of applications such as network monitoring, identifying social networks influencers, and databases. These applications share the need to handle a massive data inflow, which requires algorithms to process new data exceptionally efficiently. Additionally, the data stream is often too large to be stored in memory, thus algorithms can only save a small representation of the stream, known as a sketch.

This thesis improves the speed and space requirements for streaming problems that are fundamental to networking. Our first contribution is efficient algorithms for summing over a sliding window on an integer stream, with an additive error. Our algorithms are succinct - space optimal up to a $1 + o(1)$ multiplicative factor from our lower bound.

Our second contribution is an efficient algorithm for identifying the most frequent elements in a data stream. Existing algorithms exhibit poor performance when applied to networking data. We introduce Randomized Admission Policy (RAP) – a novel solution for the frequency and top-k estimation problems. We demonstrate space reductions compared to the alternatives by a factor of up to 32 on real packet traces and up to 128 on heavy-tailed workloads. For top-k identification, RAP exhibits memory savings by a factor of between 4 and 64 depending on the workloads' skewness. Additionally, we present d-Way RAP, a hardware-friendly variant of RAP that empirically maintains its space and accuracy benefits.

The increasing popularity of jumbo frames means growing variance in the size of packets transmitted in modern networks. Our third contribution is developing constant time algorithms for volume estimations in streams and sliding windows, which are faster than previous work. Our solutions are formally analyzed and are extensively evaluated over multiple real-world packet traces as well as synthetic ones. For streams, we demonstrate a run-time improvement of up to 2.4X compared to state of the art. On sliding windows, we exhibit a memory reduction of over 100X on all traces and an asymptotic runtime improvement to a constant.

Monitoring tasks, such as anomaly and DDoS detection, require identifying frequent flow aggregates based on common IP prefixes. These are known as hierarchical heavy hitters (HHH). The per-packet complexity of existing HHH algorithms is proportional to the size of the hierarchy, imposing significant overheads. The next contribution is a randomized constant time algorithm for HHH. Using four real Internet packet traces,
we demonstrate that our algorithm indeed obtains comparable accuracy and recall as previous works, while running up to 62 times faster. Finally, we extended Open vSwitch (OVS) with our algorithm and showed it could handle 13.8 million packets per second. In contrast, incorporating previous works in OVS only obtained 2.5 times lower throughput.

Our last contribution revisits the sliding window regime and considers additional measurement tasks. Various capabilities, such as DDoS detection and load balancing, require insights about multiple metrics including Bloom filters, per-flow counting, count distinct and entropy estimation. Our work presents a unified construction that efficiently solves all the above problems in the sliding window model.
Chapter 1

Introduction

1.1 Background

The ability to monitor and analyze a massive stream of data is a fundamental enabler for a vast range of applications. In this setting, data keeps arriving and the algorithm is required to update its data structure to reflect the current stream. For example, consider monitoring the traffic routed through a backbone switch in a datacenter. The goal of such an algorithm may be to measure the traffic volume passed in the last hour, decide whether an arriving packet belongs to a new connection, or to identify the most frequent flows.

Networking devices typically collect traffic statistics that can be used for detecting denial of service (DoS) attacks, providing efficient load balancing, and identifying traffic anomalies. For discovering DoS attacks for example, it is important to assess whether the incoming packet comes from an address which transmits significantly more messages than expected.

Applications for these algorithms also appear in algorithmic trading, where decisions are often based on a moving average of the stock price. In databases, heavy hitters algorithms are often used for finding frequent elements in excessively large tables, while in social networks they are used for finding influential users.

For all of the examples above, counter based algorithms are a popular solution for tackling the problems. If we can count the number of packets associated with each flow, answering queries about its frequency becomes straightforward. Nevertheless, it is often infeasible to allocate a counter to every potential element in the stream.

Across the multiple domains in which they are used, a prime requirement of these algorithms is to keep up with the rate of incoming data, also known as line speed. In networking devices, the scarce and expensive SRAM memory allows reading and writing counters in line speed, while the slower DRAM cannot be accessed online. Some solutions require periodic synchronization (e.g., keeping the most significant bits in DRAM) between the two, thereby keeping only a small number of bits per flow tracked in SRAM. These solutions, however, do not allow real-time queries as they need to read from DRAM.
Further, it is sometimes the case where the number of monitored flows is so large, that we cannot even save a single bit per flow in SRAM. Lastly, even in cases where one can fit millions of counters on a memory fast enough to work at line speed, answering queries using that many counters may prove difficult. Our solutions focuses on minimizing the number of counters needed, thereby allowing them to monitor a large number of flows using only SRAM. The problem is illustrated in Figure 1.1.

Consider the simplest counting problem, where it is required to count the number of ones in a binary stream consisting of 1’s and 0’s. This problem seems straight forward, and is solvable using $O(\log z)$ memory, where $z$ is the number of ones. Previous works have suggested replacing the deterministic counting with randomized algorithm that only uses $O(\log \log z)$ bits and provides decent estimation \[1\]. The more general problem of summing an integer stream seems to be tractable using the same techniques. Our research focuses on efficient solutions for generalizations of the counting and summing problems.

In real life, recent data is often more important than old one; accordingly, there have been an extensive study of algorithms that are able to forget. Our research continues this line of works, and greatly focuses on the sliding window model, where only the last $W$ data elements are of interest. While applications like counting and summing are straightforward if the entire stream is taken into consideration, tackling these problems in the sliding window setting is much more challenging.

Another extensively studied problem is estimating element frequencies, i.e., how many times an item has appeared in the stream. This can be seen as a generalization of the counting problem, where we allow queries not only for the number of 1’s, but for an arbitrary stream element. While it seems immediate to allocate a counter for each element,
in many cases memory constraints (such as using the small SRAM) prohibits such an approach. Interestingly, if we allow a one-sided estimation error, which is bounded by a fixed percentage $\epsilon$ of the stream length, the problem can be solved using $\Theta(\epsilon^{-1})$ counters. Further, this is related to another fundamental problem — finding the heavy hitters of the stream, where we need to identify all elements which appeared at least a $\theta$ fraction of the stream. An exact solution for the heavy hitters problem requires allocating a counter for each element. To overcome this difficulty, we allow a slack of $\epsilon$; that is, we are required to report any item that appeared more than a $\theta$ fraction, leave out items appearing less than a $\theta - \epsilon$ fraction, and may act arbitrarily otherwise. Given an algorithm that solves the $\epsilon$-error frequency estimation problem, we can identify the heavy hitters simply by returning all items for which the estimated frequency is at least $\theta - \epsilon$.

Another well studied generalization of the simple counting problem is BASIC-COUNTING, in which we wish to count the number of 1’s appearing in the last $W$ bits of a binary stream. The positive aspect of a fixed-size domain for the stream elements is that we can easily associate a counter with each item. Specifically, on a binary stream it is enough to focus on approximating the number of 1’s in a window. This allows non-trivial algorithms that approximate BASIC-COUNTING within a multiplicative error of $1 + \epsilon$ using $\Theta(\epsilon^{-1} \cdot \text{polylog } W)$ bits. Such approximation is infeasible for general frequency estimation without allocating a counter for each element. Our work studied the potential benefits of trading the multiplicative error guarantees for a weaker additive $W\epsilon$-error, showing it can reduce the memory requirement to $\Theta(\epsilon^{-1} + \log W)$ bits. This then generalizes to the useful problem of BASIC-SUMMING, where each stream element is in the range $\{0, 1, \ldots, R\}$, and we wish to approximate the sum of the last $W$ elements. Once again, our results prove that by weakening the estimation guarantee, we can provide efficient additive approximation using significantly less memory.

The frequency estimation and BASIC-COUNTING problems can then be combined into approximating item frequencies over sliding windows. Here, we provide the first memory-optimal, constant time algorithm that approximates up to an additive $W\epsilon$ error how many times an element has appeared within the last $W$ elements of the stream.

Additional important variation is the weighted frequency estimation, where each arriving element is associated with a weight, and the goal is to approximate the total weight of an item to within an additive error. Finally, we are currently generalizing our sliding window framework to manage windows whose size may change over time. The complete diagram of problems and their generalizations is given in Figure 1.2.

As discussed above, we emphasize the importance of memory efficient streaming algorithms for many applications in network monitoring, databases and other fields. This is becoming crucial as the sheer amounts of data generated in our world continues to grow exponentially.
Figure 1.2: Relations between the discussed problems. Each line goes from a base problem to a generalized one. The gray boxes represent related works while our projects focus on the green ones.

1.2 Related Work

In this section, we review previous works related to our research.

1.2.1 Sliding Windows

In [2], Datar et al. first presented the problem of counting the number of 1’s in a sliding window of size $W$ over a binary stream, known as BASIC-COUNTING. They have also studied its generalization to BASIC-SUMMING, in which one approximates the sum of the last $W$ integers over a stream of integers in the range $\{0, 1, \ldots, R\}$. They have introduced a data structure called exponential histogram ($EH$). $EH$ is a time-stamp based structure that partitions the stream into buckets, saving the time elapsed since the last 1 in the bucket was seen. Using $EH$, they have derived a space-optimal algorithm for approximating BASIC-SUMMING within a multiplicative-factor of $(1 + \epsilon)$, which uses $O\left(\frac{1}{\epsilon} \cdot \left(\log^2 W + \log R \cdot (\log W + \log \log R)\right)\right)$ memory bits. The structure allows estimating a class of aggregate functions such as counting, summing and computing the $\ell_1$ and $\ell_2$ norms of a sliding window in a stream containing integers. The exponential histogram technique was later expanded [3] to support computation of additional functions such as $k$-median and variance. Gibbons and Tirthapura [4] presented a different structure called waves, which improved the worst-case runtime of processing a new element to a constant, keeping space requirement comparable when $R = poly(W)$. Braverman and Ostrowsky [5] defined smooth histogram, a generalization of the exponential histogram, which allowed estimation of a wider class of aggregate functions and improved previous results for several functions such as $l_p$ norms and frequency moments. Lee and Ting [6] presented an improved algorithm, requiring less space if a $(1 + \epsilon)$ approximation is
guaranteed only when the ones consist of a significant fraction of the window. They also presented the \( \lambda \) counter [6] that counts bits over a sliding window as part of a frequent items algorithm.

In [7], Cohen and Strauss considered a generalization of the bit-counting problem on a sliding window for computing a weighted sum for some decay function, such that the more recent bits have higher weights. Cormode and Yi [8] solved bit counting in a distributed setting with optimal communication between nodes.

Extensive studies were conducted on many other streaming problems over sliding windows such as Top-K [9,10], Top-K tuples [11], Quantiles [12], heavy hitters [13,14], distinct items [15], duplicates [16], Longest Increasing Subsequences [5,17], Bloom filters [18,19], graph problems [20,21] and more.

1.2.2 Frequency Estimation and Heavy Hitters

The frequency estimation problem, in which one tries to approximate the number of times a specific element have appeared in the stream, is one of the most fundamental problems of network monitoring and received a lot of attention from the research community.

Algorithms for frequency estimation may focus on reducing the number of bits needed per counter while storing a dedicated counter for each flow. These algorithms include hybrid SRAM/DRAM techniques which store the least significant bits at the SRAM, thereby minimizing the number of access to DRAM for updates. Other algorithms use approximate counting techniques which probabilistically increment the counter values, thus slowing their growth and the number of bits they require [1,22,23,24,25,26]. Unfortunately, sometimes the fast memory that supports line-speed updates is so scarce, there is not enough space to store even a single bit per counter. Additionally, even if there is enough space to store all counters, identifying the heavy hitters, or the \( k \) most frequent elements might be challenging when the number of distinct items is large.

This motivated the research of shared counters algorithms [27,28,29,30]. In these, the same counter may be used for counting several flows, and each flow may be associated with multiple counters. Traditionally, shared counters algorithms are divided into two subgroups, sketch based and counter based.

**Sketch Based Techniques**

For many problems, Counting Sketches such as Multi Stage Filters [31] and Count Min Sketch [28] are a popular solution. These algorithms offer the following guarantee: for a stream of size \( N \), their frequency estimation is correct up to an additive \( N\epsilon \)-error, only with probability of at least \( 1 - \delta \).

These methods are typically optimized for hardware implementation as they use statically allocated memory and do not store explicit flow identifiers. Therefore, these methods can mainly answer point queries. That is, given a flow identifier, the sketching
method generates an estimation for that flow. Despite these limitations, many networking applications found this approach useful \cite{28, 30, 31, 32, 33, 34, 35}.

**Counter Based Techniques**

Counter based techniques are typically not tailored for hardware and therefore do not limit themselves to point queries or statically allocated memory. These methods usually use hash tables to store the explicit keys of frequent elements and can also answer more complex queries such as “what are the frequent elements?” on demand.

These methods are typically analyzed only with respect to the number of items they store, disregarding other implementation overheads. Famous examples include *Space Saving* \cite{30}, *Lossy counting (LC)* \cite{35} and *Frequent* \cite{29, 36}.

**Space Saving**

In this subsection, we review in detail Space Saving (SS) \cite{30} — a counter based algorithm for frequency estimation and heavy hitters which is considered to be the state of the art \cite{37}. Being the best solution for the problem, Space Saving was a basic component in many of our works.

**Algorithm 1 Space-Saving**

Initialization: $C = \emptyset$

1. function \texttt{ADD}(Item $x$)
2. 	if $x \in C$ then
3. 		Increment $c_x$
4. 	else
5. 		if $|C| < M$ then
6. 			$c_x = 1$
7. 		else
8. 			$m = \text{argmin}_{y \in C} c_y$
9. 			$C = C \setminus \{m\} \cup \{x\}$
10. 			$c_x \leftarrow c_m$
11. 		end if
12. 	end if
13. end function

14. function \texttt{QUERY}(Item $x$)
15. 	if $x \in C$ then return $c_x$
16. 	else return $\min_{y \in C} c_y$
17. end if
18. end function

Schematically, Space Saving allocates a fixed number of counters; whenever an item arrives, Space Saving increments its counter. If the arriving item does not have an allocated counter, this item inherits the minimal value counter while the previously associated item is “forgotten”. Algorithm 1 provides a high level pseudo code of Space Saving without implementation details. The code utilizes $M$ counters and denotes the current counter-set by $C$. 
Space Saving operates in $O(1)$ hash-table operations per update and requires the optimal number of $O(1/\varepsilon)$ counters. Denoting by $N$ the size of the stream, the frequency of $x$ in the stream by $f_x$, and the algorithm’s approximation by $\hat{f}_x$, it satisfies the following:

**Theorem 1.2.1.** Using $\frac{1}{\varepsilon}$ counters, Space Saving answers Query($x$) in $O(1)$ time (w.h.p.) such that $f_x \leq \hat{f}_x \leq f_x + N\varepsilon$.

### 1.2.3 Hierarchical Heavy Hitters

The attacking devices of a DDoS attack are often located within a small number of subnetworks and thus share some of their IP prefixes with each other. *Hierarchical Heavy Hitters (HHH)* [38][39][40] is a measurement technique that is designed to identify such attacks. Specifically, the technique is capable of observing that specific subnets are responsible for a large portion of the traffic even if no individual flow within these subnets is a heavy hitter by itself.

Hence, HHH identification is a powerful tool that can be used to mitigate DDoS attacks [41][42]. Intuitively, one can monitor the system and record which subnets usually deliver a significant portion of the traffic. Then, we can identify new and unexpected hierarchical heavy hitters as potential sources of a DDoS attack. Such an identification can then be used to block the attack traffic with minimal impact on legitimate traffic. In Chapter 6 we provide the first HHH solution fast enough to cope with modern line rates of virtual switches.

HHH are usually calculated with regard to packet counts. However, the rise of technologies such as jumbo frames [44][45] and of networking devices that support large packet payloads exposes a new attack vector. A rogue subnet can deliver a small number of large packets to avoid being detected as a hierarchical heavy hitter. Yet, the combined volume of its traffic can exhaust the bandwidth of the receiving network. In chapter 5 we show how our approach can accelerate existing techniques for identifying volume-heavy hierarchical heavy hitters.

### 1.3 Thesis Outline

In Chapter 3 we show algorithms and lower bound for the problem of maintaining sum (or average) over sliding windows. Our solution is succinct (optimal up to a $1 + o(1)$ factor) and requires significantly less space than previous approaches. Next, Chapter 4 discusses how to employ randomness to effectively filter out tail elements. Focusing on the frequency estimation and top-$k$ identification problems, we illustrate how a randomized admission filter can reduce the space required for a given error by a $x2$-$x32$ factor. Chapter 5 then discusses a generalization of the frequency estimation problem to weighted streams in which we find the flows that are heavy in terms of bandwidth / byte-volume. We also combine this with the approach from Section 3 to find heavy flows over sliding windows,
and discuss another extension to hierarchical heavy hitters. In Chapter 6 that follows, we study the hierarchical heavy hitters problem in depth and provide a randomized algorithm with constant worst case update time. This asymptotically improves over previous works and allows deployment on real virtual switches. Finally, in Chapter 7 we then consider additional measurement types over sliding windows, including Bloom filters, Count Distinct queries and Entropy.
Chapter 2

Preliminaries

In this section, we discuss the key terms that frequently appear throughout the thesis.

Flow In networking, a flow is the identifier of a connection and is usually defined using a 5-tuple that includes the source IP address, destination address, source and destination ports, and protocol. This identifier is considered unique (per connection) and we often wish to provide per-flow analysis of the traffic.

Stream A stream is a quasi-infinite ordered sequence of elements that arrive in an online fashion. That is, we always assume that new elements can be added to the stream and we are required to process them sequentially.

Streaming Algorithms Algorithms that process streams and answer queries upon request are called streaming algorithms. They are required to be space-efficient (i.e., storing the entire stream is considered infeasible) and process elements quickly to keep up with the rate in which elements arrive. Queries refer to the current state of the stream and these algorithms are not required to make forecasts for future data.

Sliding Windows As recent data is often more useful than old one, the sliding window model focuses on the last \( W \) elements alone. That is, a sliding window stream algorithm processes a stream of elements but when queried, its answer reflects the characteristics of the last \( W \) elements sequence alone.

Heavy Hitters Also known (in a networking context) as elephant flows, heavy hitters are the most frequent elements of a stream. The identification of the heavy hitters, under various models, is at the core of many networking and database applications.
Chapter 3

Summing over Sliding Windows

3.1 Introduction

This chapter considers the problem of estimating the sum the last \( W \) elements of a stream of integers in \( \{0, 1, \ldots, R\} \). Specifically, we study the memory requirements for computing a \( RW\epsilon \)-additive approximation for the window’s sum. We derive a lower bound of \( W \log \left\lfloor \frac{1}{2W\epsilon} + 1 \right\rfloor \) bits when \( \epsilon \leq 1/2W \) and show a matching succinct algorithm that uses \( (1 + o(1))(W \log \left\lfloor \frac{1}{2W\epsilon} + 1 \right\rfloor) \) bits. Next, we prove a \( (1 - o(1))\epsilon^{-1}/2 \) bits lower bound when \( \epsilon = \omega(W^{-1}) \) and \( \epsilon = o(\log^{-1} W) \) and provide a succinct algorithm that requires \( (1 + o(1))\epsilon^{-1}/2 \) bits. We show that when \( \epsilon = \Omega(\log^{-1} W) \) any solution to the problem must consume at least \( (1 - o(1)) \cdot (\epsilon^{-1}/2 + \log W) \) bits, while our algorithm needs \( (1 + o(1)) \cdot (\epsilon^{-1}/2 + 2\log W) \) bits. Finally, we show that our lower bounds generalize to randomized algorithms as well, while our algorithms are deterministic and can process elements and answer queries in \( O(1) \) worst-case time.

Background

The ability to process and maintain statistics about large streams of data is useful in many domains, such as security, networking, sensor networks, economics and business intelligence. Since the data may change considerably over time, there is often a need to maintain the statistics with respect to some window of the last \( W \) elements at any given point. A naive solution to this problem is to keep the \( W \) most recent elements, add an element to the statistic when it arrives, and subtract it when it leaves the window. Yet, when the window of interest is large, which is often the case when data arrive at high rate, the required memory overhead might become a performance bottleneck.

Though it may be tempting to think that RAM memory is cheap, a closer look indicates that there are still performance benefits in maintaining small data structures. For example, hardware devices such as network switches prefer to store important data in the faster and scarcely available SRAM than in DRAM. This is in order to keep up with the ever increasing line-speed of modern networks. Similarly, on a CPU, caches provide much faster
performance than DRAM memory. Thus, small data structures that fit inside a single cache line and can possibly be pinned there are likely to result in much faster performance than a solution that spans multiple lines that are less likely to be constantly maintained in the cache.

A well known method to conserve space is to approximate the statistics. Basic-Counting is a basic textbook example of such approximated stream processing problems [2]. In this problem, one is required to keep track of the number of 1’s in a stream of binary bits. The work of [2] includes a \((1 + \epsilon)\)-multiplicative approximation algorithm that requires \(O\left(\frac{1}{\epsilon} \log^2 W \epsilon\right)\) bits. The solution works with amortized \(O(1)\) time but its worst case time complexity is \(O(\log W)\).

A more practical problem is Basic-Summing, in which the goal is to maintain the sum of the last \(W\) non-negative integers in the range \([R] = \{0, 1, \ldots, R\}\). The work in [2] provides a \((1 + \epsilon)\)-multiplicative approximation Basic-Summing using \(O\left(\frac{1}{\epsilon} \cdot (\log^2 W + \log R \cdot (\log W + \log \log R))\right)\) bits. The amortized time complexity is \(O\left(\frac{\log R}{\log W}\right)\) and the worst case is \(O(\log W + \log R)\).

**Our Contributions**

In this chapter, we explore the benefits of changing the approximation guarantee from multiplicative to additive. With a multiplicative approximation, the estimation deviates from the true value by a multiplicative factor, e.g., 5%. In an additive approximation, the absolute error is bounded, e.g., a deviation of up to ±5. Multiplicative approximation is more appealing when the expected sum is small compared to \(R\); specifically, if the average item is \(o\left(\frac{R}{\log 2 W}\right)\) then the multiplicative algorithms produce better estimates for the same amount of memory. However, in many cases the average item is \(\Theta\left(\frac{R}{\log 2 W}\right)\). For example, if we measure the actual bandwidth of a 100GBps link in a 24 hours window, we can get far better results by using the additive algorithm unless the link is extremely under-utilized and only forwards a few megabytes per second. Furthermore, the potential space saving is significant in this case, motivating our exploration.

We explore the Basic-Summing problem in which we are required to process a stream of integers in \(\{0, 1, \ldots, R\}\) and upon query provide a \(RW\epsilon\)-additive approximation for the sum of the last \(W\) elements. Here, the results are split based on the value of \(\epsilon\). Specifically, our contribution is an (asymptotically) space optimal algorithm providing an \(RW\epsilon\)-additive approximation for the Basic-Summing problem when \(\epsilon^{-1} \leq W \left(2 - \frac{1}{\log 2 W}\right)\). It uses \(O(\epsilon^{-1} + \log W)\) memory bits and has \(O(1)\) worst case time complexity. For other values of \(\epsilon\), we show a lower bound of \(\Omega(W \log \left(\frac{1}{W\epsilon}\right))\) and a corresponding algorithm requiring \(O\left(W \log \left(\frac{1}{2W\epsilon} + 1\right)\right)\) memory bits with \(O(1)\) worst case time complexity. Furthermore, we show that this algorithm is succinct for \(\epsilon = o(W^{-1})\), i.e., its space requirement is only \((1+o(1))\) times the lower bound.

\[1\text{In this chapter, the logarithms are of base 2 and the } o(1) \text{ notation is for } W \to \infty.\]
Table 3.1: Comparison of Basic-Summing Algorithms.

<table>
<thead>
<tr>
<th>Case</th>
<th>Upper bound</th>
<th>Lower bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon &lt; 1/2W$</td>
<td>$(1 + o(1))W \log \left\lfloor \frac{W}{2W_\epsilon} + 1 \right\rfloor$</td>
<td>$W \log \left\lfloor \frac{W}{2W_\epsilon} + 1 \right\rfloor$</td>
</tr>
<tr>
<td>$(\epsilon &gt; 1/2W) \land (\epsilon = \Theta(W^{-1})) \land (2W_\epsilon \notin N)$</td>
<td>$(1 + o(1))\epsilon^{-1}/2$</td>
<td>$W_\epsilon &gt; \epsilon^{-1}/4$</td>
</tr>
<tr>
<td>$(\epsilon &gt; 1/2W) \land (\epsilon = \Theta(W^{-1})) \land (2W_\epsilon \in N)$</td>
<td>$(1 + o(1))\epsilon^{-1}/2$</td>
<td>$W_\epsilon &gt; \epsilon^{-1}/2$</td>
</tr>
<tr>
<td>$(\epsilon = o(\log^{-1} W)) \land (\epsilon = \omega(W^{-1}))$</td>
<td>$(1 + o(1))(\epsilon^{-1}/2 + 2 \log W)$</td>
<td>$(1 - o(1))(\epsilon^{-1}/2 + \log W)$</td>
</tr>
<tr>
<td>$\epsilon = \Omega(\log^{-1} W)$</td>
<td>$(1 + o(1))(\epsilon^{-1}/2 + 2 \log W)$</td>
<td>$W_\epsilon &gt; \epsilon^{-1}/2$</td>
</tr>
</tbody>
</table>

Table 3.2: Summary of our results for a $RW_\epsilon$-approximation of the Basic-Summing problem.

To get a feeling for the applicability of these results, consider for example an algorithmic trader that makes transactions based on a moving average of the gold price. He samples the spot price once every millisecond and wishes to approximate the average price for the last hour, i.e., $W = 3.6 \times 10^6$ samples. The current price is around $1200, and with a standard deviation of $10, he safely assumes the price is bounded by $R = 1500. The trader is willing to withstand an error of $0.1\%$, which is approximately $1.2. Our algorithm provides a $W R_\epsilon A$ (the ‘A’ stands for Additive) additive-approximation using $\left(\frac{1}{2A} + 2 \log W\right)(1 + o(1))$ memory bits, while the algorithm by Datar et al. [2] computes a multiplicative $(1 + \epsilon_M)$ (the ‘M’ stands for Multiplicative) approximation using $\left[\frac{1}{2M} + 1\right]\left\lceil \log (2WR_\epsilon A + 1) + 1 \right\rceil$ buckets of size $\lceil \log W + \log (\log W + \log R) \rceil$ bits

---

2Except for the case of $(\epsilon > 1/2W \land \epsilon = \Theta(W^{-1}) \land 2W_\epsilon \notin N)$, in which we get a $\Theta(W)$ bits bound but are not succinct.
each. Using our algorithm, the trader sets $\epsilon_A = R^{-1} = \frac{1}{1500}$, which guarantees that as long as the price of gold stays above $1000, the error remains lower than required. The overall memory requirement of our algorithm with the parameters above is about 100 bytes. The multiplicative approximation algorithm requires setting $\epsilon_M = 0.1\%$, and uses $501 \cdot \lceil \log (1080001) + 1 \rceil = 12525$ buckets of size 27 bits each and about 41 KB overall!

In summary, we show that additive approximations offer significant space reduction opportunities. Additionally, we obtained a constant worst case time complexity which is important in real-time and time sensitive applications.

### 3.2 Related Work

In [2], Datar et al. first presented the problem of counting the number of 1’s in a sliding window of size $W$ over a binary stream, and its generalization to summing a window over a stream of integers in the range $\{0, 1, \ldots, R\}$; this generalization is known as the BASIC-SUMMING problem. They have introduced a data structure called Exponential Histogram ($EH$). $EH$ is a time-stamp based structure that partitions the stream into buckets, saving the time elapsed since the last 1 in the bucket was seen. Using $EH$, they derive a space-optimal algorithm for approximating BASIC-SUMMING within a multiplicative-factor of $(1 + \epsilon)$, which uses $O \left( \epsilon^{-1} \left( \log^2 W + \log R \cdot (\log W + \log \log R) \right) \right)$ memory bits. $EH$ allows estimating a class of aggregate functions such as counting, summing and computing the $\ell_1$ and $\ell_2$ norms of a sliding window in a stream of integers. The exponential histogram technique was later expanded [3] to compute additional functions such as $k$-median and variance. Gibbons and Tirthapura [4] presented a different structure called Waves that has constant worst case update time. Waves requires comparable space to $EH$ as long as $R = \text{poly}(W)$. Braverman and Ostrowsky [5] defined Smooth Histogram, a generalization of $EH$ that can estimate a wider class of aggregate functions, and improved previous results for several problems such as $l_p$ norms and frequency moments. For the special case of binary streams, Lee and Ting [6] presented an improved algorithm requiring less space when the number of ones is a significant fraction of the window. They also presented the $\lambda$ counter [6] that counts bits over a sliding window as part of a frequent items algorithm. Their results are not comparable to ours as they support only counting but not summing.

In [7], Cohen and Strauss considered a generalization of the bit-counting problem on a sliding window for computing a weighted sum for some decay function, such that the more recent bits have higher weights. Cormode and Yi [8] solved bit counting in a distributed setting with optimal communication between nodes. Extensive studies were conducted on many other streaming problems over sliding windows such as top-k [9, 10], top-k tuples [11], quantiles [12], heavy hitters [14, 13], distinct items [15], duplicates [16], longest increasing subsequences [17, 5], Bloom filters [18, 19], graph problems [20, 21] and more.
3.3 Definitions

For any \( k \in \mathbb{N} \), we denote \([k] \triangleq \{0, 1, \ldots, k\}\).

**Approximation** Given a value \( V \) and a constant \( \epsilon \), we say that \( \hat{V} \) is an \( \epsilon \)-multiplicative approximation of \( V \) if \( |V - \hat{V}| < \epsilon V \). We say that \( \hat{V} \) is an \( \epsilon \)-additive approximation of \( V \) if \( |V - \hat{V}| < \epsilon \).

We now formally define the Basic-Summing problem.

**Definition (Basic-Summing)** Given a stream of elements comprising of integers in the range \([R] = \{0, 1, \ldots, R\}\), maintain the sum \( S \) of the last \( W \) elements.

In this chapter, we consider the problem of computing a \( RW\epsilon \)-additive approximation of Basic-Summing. Notice that if \( \epsilon > 1/2 \), one may always return \( RW\epsilon/2 \) and provide the required approximation. Thus, we hereafter assume that \( \epsilon \leq 1/2 \).

3.4 Lower Bounds

We now show lower bounds for the memory required to provide a \( RW\epsilon \)-additive approximation of Basic-Summing.

**Lemma 3.4.1.** For any \( W \) and \( \epsilon \), any deterministic algorithm that provides a \( RW\epsilon \)-additive approximation for Basic-Summing requires at least \( \left\lceil \frac{W}{2W\epsilon} \right\rceil \) bits.

**Proof.** Denote \( z \triangleq \left\lceil \frac{W}{2W\epsilon} \right\rceil \). Consider the language \( L_1 \) in which every word is a sequence of \( W \) zeros followed by \( z \) blocks of size \( [2W\epsilon] \), such that each block consists of only zeros or entirely of the symbol \( R \). Formally, we define the language as follows:

\[
L_1 \triangleq \left\{ 0^W \cdot w_0 \cdot w_1 \cdots w_{z-1} \mid \forall j \in [z-1] : w_j = 0^{[2W\epsilon]} \vee w_j = R^{[2W\epsilon]} \right\}.
\]

Notice that we have \( \log |L_1| = z \). We now show that every two distinct words in \( L_1 \) must reach different memory configurations, thereby implying a \( \lceil \log |L_1| \rceil \) bits lower bound and concluding the proof. Assume by contradiction, that two different words \( s_1, s_2 \in L_1 \) such that \( s_1 = 0^W \cdot w_{0,1} \cdots w_{z-1,1}, s_2 = 0^W \cdot w_{0,2} \cdots w_{z-1,2} \)
lead the algorithm to the same configuration. In that case, there exists some \( i \in [z-1] \) such that \( w_{i,1} \neq w_{i,2} \). Denote the index of the last block that differs between \( s_1 \) and \( s_2 \) by \( t \triangleq \max \{ \tau \mid w_{\tau,1} \neq w_{\tau,2} \} \). Next, consider the sequences \( \tilde{s}_1 = s_1 \cdot 0^{(t-1)[2W\epsilon]} \) and \( \tilde{s}_2 = s_2 \cdot 0^{(t-1)[2W\epsilon]} \). The algorithm must reach the same configuration after processing these sequences, even though the sums of the last \( W \) elements in \( \tilde{s}_1 \) and \( \tilde{s}_2 \) differ by \([2RW\epsilon]\). Therefore, the algorithm’s error must be at least \( RW\epsilon \) at least for one of the sequences, in contradiction to the assumption. We conclude that in all cases, we have shown that distinct words in \( L_1 \) must lead to distinct configuration thus proving a \( \lceil \log |[L_1]| \rceil \) bits lower bound.

\[\square\]
We note that the asymptotic behavior of our bound for large $\epsilon$ values satisfies the following.

**Corollary 3.4.2.** For any $W$ and $\epsilon$ such that $\epsilon = \omega(W^{-1})$, any deterministic algorithm that provides a $RW\epsilon$-additive approximation for **Basic-Summing** requires at least $(1 - o(1))\epsilon^{-1}/2$ bits.\(^3\)

Note that if $2W\epsilon \in \mathbb{N}$, then $\left\lfloor \frac{W}{2W\epsilon} \right\rfloor = \left\lfloor \epsilon^{-1}/2 \right\rfloor$ and the same bound also holds for $\epsilon = \Theta(W^{-1})$; that is, we have:

**Lemma 3.4.3.** For any $W$ and $\epsilon$ such that $2W\epsilon \in \mathbb{N}$, any deterministic algorithm that provides a $RW\epsilon$-additive approximation for **Basic-Summing** requires at least $(1 - o(1))\epsilon^{-1}/2$ bits.\(^3\)

We now proceed with an improved bound for larger values of $\epsilon$.

**Lemma 3.4.4.** For any $W$ and $\epsilon$ such that $\epsilon \geq \log \frac{W}{2W}$, the minimum number of memory bits used by any deterministic algorithm that provides a $RW\epsilon$-additive approximation for **Basic-Summing** is at least

\[
\log \left( 2 \left\lceil \frac{W}{2W(1 + \log^{-1} W)} \right\rceil^{-1} - 1 \right) \cdot \left\lfloor \frac{2W\epsilon}{\log W + 1} \right\rfloor.
\]

**Proof.** Denote $z \triangleq \left\lfloor \frac{W}{2W(1 + \log^{-1} W)} \right\rfloor - 1$. Consider the language $L_2$, in which every word is a concatenation of:

1. $W$ zeros.
2. $z$ blocks of size $\left\lceil 2W\epsilon (1 + \log^{-1} W) \right\rceil$, such that each block consists of only zeros or entirely of the symbol $R$. Further, at least one of the sequences must consists of $R$’s and not zeros.
3. $\ell$ zeros for some $\ell \in \left\lfloor \frac{2W\epsilon}{\log W} \right\rfloor$.

Formally, we define the language as follows:

\[
L_2 \triangleq \left\{ 0^W \cdot w_0 \cdot w_1 \cdots w_{z-1} \cdot 0^\ell \mid \ell \in \left\lfloor \frac{2W\epsilon}{\log W} \right\rfloor \right\}.
\]

Notice that we have $\log |L_2| = \log \left((2^z - 1) \cdot \left\lfloor \frac{2W\epsilon}{\log W + 1} \right\rfloor\right)$. We now show that every two distinct words in $L_2$ must reach different memory configurations, thereby implying a $\lceil \log |L_2| \rceil$ bits lower bound and concluding the proof. Assume by contradiction, that two different words $s_1, s_2 \in L_2$ such that

\[
s_1 = 0^W \cdot w_{0,1} \cdot w_{1,1} \cdots w_{z-1,1} \cdot 0^{\ell_1},
\]

\[
s_2 = 0^W \cdot w_{0,2} \cdot w_{1,2} \cdots w_{z-1,2} \cdot 0^{\ell_2}
\]

lead the algorithm to the same configuration. First, assume that there exists some $i \in [z-1]$ such that $w_{i,1} \neq w_{i,2}$. Denote the index of the last block that differs between $s_1$ and $s_2$\(^3\) This does not follow immediately for $\epsilon = \Theta(1)$. In this case, the correctness follows from Lemma 3.4.4.
by $t \triangleq \max \{\tau \mid w_{\tau,1} \neq w_{\tau,2}\}$. Next, consider the sequences $s_1 = s_1 \cdot 0^{(t-1)}[2W\epsilon(1+\log^{-1}W)]$ and $s_2 = s_2 \cdot 0^{(t-1)}[2W\epsilon(1+\log^{-1}W)]$. The algorithm must reach the same configuration after processing these sequences, even though the sums of the last $W$ elements in $s_1$ and $s_2$ differ by $[2RW\epsilon]$, regardless of the values of $\ell_1, \ell_2$. Therefore, its error must be at least $RW\epsilon$ at least for one of the sequences, in contradiction to the assumption. Next, we cover the case where $$w_{0,1} \cdot w_{1,1} \cdots w_{z-1,1} = w_{0,2} \cdot w_{1,2} \cdots w_{z-1,2},$$
and $s_1$ differs from $s_2$ only in the trailing number of zeros. We assume without loss of generality that $\ell_1 < \ell_2$, i.e., that $s_2 = s_1 \cdot 0^{2-\ell_1}$. This means that the algorithm reached some memory configuration after reading $s_1$, and then returned to it after seeing another $\ell_2 - \ell_1$ zeros. The algorithm is deterministic and must return to that configuration once again after seeing an additional $\ell_2 - \ell_1$ zeros. Further, we can add any multiple of $\ell_2 - \ell_1$ zeros to the input and reach that configuration once again. Thus, after processing $s_1 \cdot 0^{W(t-\ell_1)}$ we have a window which is all zeros, and the algorithm reaches the same configuration as when seeing $s_1$. Finally, since $\exists k \in [z-1]$ such that $w_{k,1} = R[2W\epsilon]$, we get that these two windows’ sum differs by at least $2RW\epsilon$ and thus the algorithm cannot be correct on both. We conclude that in all cases, distinct words in $L_2$ lead to distinct configuration thus proving a $[\log ||L_2||]$ bits lower bound.

Next, we show the merit of Lemma 3.4.4 by proving that its bound improves over Corollary 3.4.2 for large $\epsilon$.

**Lemma 3.4.5.** For any $W$ and $\epsilon$ such that $\epsilon = \omega \left(\frac{\log W}{W}\right)$, any deterministic algorithm that provides a $RW\epsilon$-additive approximation for BASIC-SUMMING requires at least $\left(1 - o(1)\right) \cdot (\epsilon^{-1}/2 + \log W)$ bits.

**Proof.** Lemma 3.4.4 shows a lower bound of $$\log \left(2\left[\frac{W}{2W\epsilon(1+\log^{-1}W)}\right]^{-1} - 1\right) \cdot \left[2W\epsilon/\log W + 1\right]$$ bits. Thus, in the $\epsilon = \omega(W^{-1})$ case we can express it as follows:

$$\log \left(2\left[\frac{W}{2W\epsilon(1+\log^{-1}W)}\right]^{-1} - 1\right) \cdot \left[2W\epsilon/\log W + 1\right] \geq \log \left(2\left[\frac{W}{2W\epsilon(1+o(1))}\right]^2 - 1\right) + \log (2W\epsilon/\log W)$$

$$\geq \frac{1}{2\epsilon(1+o(1))} + \log W - \log \epsilon^{-1} - \log \log W - O(1) = (1 - o(1)) \cdot (\epsilon^{-1}/2 + \log W) . \quad \Box$$

Next, we show a lower bound for smaller values of $\epsilon$.

**Lemma 3.4.6.** For any $\epsilon$, any deterministic algorithm that provides a $RW\epsilon$-additive approximation for BASIC-SUMMING requires at least $W \log \left[\frac{1}{2W\epsilon} + 1\right]$ memory bits.

**Proof.** First, since the sum of the last $W$ elements is an integer, we can assume that $2RW\epsilon \in \mathbb{N}$ as otherwise, by rounding the estimate to the nearest half-integer, we
also provide a \(\lfloor \frac{2 RW \epsilon}{2} \rfloor\)-additive approximation. Denote \(x \triangleq 2 RW \epsilon\) and \(C \triangleq \{n \cdot x \mid n \in \{0, 1, \ldots, \left\lfloor \frac{1}{2 W \epsilon} \right\rfloor \}\); notice that \(\forall n \in C: 0 \leq n \leq R\). Let \(L_3\) be the language of all \(W\) length strings over \(C\), i.e.,

\[ L_3 = \{\sigma_0 \sigma_1 \cdots \sigma_{W-1} \mid \forall j \in [W-1]: \sigma_j \in C\} \]

We show that every two distinct sequences in \(L\) must lead the algorithm to distinct configurations implying a lower bound of

\[ \lceil \log |L| \rceil \geq W \log |C| = W \log \left\lfloor \frac{1}{2 W \epsilon} + 1 \right\rfloor \]

bits. Assume, by way of contradiction, that two different words \(s_1, s_2 \in L\) lead the algorithm to the same configuration and denote:

\[ s_1 = \sigma_0^1 \sigma_1^1 \cdots \sigma_{W-1}^1, s_2 = \sigma_0^2 \sigma_1^2 \cdots \sigma_{W-1}^2 \in L. \]

Denote the index of the last letter that differs between \(s_1\) and \(s_2\) by \(t \triangleq \max\{\tau \mid \sigma_\tau^1 \neq \sigma_\tau^2\}\). Next, consider the sequences \(s_1 \cdot 0^t\) and \(s_2 \cdot 0^t\). The algorithm must reach the same configuration after processing these sequences, even though the sum of the last \(W\) elements differ by at least \(x\). Therefore, the algorithm’s error must be at least \(RW \epsilon\) at least for one of the sequences, in contradiction to the assumption.

An immediate corollary of Theorem 3.4.6 is that any exact algorithm for Basic-Summing requires at least \(W \lfloor \log (R + 1) \rfloor\) bits, i.e., the naive solution of maintaining a \(W\)-sized array of the elements in the window, encoding each using \(\lceil \log (R + 1) \rceil\) bits, is essentially optimal (for exact Basic-Summing). We now summarize the above lemmas in a theorem.

**Theorem 3.4.7.** Approximating Basic-Summing to within an additive error of \(RW \epsilon\) requires

1. \(W \log \left\lfloor \frac{1}{2 W \epsilon} \right\rfloor\) bits for \(\epsilon \leq 1/2W\).
2. \(\Omega (W)\) bits if \(\epsilon = \Theta(W^{-1}) \land 2W \epsilon \notin \mathbb{N}\).
3. \((1 - o(1))\epsilon^{-1}/2\) bits when \(\epsilon > 1/2W \land \epsilon = o(\log^{-1} W) \land (\epsilon = \omega(W^{-1}) \lor 2W \epsilon \in \mathbb{N})\).
4. \(\Omega (\log W)\) bits for \(\epsilon = \Omega (\log^{-1} W)\).

Finally, we extend our lower bounds to randomized algorithms.

**Theorem 3.4.8.** Let \(B\) be the lower bound showed in Theorem 3.4.6 on the space requirement for any deterministic algorithm for Basic-Summing. Any randomized Las Vegas algorithm that provides a \(RW \epsilon\)-additive approximation for the Basic-Summing problem requires at least \(B\) bits. Further, for any fixed \(\delta \in (0, 1/2)\), any randomized Monte Carlo algorithm that with probability at least \(1 - \delta\) approximates Basic-Summing within \(RW \epsilon\) error at any time instant, requires \((1 - o(1))B/2\) bits.
Proof. We say that algorithm $A$ is $\epsilon$-correct on an input instance $S$ if it is able to approximate the sum of the last $W$ elements, at every time instant while reading $S$, to within an additive error of $RW\epsilon$.

We remind the reader that in our case, a Las Vegas (LV) algorithm for the BASIC-SUMMING approximation problem is a randomized algorithm which is always $\epsilon$-correct. In contrast, a Monte Carlo (MC) algorithm is a randomized procedure that is allowed to provide approximation with error larger than $RW\epsilon$ with probability at most $\delta$.

The Yao Minimax principle \cite{46} implies that the amount of memory required for a deterministic algorithm to approximate a random input chosen according to a distribution $p$ is a lower bound on the expected space consumption of a Las Vegas algorithm for the worst input. To prove a $\left\lfloor \frac{W}{2W\epsilon} \right\rfloor$ bits lower bound, we consider padding the language $L_1$, which is defined in Lemma $3.4.1$ with zeros. Specifically, we define $p$ as the uniform distribution over all inputs in the language $L_{LV} = L_1 \cdot \{0^W\}$.

That is, the input consist of all bit sequences in $L_1$, followed by a sequence of $W$ zeros. The trailing 0’s are used to force the algorithm to reach distinct configurations after reading the first $W$ input bits, as the algorithm is required to be correct on all prefixes of the input. As implied by the lemma, any deterministic algorithm which is always correct for a random instance requires at least $\left\lfloor \frac{W}{2W\epsilon} \right\rfloor$ bits, as it has to reach a distinct state for each input $s \in L_1$. The argument for lower bounds of $W \log \left\lfloor \frac{1}{2W\epsilon} + 1 \right\rfloor$ bits and $\log \left( 2^{\left\lfloor \frac{W}{2W\epsilon(1+\log W)} \right\rfloor - 1} - 1 \right) \cdot \left\lfloor \frac{2W\epsilon}{\log W} + 1 \right\rfloor$ bits (when $\epsilon \geq \log W 2W$) is similar, by using the padded versions of $L_2, L_3$ from lemmas $3.4.4$ and $3.4.6$. Notice that given these bounds, we get a $B$ lower bound similarly to before.

Next, we use the Minimax principle analogue for Monte Carlo algorithms \cite{46}, which states that for any input distribution $p$ and $\delta \in [0, 1/2]$, any randomized algorithm that is always (for any input) $\epsilon$-correct with probability at least $1 - \delta$ uses in expectation at least half as much memory as the optimal deterministic algorithm that errs (i.e., is not $\epsilon$-correct) with probability at most $2\delta$ on a random instance drawn according to $p$. Once again, we consider $p$ to be the uniform distribution over

$$L_{MC} = L_1 \cdot \{0^W\}.$$ 

Since the distribution is uniform, any deterministic algorithm, which is $\epsilon$-correct with probability at least $1 - 2\delta$ on a random instance drawn according to $p$, is actually $\epsilon$-correct on $1 - 2\delta$ fraction of the inputs. Similar to the LV case, the argument in Lemma $3.4.1$ implies that the algorithm must reach a distinct configuration after reading the first $W$ bits of each of the $(1 - 2\delta) \cdot |L_{MC}|$ inputs it is $\epsilon$-correct on. Consequently, the algorithm
must use at least \( \log((1 - 2\delta) \cdot |L_{MC}|) \) bits of memory. Applying the Minimax principle, the derived lower bound \( B_{MC} \) for any MC algorithm is:
\[
B_{MC} \geq \frac{1}{2} \log((1 - 2\delta) \cdot |L_{MC}|) \geq \frac{1}{2} \left\lceil \frac{W}{|2W\epsilon|} \right\rceil + \frac{1}{2} \log(1 - 2\delta) = \frac{1}{2} \left\lceil \frac{W}{|2W\epsilon|} \right\rceil - O(1).
\]

Once again, the case for \( \frac{1}{2}W \log \left\lceil \frac{1}{2W\epsilon} + 1 \right\rceil - O(1) \) lower bound is based on \( L_3 \) from Lemma 3.4.6, the case for a \( \frac{1}{2} \log \left( 2^{\left\lceil \frac{W}{2W(1+\log^{-1} W)} \right\rceil - 1} - 1 \right) \cdot \left\lfloor \frac{2W\epsilon}{\log W + 1} \right\rfloor \) - \( O(1) \) can be derived from \( L_2 \), and both follow from similar arguments.

\[\square\]

3.5 Upper Bounds

In this section, we provide algorithms that imply upper bounds on the memory needed for the Basic-Summing problem. We consider two cases, depending on whether \( \epsilon \) satisfies \( \epsilon^{-1} \leq W \left( 2 - \frac{1}{\log W} \right) \) or not, and propose a different algorithm for each case.

### 3.5.1 Large \( \epsilon \) values

In this section, we present an upper bound for the case of \( \epsilon^{-1} \leq W \left( 2 - \frac{1}{\log W} \right) \). In Section 3.5.2 we complete the picture by giving an alternative algorithm for smaller values of \( \epsilon \). Intuitively, we break the window into blocks of size \( T \triangleq \left\lfloor W (2\epsilon - 2^{-v}) \right\rfloor \), where the value of \( v \) is determined in Equation (3.11) below. Each block is represented by a single bit as described below, and the bits that represent the blocks that overlap with the window are stored in a cyclic bit array \( b \). Upon an element arrival, we first divide its value by \( R \) to scale it into a real number in \([0, 1]\). We then round the fractional value into \( v \) bits. By keeping only the rounded fraction we allow the algorithm’s state size to remain independent of \( R \). Within a block, we sum the rounded fractions using a fixed point variable \( y \); \( y \) is allocated with \( \lceil \log (2T) \rceil \) bits for its integral part and another \( v \) bits for the fractional value. Whenever a block ends, we compress its representation to a single bit that depends on the value of our counter \( y \). Specifically, if \( y < T \) the bit is set to zero; in the case of \( y \geq T \), we denote the block’s bit by 1 and subtract \( T \) from the value of \( y \). Since each rounded fraction’s value is at most 1, this ensures that the value of \( y \) is always smaller than \( 2T \) and thus its integral part can indeed be represented using \( \lceil \log (2T) \rceil \) bits.

Notice that at the end of each block, our approach “propagates” the current value of \( y \) to the following block. Interestingly, this propagation asymptotically reduces the memory requirement of the algorithm for a given error.

Our algorithm keeps the following variables:

- \( b \) - a bit-array of size \( k \triangleq \left\lceil \frac{W}{T} \right\rceil \).
- \( y \) - a counter for the sum of elements which is not yet accounted for in \( b \).
i - the index of the “oldest” block in b.

B - the sum of all bits in b.

m - a counter for the current offset within the block.

Our Basic-Summing algorithm is presented in Algorithm 2. We use Round_v (z) for some \( z \in [0,1] \) to denote rounding of \( z \) to the nearest value \( \tilde{z} \) such that \( 2^v \tilde{z} \) is an integer. Next follows an analysis of our algorithm.

**Theorem 3.5.1.** For any \( \epsilon^{-1} \leq W \left( 2 - \frac{1}{\log W} \right) \), Algorithm 2 provides an \( RW\epsilon \)-additive approximation for Basic-Summing.

**Proof.** First, let us introduce some notations used in the proof. Element \( j \) is marked \( x_j \) or \( x'_j \) after we divide it by \( R \) and round it. Assume that the index of the last element is \( W + m \), where \( x_W \) is the last element of a block. Block index \( i \) refers to its value after \( W + m \) elements have been processed. The setting for the proof is given in Figure 3.1.

We denote by \( y_j \) the value of \( y \) after adding \( x_j \). Since \( \forall x \in \{0,1,\ldots,R\} \) we have \( x' \in [0,1] \) (Line 3), and as every \( T \) integers we decrease the value of \( y \) by \( T \) if its value exceeds \( T \), we conclude that whenever a block ends, and specifically for \( y_0 \) we have:

\[
0 \leq y_0 \leq T - 2^{-v}. \tag{3.1}
\]

Our goal is to approximate

\[
S^W \triangleq \sum_{j=m+1}^{W+m} x_j. \tag{3.2}
\]

Algorithm 2 uses the following approximation to answer this query:

---

**Algorithm 2** Additive Approximation for Basic-Summing

1: Initialization: \( y \leftarrow 0, b \leftarrow 0, B \leftarrow 0, i \leftarrow 0, m \leftarrow 0. \)

2: function **Add**(ELEMENT \( x \))

3: \[ x' \leftarrow \text{Round}_v \left( \frac{x}{R} \right) \]

4: \( m \leftarrow (m + 1) \mod T \)

5: \( y \leftarrow y + x' \)

6: if \( m = 0 \) then

7: \( B \leftarrow B - b_i \)

8: \( b_i \leftarrow \left\lfloor \frac{y}{T} \right\rfloor \)

9: \( y \leftarrow y - T \cdot b_i \)

10: \( B \leftarrow B + b_i \)

11: \( i \leftarrow (i + 1) \mod k \)

12: end if

13: end function

14: function **Query**()

15: return \( R \cdot \left( T \cdot B + y - \frac{W}{2k} - m \cdot b_i + 2^{-v-1} \right) \)

16: end function
\[
\hat{S}_W = R \cdot \left( T \cdot B + y_{W+m} - \frac{W}{2k} - m \cdot b_i + 2^{-v-1} \right) \\
= R \cdot \left( T \cdot B + y_W + \sum_{j=W+1}^{W+m} x_j' - \frac{W}{2k} - m \cdot b_i + 2^{-v-1} \right). 
\]

(3.3)

At the end of block \( j \), if \( y \) is decreased by \( T \), then \( b_j \) is set and will not be cleared before time \( W + m \). Therefore

\[
T \cdot B + y_W = y_0 + \sum_{j=1}^{W} x_j'.
\]

Substituting \( \frac{W}{k} \cdot B + y_W \) by \( y_0 + \sum_{j=1}^{W} x_j' \) in (3.3) gives us

\[
\hat{S}_W = R \cdot \left( y_0 + \sum_{j=1}^{W} x_j' + \sum_{j=W+1}^{W+m} x_j' - \frac{W}{2k} - m \cdot b_i + 2^{-v-1} \right) \\
= R \cdot \left( y_0 + \sum_{j=1}^{m} x_j' + \sum_{j=m+1}^{W+m} x_j' - \frac{W}{2k} - m \cdot b_i + 2^{-v-1} \right). 
\]

(3.4)

Denote the rounding error over the entire window

\[
\xi \triangleq \sum_{j=m+1}^{W+m} x_j - R \cdot \sum_{j=m+1}^{W+m} x_j'.
\]

From Line 3, we know that \( x_j' = \text{Round}_v \left( \frac{x_j}{R} \right) \), while \( \left| \frac{x_j}{R} - \text{Round}_v \left( \frac{x_j}{R} \right) \right| \leq 2^{-1-v} \), and thus:

\[
-2^{-1-v} \cdot RW \leq \xi \leq 2^{-1-v} \cdot RW 
\]

(3.5)

Thus, we can write (3.4) as:

\[
\hat{S}_W = \sum_{j=m+1}^{W+m} x_j + \xi + R \cdot \left( y_0 + \sum_{j=1}^{m} x_j' - \frac{W}{2k} - m \cdot b_i + 2^{-v-1} \right).
\]

Plugging in (3.2), we get

\[
\hat{S}_W = S^W + \xi + R \cdot \left( y_0 + \sum_{j=1}^{m} x_j' - \frac{W}{2k} - m \cdot b_i + 2^{-v-1} \right).
\]
Therefore, the error is
\[
\widehat{S^W} - S^W = \xi + R \cdot \left( y_0 + \sum_{j=1}^{m} x'_j - \frac{W}{2k} - m \cdot b_i + 2^{-v-1} \right).
\] (3.6)

We consider two cases:

- \( b_i = 1 \): This means that \( y \) has crossed the threshold by time \( T \), i.e.,
  \[
y_0 + \sum_{j=1}^{T} x'_j \geq T \quad \text{and equivalently} \quad y_0 + \sum_{j=1}^{m} x'_j \geq T - \sum_{j=m+1}^{T} x'_j.
  \]

Going back to (3.6), we get on one side
\[
\widehat{S^W} - S^W = \xi + R \cdot \left( y_0 + \sum_{j=1}^{m} x'_j - \frac{W}{2k} - m \cdot b_i + 2^{-v-1} \right) \\
\geq \xi + R \cdot \left( T - \sum_{j=m+1}^{T} x'_j - \frac{W}{2k} - m + 2^{-v-1} \right) \\
\geq \xi + R \cdot \left( W \frac{2k}{2k} - \left( \sum_{j=m+1}^{T} 1 \right) - m + 2^{-v-1} \right) \geq \xi - R \cdot \left( \frac{W}{2k} - 2^{-v-1} \right) \quad (3.7)
\]

In the beginning of the proof we noted that the value of \( y \) at the end of a block never exceeds \( T \). Therefore, (3.6) can be bounded as follows:
\[
\widehat{S^W} - S^W = \xi + R \cdot \left( y_0 + \sum_{j=1}^{m} x'_j - \frac{W}{2k} - m \cdot b_i + 2^{-v-1} \right) \\
\leq \xi + R \cdot \left( y_0 - \frac{W}{2k} + 2^{-v-1} \right),
\]
and according to (3.1) we have
\[
\widehat{S^W} - S^W \leq \xi + R \cdot \left( y_0 - \frac{W}{2k} + 2^{-v-1} \right) \leq \xi + R \cdot \left( \frac{W}{2k} - 2^{-v-1} \right). \quad (3.8)
\]

- \( b_i = 0 \): Similarly, this means that \( y \) was smaller than the threshold at the end of block \( i \). Hence \( y_0 + \sum_{j=1}^{T} x'_j \leq T - 2^{-v} \) or equivalently
  \[
y_0 + \sum_{j=1}^{m} x'_j \leq T - \sum_{j=m+1}^{T} x'_j - 2^{-v}.
  \]

Thus, the upper bound is
\[
\widehat{S^W} - S^W = \xi + R \cdot \left( y_0 + \sum_{j=1}^{m} x'_j - \frac{W}{2k} + 2^{-v-1} \right) \\
\leq \xi + R \cdot \left( T - \sum_{j=m+1}^{T} x'_j - 2^{-v} - \frac{W}{2k} + 2^{-v-1} \right) \leq \xi + R \cdot \left( \frac{W}{2k} - 2^{-v-1} \right). \quad (3.9)
\]
Our error is hence bounded from below by
\[
\hat{S}^\text{W} - S^\text{W} = \xi + R \cdot \left( y_0 + \sum_{j=1}^{m} x'_j - \frac{W}{2^k} + 2^{-v-1} \right) \geq \xi - R \cdot \left( \frac{W}{2^k} - 2^{-v-1} \right).
\]

We need to bound (3.7), (3.8), (3.9) and (3.10) by \(R W \epsilon\). Using (3.5) we can imply a summing error of at most \(R W \epsilon + 2^{-v-1} R\) if we require \(2^{-v} RW + \frac{RT}{2} \leq R W \epsilon\), or equivalently, \(T \leq W (2 \epsilon - 2^{-v})\). We thus choose \(T = \lfloor W (2 \epsilon - 2^{-v}) \rfloor\) and the number of blocks becomes
\[
k = \left\lfloor \frac{W}{W (2 \epsilon - 2^{-v})} \right\rfloor.
\]

We got an approximation whose additive error that is strictly lower than \(R W \epsilon\), as required. Finally, we choose the number of bits representing the fractional value of \(y\) to be
\[
v \triangleq \lceil \log \left( \epsilon^{-1} \log W \right) \rceil \geq \log \left( \epsilon^{-1} \log W \right).
\]

Therefore, the number of blocks is
\[
k = \left\lfloor \frac{W}{W (2 \epsilon - 2^{-v})} \right\rfloor = \left\lfloor \frac{W}{W (2 \epsilon - \frac{\epsilon}{\log W})} \right\rfloor.
\]

To have a positive block size, it suffices to require
\[
W \left( 2 \epsilon - \frac{\epsilon}{\log W} \right) \geq 1,
\]

or equivalently, \(\epsilon^{-1} \leq W \left( 2 - \frac{1}{\log W} \right)\).

The following theorem analyzes the memory requirements of our algorithm.

**Theorem 3.5.2.** For any \(\epsilon^{-1} \leq W \left( 2 - \frac{1}{\log W} \right)\), Algorithm 2 requires \(\lceil (2 \log W + \epsilon^{-1}/2) (1 + o(1)) \rceil\) bits.

**Proof.** We represent \(y\) using \([\log 2T]+v = [1 + \log \lfloor W (2 \epsilon - 2^{-v}) \rfloor]+v \leq [2 + \log (W \epsilon) + v]\) bits, \(m\) using \([\log T] \leq [1 + \log (W \epsilon)]\) bits and \(b\) using \(k\) bits. Additionally, \(i\) requires \([\log k]\) bits, and \(B\) another \([\log (k+1)]\) bits. Overall the number of bits required is
\[
k + [\log (W \epsilon) + v] + [\log (W \epsilon)] + [\log k] + [\log (k+1)] + O(1) = k + 2 \log (W \epsilon) + v + 2 \log k + O(1).
\]

The number of blocks is

26
\[
\begin{align*}
k &= \left\lceil \frac{W}{W(2\epsilon - 2^{-\nu})} \right\rceil \leq 1 + \frac{W}{W(2\epsilon - 2^{-\nu}) - 1} \leq 1 + \frac{W}{W(2\epsilon - \frac{\epsilon}{\log W}) - 1} \\
&\leq 1 + \frac{1}{2}\epsilon^{-1} \cdot \frac{2W \log W}{2W \log W - W - 1} \leq 1 + \frac{1}{2}\epsilon^{-1} \cdot \left( 1 + \frac{W + 1}{2W \log W - W - 1} \right) \\
&= 1 + \frac{1}{2}\epsilon^{-1}(1 + o(1)).
\end{align*}
\]

Thus, the space consumption becomes
\[
\begin{align*}
k + 2 \log (W\epsilon) + \nu + 2 \log k + O(1) \\
&= \left( \frac{\epsilon^{-1}}{2} + 2 \log W - 2 \log \epsilon^{-1} + \lceil \log (\epsilon^{-1} \log W) \rceil + 2 \log \left( 1 + \frac{\epsilon^{-1}}{2} \right) \right) \cdot (1 + o(1)) \\
&= \left( \frac{\epsilon^{-1}}{2} + 2 \log W - \log \epsilon^{-1} + 2 \log \left( 1 + \frac{\epsilon^{-1}}{2} \right) \right) (1 + o(1)) \\
&\leq \left( \frac{\epsilon^{-1}}{2} + 2 \log W + 2 \log \left( 1 + \frac{\epsilon^{-1}}{2} \right) \right) (1 + o(1)).
\end{align*}
\]

There are two cases. If \(\epsilon^{-1} \leq \log W - 1\), the space is less than or equal to
\[
\left( \frac{\epsilon^{-1}}{2} + 2 \log W + 2 \log \log W \right) (1 + o(1)) = \left( 2 \log W + \frac{1}{2\epsilon} \right) (1 + o(1)).
\]

Otherwise,
\[
\left( \frac{\epsilon^{-1}}{2} + 2 \log W + \epsilon^{-1} \cdot \frac{2}{\epsilon^{-1}} \log \left( 1 + \frac{\epsilon^{-1}}{2} \right) \right) (1 + o(1)) \\
\leq \left( \frac{\epsilon^{-1}}{2} + 2 \log W + \epsilon^{-1} \frac{\log \log W}{(\log W - 1)} \right) (1 + o(1)) = \left( 2 \log W + \frac{1}{2\epsilon} \right) (1 + o(1)). \quad \Box
\]

Finally, we conclude that our algorithm is succinct when \(\epsilon = o(\log^{-1} W)\) and that it is asymptotically optimal otherwise.

**Theorem 3.5.3.** For any \(\epsilon^{-1} \leq W \left( 2 - \frac{1}{\log W} \right)\), Algorithm 2 uses \(2 + o(1)\) as many bits as the lower bound \(B\) of Theorem 3.4.7. Further, when \(\epsilon = o(\log^{-1} W) \land (\epsilon = \omega (W^{-1}) \lor 2W\epsilon \in \mathbb{N})\) it is succinct and uses no more than \(B(1 + o(1))\) bits.

### 3.5.2 Small \(\epsilon\) Values

Algorithm 2 only works for \(\epsilon^{-1} \leq W \left( 2 - \frac{1}{\log W} \right)\); otherwise, the blocks become of size 0. To complete the picture, we present Algorithm 3 that works for smaller errors. Intuitively, we keep an array \(b\) of size \(W\), such that every cell represents the number of integer multiples of \(RT\) in an arriving item. Here, we use \(T \triangleq W \left( 2\epsilon - 2^{-\nu} \right)\) and it is no longer an integer. Similarly to the above algorithm, we reduce the error by tracking the remainder in a variable \(y\), propagating uncounted fractions to the following item. In this case as well,
this approach reduces the error compared with keeping a $W$-sized array of rounded values for approximating the sum. Each cell in $b$ needs to represent a value in $\{0, 1, \ldots, \lfloor \frac{k}{W} \rfloor \}$; the remainder $y$ is now a fractional number, represented using $v$ bits. When a new item is added, we scale it, add the result to $y$, and update both $b_i$ and the remainder.

**Algorithm 3** Additive Approximation for Basic-Summing with Small Error

1: Initialization: $y \leftarrow 0, b \leftarrow 0, B \leftarrow 0, i \leftarrow 0.$

2: function **Add(element $x$)**

3: \[ x' \leftarrow \text{Round}_v \left( \frac{x}{R} \right) \]

4: \[ B = B - b_i \]

5: \[ b_i \leftarrow \left\lfloor \frac{y + x'}{T} \right\rfloor \]

6: \[ y \leftarrow y + x' - b_i \cdot T \]

7: \[ B \leftarrow B + b_i \]

8: \[ i \leftarrow (i + 1) \mod W \]

9: end function

10: function **Query()**

11: return $R \cdot \left( T \cdot B + y - \frac{W}{2k} + 2^{-v-1} \right)$

12: end function

**Theorem 3.5.4.** Algorithm 3 provides an $RW\epsilon$-additive approximation for BASIC-SUMMING.

**Proof.** The notations for this proof are the same as in Theorem 3.5.1. In this case all ”blocks” are of size 1 so $m$ is always 0. Our estimator in Algorithm 3 is

\[
\hat{S}^W = R \cdot \left( T \cdot B + y_W - \frac{W}{2k} + 2^{-v-1} \right). \tag{3.12}
\]

If at time $j$ the value of $y$ is decreased by $\alpha \cdot T$, then $b_j$ is set to $\alpha$ and will not change before time $W$. Therefore we have

\[ T \cdot B + y_W = y_0 + \sum_{j=1}^{W} x_j'. \]

Substituting $T \cdot B + y_W$ in (3.12) we get

\[
\hat{S}^W = R \cdot \left( y_0 + \sum_{j=1}^{W} x_j' - \frac{W}{2k} + 2^{-v-1} \right). \tag{3.13}
\]

Notice that every element $x_j$ is divided by $R$ and rounded into $v$ bits; thus, if we denote

\[
\xi \triangleq \sum_{j=m+1}^{W+m} x_j - R \cdot \sum_{j=m+1}^{W+m} x_j',
\]

we have that $-2^{-1-v} \cdot RW \leq \xi \leq 2^{-1-v} \cdot RW$. Therefore, we can express our estimation error as

\[
\hat{S}^W - S^W = \xi + R \cdot \left( y_0 - \frac{W}{2k} + 2^{-v-1} \right), \tag{3.13}
\]

28

Technion – Computer Science Department - Ph.D. Thesis PHD-2018-08 - 2018
To bound the error we use the fact that $0 \leq y \leq T - 2^{-\nu}$. Therefore, (3.13) can be bounded as follows:

$$\xi - R \cdot \frac{W}{2k} < +2^{-\nu - 1} \leq S^{W} - S^{W} \leq \xi + R \cdot \frac{W}{2k} - 2^{-\nu - 1} < \xi + R \cdot \frac{W}{2k}.$$ 

$T$ remains $\lfloor W (2\epsilon - 2^{-\nu}) \rfloor$ and the absolute error is strictly lower than $RW\epsilon$, similarly to Theorem 3.5.1.

We proceed with a space analysis of the algorithm. Notice that for $\epsilon < 1/2W$, Algorithm 3 requires less space than Algorithm 2.

**Theorem 3.5.5.** For any $\epsilon^{-1} > 2W \left( 1 - \frac{1}{\log W} \right) = 2W (1 - o(1))$, the number of memory bits used by Algorithm 3 is

$$W \log \left( \frac{1}{2W \epsilon} + 1 \right) \cdot (1 + o(1)) \leq \frac{1}{2\epsilon} \cdot (1 + o(1)).$$

**Proof.** We represent $y$ using $\nu$ bits and $b$ using $W \lceil \log \left( \frac{k}{W} + 1 \right) \rceil$ bits. Additionally, $i$ requires $\lceil \log W \rceil$ bits, and $B$ another $\lceil \log (W + 1) \rceil$ bits. Overall the number of bits required is

$$\nu + W \left[ \log \left( \frac{k}{W} + 1 \right) \right] + \lceil \log W \rceil + \lceil \log (W + 1) \rceil = \left( W \log \left( \frac{k}{W} + 1 \right) + \nu + 2 \log W \right) (1 + o(1)).$$

We choose the number of bits representing the fractional value of $y$ to be $\nu = \lceil \log (\epsilon^{-1}W) \rceil$. We have also chosen $T = \lfloor W (2\epsilon - 2^{-\nu}) \rfloor$. Thus, $T \geq \lfloor W (2\epsilon - \epsilon/\sqrt{W}) \rfloor \geq \epsilon (2W - 1)$ and the space consumption is

$$\left( W \log \left( \frac{k}{W} + 1 \right) + \nu + 2 \log W \right) (1 + o(1))$$

$$= \left( W \log \left( \frac{k}{W} + 1 \right) + \log (\epsilon^{-1}W) + 2 \log W \right) (1 + o(1))$$

$$\leq \left( W \log \left( \frac{1}{2W \epsilon - \epsilon} + 1 \right) + \log (\epsilon^{-1}) + 3 \log W \right) (1 + o(1))$$

$$\leq \left( W \log \left( \frac{1}{2W \epsilon} + 1 \right) + \log \left( \frac{1}{W \epsilon} \right) + 4 \log W \right) (1 + o(1))$$

$$\leq \left( W \log \left( \frac{1}{2W \epsilon} + 1 \right) \right) (1 + o(1)) \leq \left( \frac{W}{2W \epsilon} \right) (1 + o(1)) = \frac{\epsilon^{-1}}{2} (1 + o(1)). \quad \square$$

We conclude the section by noting that Algorithm 3 is succinct, requiring only $(1 + o(1))$ times as much memory as the lower bound proved in Theorem 3.4.7.

**Theorem 3.5.6.** Denote by $B \equiv W \log \left( \frac{1}{3W \epsilon} + 1 \right)$ the lower bound of Lemma 3.4.6. For any $\epsilon^{-1} > 2W \left( 1 - \frac{1}{\log W} \right) = 2W (1 - o(1))$, Algorithm 3 provides an $RW \epsilon$ additive approximation to Basic-Summing using $B \cdot (1 + o(1))$ memory bits.
3.6 Discussion

In this chapter, we investigated additive approximations for the Basic-Summing problem in which we track the sum of the last $W$ elements in a stream of integers in $\{0, 1, \ldots, R\}$. We provided several lower bounds for a $RW\epsilon$-additive approximation, each of which is higher than the rest for some parameter combinations. Matching these lower bounds, we provided two efficient algorithms; these algorithms are succinct for many parameter combinations and use at most $2 + o(1)$ times the required memory in all others. Further, our algorithms are deterministic and operate in constant time, while our lower bounds also apply for randomized algorithms with no time restrictions.
Chapter 4

Randomized Admission Policies

4.1 Introduction

Network management protocols often require timely and meaningful insight about per flow network traffic. This chapter introduces Randomized Admission Policy (RAP) – a novel algorithm for the frequency and top-$k$ estimation problems, which are fundamental in network monitoring. We demonstrate space reductions compared to the alternatives by a factor of up to 32 on real packet traces and up to 128 on heavy-tailed workloads. For top-$k$ identification, RAP exhibits memory savings by a factor of between 4 and 64 depending on the workloads’ skewness. These empirical results are backed by formal analysis, indicating the asymptotic space improvement of our probabilistic admission approach. Additionally, we present $d$-Way RAP, a hardware friendly variant of RAP that empirically maintains its space and accuracy benefits.

4.1.1 Background

Network management and traffic engineering protocols rely on flow counters based network monitoring. Examples include effective routing, load balancing, QoS enforcement, network caching, anomaly detection and intrusion detection [47, 48, 49, 50, 51, 52]. Typically, monitoring utilities track millions of flows [53, 54], and the counter of a monitored flow is updated on the arrival of each of its packets. Often, the most frequently appearing flows, known as heavy hitters, are also the most interesting, since their impact on the above is the most crucial.

Maintaining such counters is a challenging task with today’s storage technology. The difficulty arises as DRAM is too slow to keep up with line rates, while the faster SRAM is expensive and thus too small for keeping an exact counter for each flow. These limitation were tackled using various approaches.

Estimators reduce the size of counters using probabilistic techniques [22, 26, 24]. This enables maintaining one counter per flow in SRAM at the cost of reduced accuracy. The
downside of estimators is that they require an explicit flow to counter mapping for every flow. This mapping often becomes the dominant factor in memory consumption [55].

The shared counters approach, also known as sketches, solves the mapping problem using hashing algorithms that implicitly assign flows to counters. Well known examples include Multi Stage Filters [31] and Count Min Sketch [28]. Yet, to reduce the impact of hash collisions on counters’ reading accuracy, these methods must allocate considerably more space and more counters than predicted by lower bounds.

Databases and data analytics face similar problems, known in these domains as frequency estimation and top-k identification, i.e., identifying who are the k most frequent flows. These domains typically favor counter based solutions over sketches since the former are considered superior to sketches, both asymptotically and in practice [37] [56]. Counter based algorithms maintain a fixed size set of counters and aspire to allocate these counters only to the more frequent flows. These include Lossy Counting [35], Frequent [29] and Space Saving [30]. The latter is also considered state of the art [37] [56] [57]. Alas, these algorithms cannot be easily ported into networking devices as they utilize complex data structures and dynamic memory allocation.

Another significant shortcoming of counter based solutions is that they update the state of allocated counters on the arrival of each packet belonging to a unmonitored flow, regardless of how frequent this flow is. Doing so hurts their space to accuracy tradeoff to the point that they become ineffective on heavy-tailed workloads, which are common in network switches and routers.

Contributions

In this work, we promote the concept of using a randomized admission policy for allocating counters to non-monitored flows, and show that it significantly improves accuracy. Intuitively, our policy ignores most of the tail flows and is still able to eventually admit the high frequency flows.

Specifically, this idea is realized in a novel counter based algorithm called Randomized Admission Policy (RAP) as well as a hardware friendly variant called d-Way associative RAP (dW-RAP). RAP is simpler to analyze, while dW-RAP maps well into limited associativity cache designs and empirically maintains most of the benefits of RAP. We extensively evaluate RAP and dW-RAP over two real packet traces [58] [59], a YouTube access trace [60] and synthetic Zipf distributions.

For the frequency estimation problem, RAP and dW-RAP achieve the same mean square error (MSE) as the leading alternatives while using a fraction of the required memory. For top-k identification, RAP and dW-RAP exhibit significantly higher recall and precision, even when allocated with half the space given to the alternative methods. In particular, when the distribution is only mildly skewed (or heavy-tailed), RAP and dW-RAP are the only techniques that successfully identify a high percentage of the top-k flows.
4.2 Related Work

The frequent items and top-\(k\) identification problems appear in slight variations across multiple domains. Algorithms for these problems are often categorized as either counter based or sketch based. In addition, the specific challenges of network monitoring have spawned solutions that are especially tailored for the memory limitations in the networking case.

4.2.1 Counter based algorithms

Counter based algorithms are usually designed for software implementations and maintain a table of monitored items. The differences between these algorithms lie in the question of admission and eviction of entries to and from the table. From a networking perspective, counter based algorithms maintain an explicit flow to counter mapping for monitored items. For a stream of \(N\) events and an accuracy parameter \(\varepsilon\), the goal is to approximate a given flow’s frequency to within an additive error of \(N \cdot \varepsilon\). For this task, \(\Omega(\frac{1}{\varepsilon})\) counters are required \cite{30}, and this is achieved by some of the algorithms below.

*Lossy Counting* \cite{35} increments an arriving item’s counter on every arrival. If the counter is not in the table, it is admitted with a counter value of 1. Lossy Counting keeps the table size bounded by periodically decrementing table counters and evicting items whose counter reaches 0. Unfortunately, Lossy Counting requires a maximal number of \(\frac{1}{\varepsilon} \cdot \log(N)\) table entries. *Probabilistic Lossy Counting* \cite{33} requires fewer table entries on average but only provides a probabilistic guarantee.

In *Frequent (FR)* \cite{36,29}, whenever an item arrives and the table already contains \(\frac{1}{\varepsilon}\) entries, the item is not admitted. Instead, FR decrements every entry in the table, evicting entries whose counter reached 0. The main benefit of FR is that it requires the optimal number of \(O(\frac{1}{\varepsilon})\) table entries.

*Space Saving (SS)* \cite{30} requires the same number of entries as FR, but maintains additional information to improve accuracy. Space Saving admits all arriving items at the expense of evicting the minimum-frequency item. Alternatively, \cite{44} admits all entries and performs a periodic cleanup. Space Saving is considered to be state of the art \cite{37,56,57}.

4.2.2 Sketch based algorithms

*Sketches*, such as *Multi Stage Filters* \cite{31}, *Count Sketch* \cite{27} and *Count Min Sketch* \cite{28}, are very common in networking domains as they are simple to implement in hardware and have low implementation overheads. The most popular example, Count Min Sketch, provides the following guarantee — given an item \(x\), with probability of at least \(1 - \delta\), the estimation error of \(x\) is at most \(N \cdot \varepsilon\).

Count Min Sketch does not require storing flow identifiers or maintaining a flow to counter association. Instead, it maintains an array of \(\ln(\frac{1}{\varepsilon})\) rows, each with \(\frac{\varepsilon}{\varepsilon}\) counters.
When an item arrives, a hash function is calculated for each row and its corresponding counter is incremented. To estimate the frequency of an item, the corresponding counters are read and the minimum counter value is returned as the estimation.

Asymptotically, Count Min Sketch requires a suboptimal number of counters. Yet, it does not store flow ids and has minor hardware implementation overheads. Further, sketches require a sub-linear number of counters, can completely reside in SRAM, and provide online frequency estimation.

On the contrary, Counter Braids [55] and Counter Tree [61] use a hierarchical sketch where overflowing counters are hashed to a higher level sketch. They encode items just like Count Min Sketch would, but decoding is complex and can only be performed offline, estimating all flow values together.

In Randomized Counter Sharing [62], every time an item is added, a random hash function is used and the corresponding counter is incremented. The flow identifier is recorded, but without an explicit mapping to frequency. When a measurement ends, we estimate the flow’s frequency by summing all of the corresponding counters or by performing a maximum-likelihood estimation. Both of these estimations are quite slow and cannot be performed online.

In summary, sketches are space suboptimal and only solve the frequency estimation problem. Further, they only support point queries and their answers are only correct within a certain probability. Despite these limitations, sketches are used for many networking applications [32, 28, 33, 31, 34, 14, 63].

### 4.2.3 Network monitoring architectures

In hybrid SRAM/DRAM architectures [54, 53], the LSB bits of counters are stored in SRAM and the MSB in DRAM. This way, the space allocated for each flow in SRAM is small. However, the SRAM counters have to periodically be synchronized with the DRAM counters, which increases the contention on the memory bus. Further, estimating a flow’s frequency requires accessing DRAM and therefore cannot be used for online network monitoring.

Brick [64] uses an efficient encoding in order to reduce the number of bits allocated per counter. Brick is most effective when there are many very small flows.

Estimators use fixed size small counters in order to represent large numbers. These methods trade precision for space and allow more counters to be contained in SRAM. This idea was first introduced by Approximate Counting [1] and was adapted to networking devices [25, 24, 26, 22]. The downside of estimators is that they require storing a flow-to-counter mapping for every flow, a requirement that has many overheads. Sampling techniques are another alternative that trades accuracy for space. Unfortunately, these methods can only monitor large flows that are frequent enough to be sampled [65, 66].
The throughput of some hardware implementations suffers from the need to perform reads on every update when incrementing counters. This can be addressed by probabilistic multiplicity counting (PMC), as suggested in [67]. In PMC, a special coding scheme is devised in which a hashing function is used to set at most one bit on the arrival of each packet, without reading any entry in the data structure. Hence, this enables counting with write-only memory accesses. Yet, estimating a flow’s frequency requires the equivalent of scanning a table, which may be too slow for online decision making.

4.3 Randomized Admission Policy (RAP)

RAP maintains a table \( C \) which contains \( M \triangleq \frac{1}{\epsilon} \) entries. The intuition behind RAP is to minimize the error inflicted upon arrival of a non-monitored item \( x \notin C \) when the table is full. That is, we identify inefficiencies in the way previous works behave in this case. E.g., FR needlessly increases the error of all counters by decrementing all of them. In contrast, Space Saving always evicts the item with the minimal counter. This eviction introduces an error, as the monitored element is often more frequent than a randomly arriving item without a counter. This is especially true for heavy-tailed workloads, where a large fraction of the stream consists of “tail elements” that should not be admitted into the table. In RAP, we take a more conservative approach. When an item \( x \notin C \) arrives, we find the item \( (m) \) with the minimal counter value \( (c_m) \). \( x \) is then admitted into \( C \) with probability \( \frac{1}{c_m+1} \) at the expense of \( m \); otherwise, \( x \) is simply discarded. Algorithm 4 provides a pseudo code of the RAP’s Add method.

In order for an item \( x \) to replace the minimal element \( m \), an unallocated item has to arrive \( c_m + 1 \) times on average. Infrequent items are therefore unlikely to be admitted into \( C \), and most of them will not affect any of the counters. Therefore, RAP is considerably more accurate, especially for heavy tailed workloads where a large portion of the items are infrequent. In contrast, every tail item in Space Saving affects the counters, thereby contributing to the total estimation error. Our approach is not without risks, as if an arriving item turns out to be frequent, Space Saving admits that item sooner than RAP.

Given a query for the frequency of element \( x \), RAP estimates it as \( c_x \) if \( x \in C \) and 0 otherwise.

RAP can be implemented with existing data structures and it processes packets at \( O(1) \) runtime [30, 14]. It stores a single counter per table entry, while Space Saving entries are slightly larger as they store two values.

4.3.1 Analysis

We start our analysis with theoretic bounds for the top-\( k \) problem. These show that our probabilistic approach is asymptotically better for i.i.d. streams. We then explore the properties of RAP for the frequency estimation problem.
Algorithm 4 Randomized Admission Policy

Initialization: $C \leftarrow \emptyset$, $\forall i : c_i \leftarrow 0$

1: function ADD(Item $x$)
2: if $x \in C$ then
3:   $c_x \leftarrow c_x + 1$
4: else
5:   if $|C| < M$ then
6:     $c_x \leftarrow 1$
7:     $C \leftarrow C \cup \{x\}$
8:   else
9:     $m \leftarrow \text{argmin}_{y \in C} c_y$
10:    if random() < $\frac{1}{c_m + 1}$ then \(\triangleright \text{w.p } \frac{1}{c_m + 1}\)
11:       $C \leftarrow (C \setminus \{m\}) \cup \{x\}$
12:       $c_x \leftarrow c_m + 1$
13:     end if
14: end if
15: end if
16: end function

Top-$k$ Problem

We say that an algorithm successfully solves top-$k$ if it identifies the $k$ most frequent flows in a stream. Our goal is to bound the number of table entries required for successful identification of top-$k$.

Denote $\Gamma_\alpha(D) \triangleq \sum_{i=1}^{D} i^{-\alpha}$. A stream will be called an i.i.d. Zipf stream with skew $\alpha$ over domain $D$ if all of its elements are sampled independently and follow the distribution in which item $i \in \{1, 2, \ldots, D\}$ appears with probability $\frac{i^{-\alpha}}{\Gamma_\alpha(D)}$. Such a stream is denoted $Z^D_\alpha$.

**Theorem 4.3.1.** Let $\alpha < 1$. For any fixed $k$, Space Saving requires $O(D^{1-\alpha})$ counters to solve top-$k$ on $Z^D_\alpha$, while a randomized admission policy requires only $O(D\frac{1}{1+\alpha})$.

Next, we analyze the performance of a probabilistic admission filter for streams with higher skew. The proofs of both theorems appear in Section 4.6.

**Theorem 4.3.2.** For $Z^D_1$ and any fixed $k$, Space Saving requires $O(\log D)$ counters to solve top-$k$, while a randomized admission policy reduces the required number to $O(\sqrt{\log D})$.

Frequency Estimation Problem

We now present a brief mathematical analysis of RAP, including deterministic and probabilistic upper bounds for the estimation error.
Theorem 4.3.3. Let \( f_x \) be the true frequency of \( x \), \( \hat{f}_x \) be RAP’s estimation of \( f_x \), and \( m = \min_{y \in C} C_y \). Then \( \hat{f}_x \leq f_x + m \).

Proof. The proof is by a case analysis. First, suppose \( x \notin C \) at the time of the query. In this case, \( \hat{f}_x = 0 \) and the claim trivially holds. Conversely, assume that \( x \in C \) at the time of the query; consider the last time \( t \) in which \( x \) was admitted into \( C \). At that point, \( c_x^t = m_t - 1 \) where \( c_x^t \) is the value of \( x \)’s counter at time \( t \) and \( m_t \) is the minimum counter in the table just before time \( t \). Notice that the algorithm can only increase the minimum counter in the table due to a packet arrival, at which point either no counter changes or the minimal counter is incremented. Hence, \( c_x^t \leq m_t - 1 \). Next, suppose that \( x \) has arrived exactly \( n \) times between \( t \) and the present; \( n \leq f_x - 1 \) since we know that at time \( t \), \( x \) arrived once. It follows that \( c_x = c_x^t + n \leq m_t - 1 + f_x - 1 = f_x + m_t - f_x + m \).

Next, we show that the estimation given by RAP is in expectation smaller than or equal to the true frequency.

Theorem 4.3.4. \( \mathbb{E} \left[ \hat{f}_x \right] \leq f_x \).

Proof. In this proof, we use the notion of time to describe the events in the stream. The first event is at \( t = 0 \), the next at \( t = 1 \) and so on. We prove the claim by induction on the time \( t \). Base: At time 0, \( \hat{f}_x = 0 \). Step: If at time \( t \) an item different than \( x \) has arrived, then \( \mathbb{E} \left[ \hat{f}_x^t \right] \leq \mathbb{E} \left[ \hat{f}_x^{t-1} \right] \); in case \( c_x \) was the smallest counter at time \( t - 1 \), its estimation can only decrease, and otherwise its estimation does not change. However, if \( x \) arrived at time \( t \), let \( \Delta E_x^t \) be the change in \( \mathbb{E} \left[ \hat{f}_x \right] \), that is \( \Delta E_x^t = \mathbb{E} \left[ \hat{f}_x^t - \hat{f}_x^{t-1} \right] \). There can be two cases: if \( x \in C \), then \( \hat{f}_x^t - \hat{f}_x^{t-1} = 1 \). Otherwise \( x \notin C \), hence its estimation is either increased by \( c_m + 1 \) with probability \( \frac{1}{c_m + 1} \) or remains the same. Thus, in all cases \( \hat{f}_x^t - \hat{f}_x^{t-1} \) grows by 1 in expectation and \( \Delta E_x^t = 1 \). Hence, the induction hypothesis holds and \( \mathbb{E} \left[ \hat{f}_x \right] \leq f_x \).

4.4 Hardware Friendliness

RAP can be efficiently implemented in software with existing data structures [30, 14]. These complex data structures might be difficult to efficiently implement in hardware.

In this section, we present \( d \)-Way Randomized Admission Policy (\( d \)W-RAP), a hardware friendly variant of RAP. We describe \( d \)W-RAP as a cache management policy. Caches are well understood, making \( d \)W-RAP implementation as a cache policy easy to design as it does not rely on complex data structures. In addition, caches have a proven capability to operate at line speed. For self containment, Section 4.4.1 provides a brief introduction to cache topology.
4.4.1 Cache Memory Organization

In order to meet their high speed requirements, hardware caches are usually not fully associative. As a rule of thumb, the higher the associativity level – the slower the cache is since the search process becomes more complex. Limited associativity means that each item can only be placed in a certain logical place in the cache. If this place is already full, an existing item must be evicted in order to admit the new one.

These logical locations are called sets and in each set there are a certain number of places called ways. We use a hash function \( \text{Set}(x) \) to map an item to a certain set number; the item can only be stored in that set. This makes the lookup process simpler as we only need to search for the item in a specific set, rather than in the entire cache.

The more ways we add to the cache – the slower the cache works, as there are more places that an item could be found in. Therefore, to ensure fast performance, the number of ways is kept small, typically \( 2 - 32 \). A cache with \( d \) different ways is called \( d \)-way set associative; a cache with only a single set is called fully associative; a cache with a single way is called direct mapped.

Figure 4.1 illustrates the basic topology of a 4-way set associative cache. In this example, the \( \text{Set} \) function is used to determine the set for \( x \). The set selected is the one marked with orange (horizontal line) and since the cache has 4 ways, \( x \) could be placed in either of these ways. The cache first checks whether \( x \) appears in these ways. If it is not found, a cache policy is used to decide whether to admit \( x \) into the cache, at the expense of evicting some other item, or not.

![Figure 4.1: A 4-way set associative cache with 8 lines. When an item \( x \) arrives, \( \text{Set}(x) = 2 \) is calculated and the item can be stored in any of the ways of set 2.](image)

4.4.2 Cache Policy

A fundamental cache management question is what to do when an item arrives and its corresponding set is full. A cache policy is an algorithm that answers these questions. Cache policies can sometimes be partitioned into two sub policies: an admission policy and an eviction policy \[52\]. The former decides whether to admit an item into the cache and the latter decides on the cache victim.
4.4.3  \(d\)W-RAP as a Cache Policy

Algorithm \[4\] implements RAP assuming (implicitly) a fully associative memory organization. We now describe \(d\)W-RAP as a cache policy for a \(d\)-way cache organization.

**Metadata**  In \(d\)W-RAP, each entry contains a counter that is used for both frequency/top-
\(k\) estimation, and for the cache admission and eviction policies.

**Metadata Update**  In \(d\)W-RAP, every time a cached item is accessed, including right after the initial admission, its counter is incremented by 1.

**Eviction Policy**  When a set is full the cache victim is always an entry with the minimal counter in the set.

**Admission Policy**  \(d\)W-RAP’s cache policy does not always admit an item into the cache. Instead, it first identifies the set entry with minimal counter as a potential cache victim. If that entry’s ID is \(m\) and its counter value is \(c_m\), a new item is admitted with probability \(\frac{1}{c_m+1}\). The counter of a new item remains with the same value \((c_m)\), and is later incremented by the metadata update.

![Figure 4.2: A 4-way set associative cache with RAP policy.](image)

Figure 4.2: A 4-way set associative cache with RAP policy. Item \(x\) has Set\((x)\)=2. Set 2 includes \(r\), \(w\), \(c\) and \(z\), while \(x\) is not in the cache. The eviction policy selects \(c\), because its frequency is the smallest. \(x\) will be admitted into the cache with probability \(\frac{1}{8}\). If \(x\) is admitted, its counter will be incremented to 8.

An example of \(d\)W-RAP is given in Figure 4.2. When \(x\) arrives, we first look for it in Set\((x)\) = 2. There are 4 items in set 2 and \(x\) is not one of them. Thus we need to decide on eviction and admission. If we choose to admit \(x\), we evict the minimal item \((c)\) in way 2. The frequency of \(c\) is \(c_c = 7\) and therefore \(x\) is only admitted into the cache with probability \(\text{Pr}[\text{Admit}(x)] = \frac{1}{c_c+1} = \frac{1}{8}\). If \(x\) is admitted into the cache, \(c\) is removed from the cache and \(c_x\) is set to be \(c_c + 1 = 8\).

Fortunately, the complexity of implementing \(d\)W-RAP no longer depends on the number of counters, but only on the associativity level \((d)\). The larger \(d\) is, the more combinatoric logic is used for searching the cache and identifying the minimum. As mentioned above, \(d\) is typically very small and we can treat the complexity as \(O(1)\). In Section 4.5, we evaluate \(d\)W-RAP and show that it is almost as accurate as (the fully
associative) RAP, even for relatively small values of $d$. We have also experimented with different associativity levels and evaluated their impact. The experiment details and results appear in Section 4.5.5.

4.5 Evaluation

In this section, we evaluate RAP and $dW$-RAP along with the following previously suggested algorithms – Frequent ($FR$) [29] and Space-Saving ($SS$) [30]. These counter-based algorithms were proven effective for both frequency and top-$k$ estimation. The latter is considered state of the art [37-56]. For frequency estimation, we also compare with sketches such as $CS$ [27] and $CMS$ [28]. We have used a the Mersenne Twister as the core pseudo-random number generator which produces 53-bit precision floats.

For a fair comparison, we evaluate the performance of CS and CMS using 8 times as many counters as the rest of the (counter based) algorithms. To represent their low implementation overhead, they were configured to use 4 lines, which was shown effective in practice [68]. By giving the sketches more counters, we compensate for their lower overheads, as they do not maintain a flow to counter association and avoid storing flow identifiers. We consider this a generous comparison, as the flow id and metadata overheads should not take more than 7 times the counter size.

4.5.1 Datasets

Our evaluation includes the following datasets:

1. The CAIDA Anonymized Internet Trace 2015 [58], or in short, CAIDA. The data is collected from the ‘equinix-chicago’ high-speed monitor and contains 18M elements of mixed UDP, TCP and ICMP packets.

2. The UCLA Computer Science department packet trace (denoted UCLA) [59]. This trace contains 32M UDP packets passed through the border router of the CS Department, University of California, Los Angeles.

3. YouTube Trace from the UMass campus network (referred to as YouTube) [60]. The trace includes a sequence of 600K accesses to YouTube from within the university.

4. Zipf streams. Self-generated traces of identical and independently distributed elements sampled from a Zipf distribution with various skew values ($0.6, 0.8, 1.0, 1.2$ and $1.5$). Hereafter, the skew $X$ stream is denoted Zipf$X$.

4.5.2 Metrics

Our evaluation considers the following performance metrics:
1. **On-Arrival frequency estimation**

Many networking applications take decisions on a *per-packet* basis. For example, if a router identifies excessive traffic originating from a specific source, the router may suspend further routing of its packets to prevent denial of service attacks. We refer to this as the *On-Arrival* model, where upon arrival of each packet, the algorithms are required to estimate its flow frequency. Formally, a stream $S = s_1, s_2, \ldots$ is revealed one element at a time; consequent to $s_t$ arrival, an algorithm $Alg$ is required to provide an estimate $\hat{f}_{s_t}$ for the number elements in the stream with the same id. We then measure the *Mean Square Error* ($MSE$) of the algorithm, i.e.,

$$MSE(Alg) \triangleq \frac{1}{N} \sum_{t=1}^{N} (\hat{f}_{s_t} - f_{s_t})^2.$$ 

2. **Top-$k$ Identification**

The ability to identify the most frequent flows is also important to many applications. We define the *Top-$k$* identification problem as follows: Given a stream $S = s_1, s_2, \ldots$ and two query parameters $m$ and $k$, the algorithm is required to output a set of $m$ elements containing as many of the $k$ most frequent stream elements as possible. We denote the $k$-highest element frequency by $F_k$. For a set of candidates $C$, we measure its quality using the standard recall and precision metrics:

\[
\text{Precision}(C) \triangleq \frac{|e \in C : f_e \geq F_k|}{|C|}
\]

\[
\text{Recall}(C) \triangleq \frac{|e \in C : f_e \geq F_k|}{k}.
\]

4.5.3 **On Arrival Evaluation**

We begin our evaluation with the frequency estimation problem. In this section, each data point was generated by averaging 10 disjoint batches of 1 million packets each, with the exception of YouTube, which is averaged over two 300,000 batches due to the small size of that trace.

Figure 4.3 shows the MSE obtained by the different schemes when equipped with an increasing number of counters. We experimented with different associativity levels to conclude that 16W-RAP behaves almost as good as (the fully associative) RAP. The experiment, appearing in Section 4.5.5, shows that while accuracy does improve as associativity grows, there is only little gain by increasing the associativity beyond 16.

This remains true under all of our tested workloads.

Figure 4.3(a) illustrates our results on the CAIDA packet trace. For the entire range, RAP and 16W-RAP offer significantly lower error than the alternatives. Among the alternatives, none seems to be superior to the rest, as CS, CMS, FR and SS all have some settings where they are more accurate than the rest.

---

Many modern CPUs’ caches have at least 8 ways and 64B lines; thus, each line can accommodate 2 entries, yielding a 16-way structure for our purposes.
Figure 4.3: On Arrival – Mean square error vs. number of counters. Note that CMS and CS have 8 times as many counters!

Figure 4.3(b) describes the results on the UCLA packet trace. In this trace, RAP is the leader for small memory configurations while for 512 counters and onwards SS is slightly better. We believe this is due to the very high skewness of the trace, meaning that the 512 most frequent elements already consist a significant fraction of the stream and therefore our randomization approach is not needed.

The results for the YouTube traces are illustrated in Figure 4.3(c). In this trace, RAP and 16W-RAP are more accurate than the alternatives. Looking only at the alternatives, it is unclear which is the leading among them. However, they all require more than x4 the space to match the accuracy of RAP.

**Synthetic Traces**  Synthetic Zipf traces provide us with better insight on the conditions where RAP works best. The least skewed shown distribution is Zipf 0.6 in Figure 4.3(d).
and the most skewed distribution is Zipf 1.5 in Figure 4.3(i). It appears that RAP performs very well in all these distributions while the alternatives only perform well when the distribution is skewed enough. Figure 4.3(d) shows that for Zipf 0.6, SS with 2048 counters obtains worse MSE than RAP with 32 counters! In Figure 4.3(f) we see that 2048 counter SS is about as accurate as a 128 counters RAP. Figure 4.3(g) exhibits that for Zipf 1, RAP with 256 counters has similar accuracy as SS with 2048 counters. The trend continues until in Figure 4.3(i) the accuracy of RAP with 1024 counters is similar to SS with 2048. For the entire range, RAP requires significantly less space.
4.5.4 Top-k Identification

Since, CS and CMS do not solve the top-$k$ problem, we only compare RAP and 16W-RAP to SS and FR. In some of the figures we also gave RAP half the number of counters as the rest. That configuration is marked as 0.5-RAP.

**Top-32** First, we consider identifying the top-32 flows. We measure the obtained recall for a given number of counters. Our results, summarized in Figure 4.4, demonstrating that 16W-RAP is almost as accurate as RAP in all workloads. Figure 4.4(a) presents results for CAIDA. As shown, RAP and 16W-RAP achieve near perfect recall with 128 counters. In contrast, FR and SS require 1024 counters for the same recall. Results for UCLA are in Figure 4.4(b) As can be seen, RAP and 16W-RAP reach near optimal recall with 128 counters while FR and SS require 256 counters.
Figure 4.6: The Precision-Recall curves for identifying the top-512 with 1024 counters.

Figure 4.4(c) shows that in the YouTube workload, 1024 counters are not enough for SS and FR, and they reach less than 0.5 recall with 1024 counters. In comparison, RAP and 16W-RAP reach near perfect recall with 512 counters.

We now use synthetic Zipf distributions to characterize the performance of RAP and 16W-RAP. Figure 4.4(d) shows that for the mildly skewed Zipf 0.6, both RAP and 16W-RAP achieve near optimal recall with 256 counters, while for the alternatives even 2048 are not enough. In the more skewed Zipf 0.8 distribution, Figure 4.4(f) shows that RAP requires 64 counters, and 16W-RAP requires 128 to achieve near perfect recall. In contrast, SS and FR require 1024 counters to do the same.

For Zipf 1.0 distribution, Figure 4.3(g) shows that RAP and 16W-RAP still require 64 and 128 counters to achieve near optimal recall. FR and SS now require 512 counters to do the same. In Zipf 1.2 distribution, Figure 4.3(h) shows that RAP and 16W-RAP continue to require 64 and 128 counters, while SS and FR now require 256 counters to
achieve near optimal recall. Finally, for the very skewed Zipf 1.5, Figure 4.3(i) shows that RAP and 16W-RAP still require 64 and 128 counters to achieve near optimal recall while SS and FR now require 128 counters. To sum it up, RAP shows a reduction of 2x-16x, depending on workload skewness.

**Convergence Speed** Since RAP is a randomized algorithm, its convergence speed is as important as its performance for large streams. In order to evaluate the convergence speeds of the different algorithms, we consider the problem of identifying the top-512 flows with 1024 counters. Note that the top-512 items also change during the trace and the algorithms are required to constantly adjust. To quantify the improvement of RAP in these settings we use 0.5-RAP, which attempts to identify the top-512 with mere 512 counters. Our results are illustrated in Figure 4.5. As can be seen, RAP and 16W-RAP offer similar recall rates in all the tested workloads.

Figure 4.5(a) exhibits the results for CAIDA workload. As shown, RAP, 16W-RAP and even 0.5-RAP are significantly better than SS and FR. In Figure 4.5(b) we see that in the UCLA workload, RAP and 16W-RAP achieve around 97% recall while SS and FR achieve less than 92%. Interestingly, in this trace, even 0.5-RAP is above 90% recall with just 512 counters.

Figure 4.5(c) shows results for the difficult YouTube trace. Yet, RAP and 16W-RAP achieve above 50% recall, while SS and FR are constantly under 20% recall. The improvement is more than x2, as 0.5 RAP is significantly better than FR and SS.

We now use synthetic workloads to identify the performance envelope of RAP and 16W-RAP. We can observe that as the workload becomes more skewed, all the algorithms improve but RAP and 16W-RAP remain considerably better than the alternatives. For the mildly skewed Zipf 0.6 distribution, Figure 4.5(d) shows that RAP and 16W-RAP achieve over 50% recall while FR and SS are under 10%. In the slightly more skewed Zipf 0.8, Figure 4.5(f) exhibits that RAP and 16W-RAP yield over 90% recall, while the alternatives are slightly below 20%. Similarly, in Zipf 1.0, Figure 4.5(g) shows that RAP and 16W RAP are close to 100% recall during the entire trace while SS and FR are only at 40% recall. In Zipf 1.2, Figure 4.5(h) shows that SS and FR reach 60% while RAP and 16W-RAP remain at 100%. Finally for Zipf 1.5, Figure 4.5(i) shows that SS and FR are approaching 85% recall. Even in this workload, 0.5-RAP eventually achieves higher recall. In all these cases, 0.5-RAP achieves higher recall than SS and FR and the space reduction is at least x2.

**Precision and recall trade-off** While recall is an important measure of the algorithms success, when more than \( k \) items are reported as suspected top-\( k \), the precision is compromised. Returning all the items yields the maximum recall, but also poor precision when many items are monitored. The ideal behavior of a top-\( k \) algorithm is 100% precision and 100% recall (the top right position in the graphs). Figure 4.6 illustrates the precision and
recall trade-off. For each recall level, we measure how many elements were returned to achieve it, and compute the corresponding precision.

Figures 4.6(a) shows our results for the CAIDA workload. RAP and 16W-RAP perform the best on this workload as they can provide 80% recall with near 100% precision, or \( \approx 90\% \) recall with \( \approx 90\% \) precision. At the same time, their maximum recall is close to 100%, but returning all items drops precision to 50%. SS and FR perform worse as their maximum recall is 60% and they can only ensure 30% recall with high precision.

Figure 4.6(b) shows results for the UCLA workload. As can be seen, RAP and 16W-RAP offer the best precision and recall trade-off, although in this case SS and FR also perform well.

Figure 4.6(c) shows results for the YouTube workload. As can be observed, this trace is significantly more difficult. RAP and 16W-RAP perform better than the rest. Their maximal recall is slightly over 60% but over 50% recall is possible with very high precision. SS and FR achieve poor recall and precision.

We now look into what happens in synthetic traces. For Zipf 0.6 distribution, Figure 4.6(d) shows that RAP and 16W-RAP can achieve over 50% recall with good accuracy, while SS and FR achieve less than 10% recall. In the slightly more skewed Zipf 0.8, Figure 4.6(f) shows that RAP and 16W-RAP perform very well. They can offer \( \approx 90\% \) recall and precision. SS and FR improve slightly, both offer bad accuracy but the maximum recall of SS is 30% and FR is slightly less than 20%. The non-monotone rise in the SS curve is explained by coincidentally having higher rates of top-512 elements in the lower estimated frequency counters. For Zipf 1.0, 1.2 and 1.5, Figure 4.6(g), Figure 4.6(h) and Figure 4.6(i) show that while RAP and 16W-RAP provides near optimal precision and recall, SS and FR gradually improve as the skew increases. They achieve \( \approx 50\% \) recall at Zipf 1.0, \( \approx 70\% \) recall at Zipf 1.2 and slightly over 80% recall with high precision with Zipf 1.5. However, in all these cases, 0.5 RAP is better than FR and SS and thus the space reduction is more than x2 across the entire range.

### 4.5.5 Limited Associativity Impact

In this section, we compare the performance of \( d \)-Way RAP for different values of \( d \). We evaluate the associativity levels effect over several metrics which are presented in detail in Section 4.5.3 and Section 4.5.4. These include the following:

1. On-Arrival Mean Square Error, in which every arriving element is queried and we compute the average square error.

2. The percentage (recall) of elements within the top-32 successfully using various space allocations.

3. The recall for identifying the top-512 elements using 1024 counters, compared with the number of observed packets.
Figure 4.7: Comparison of the performance of \( d \)-Way RAP for different associativity levels.

4. The precision-recall curve for identifying the top-512 elements using 1024 counters.

Figure 4.7 shows the performance of the different associativity levels, averaging over 10 batches of 1M packets each from the CAIDA \cite{58} dataset. The results show a diminishing return pattern as associativity is increased; while 1W-RAP performs rather poorly, 2W-RAP is already comparable with the previous algorithms, 4W and 8W offer increased accuracy while 16W-RAP works almost as good as the 32W. Further, our evaluation in Section 4.5.3 and Section 4.5.4 shows that 16W-RAP is roughly comparable to the fully associative RAP. Our experiments suggest that associativity of 16 counters per set is a highly attractive alternative to complete associativity as it does not require any sophisticated data structures (as suggested in \cite{14,30,29}). By not using data structures, we get both simpler implementation, as well as reduction in the memory overhead they require, which can be used for allocating the algorithms with additional counters for increased accuracy.

4.6 Theoretical Guarantees for Top-\( k \) Identification Using Randomized Admission Policy

To show the benefit of probabilistic admission, we describe a variant of RAP (see Section 4.3) that aims to minimize the number of counters needed for top-\( k \) identification. We consider a setting in which the stream elements are i.i.d, i.e., each element is sampled independently and according to the same distribution. Also, we assume that elements are
from a finite domain $\mathcal{U} = \{1, 2, \ldots, D\}$, and without loss of generality, the frequencies of the elements are

$$f_1 \geq f_2 \geq \ldots \geq f_k > f_{k+1} \geq f_{k+2} \geq \ldots \geq f_D.$$  

For $r \in \{1, 2, \ldots, D\}$, we denote $F_r \triangleq \sum_{i=1}^{r} f_i$. This means that at each timestamp, element $i$ will arrive with probability $f_i$ and $\sum_{i=1}^{D} f_i = 1$. We note that this i.i.d. setting may not be applicable to certain streams exhibiting high time locality, such as packets going through a home router, but may resemble the traffic patterns appearing on major backbone routers. The goal of the top-$k$ problem is then to identify the set of the most frequent elements $\{1, 2, \ldots, k\}$ with as few counters as possible. Our assumption is that the stream may be arbitrarily long, but we wish to guarantee that with probability 1 the algorithm will eventually identify all top-$k$ items.

Formally, we say that algorithm $A$ has successfully identified the top-$k$ elements at time $t$ if after the arrival of the $t$’th element, the $k$ largest counters are allocated for items $\{1, 2, \ldots, k\}$. Since the actual items are random variables, we consider the probability, denoted $P_{m,k}^A(t)$, that algorithm $A$ will successfully identify the top-$k$ elements at time $t$ if allocated with $m$ counters. Finally, our benchmark would be the minimal number of counters that $A$ requires to achieve

$$\lim_{t \to \infty} P_{m,k}^A(t) = 1.$$  

We say that an item is a tail item if it is not amongst the top-$k$, i.e., is one of $\{k + 1, \ldots, D\}$. We call the largest $k$ counters the main counters, and the remaining ones tail counters. Our goal is then to ensure that after seeing infinitely many elements, the top-$k$ elements will be guaranteed to be allocated with the main counters. For example, notice that $D$ counters are enough for Space Saving, regardless of the actual frequencies $\{f_i\}$. However, the interesting case is when $m \ll D$.

We start by analyzing the number of counters Space Saving requires for this task. We compare to Space Saving, because it achieves asymptotic improvement over previous work [30] and, to the best of our knowledge, it is considered the state of the art. Notice that our analysis is somewhat different than the one presented in [30], as it further assumes that the stream is i.i.d., which reduces the number of counters required.

### 4.6.1 Conditions for Successful Space Saving Top-$k$ Identification

Assume that we allocate a Space Saving instance with $m$ counters, and would like to identify the top-$k$ elements. In the algorithm, whose pseudo code appears in Algorithm 1, every arriving element is associated with a counter; if a counter was associated with the element prior to its arrival, the counter is incremented by 1; otherwise, the element

\[\text{If we do not wish to assume that } f_k \text{ is strictly larger than } f_{k+1}, \text{ we can replace our demand to finding a set of } k \text{ items such that all of their frequencies are at least } f_k. \text{ Such a model leads to similar results.}\]
“takes over” the minimal counter and increments it. Consequently, we are guaranteed that the top-$k$ element will be associated with a counter upon arrival (unlike our algorithm described below). The only problem for Space Saving arises when a top-$k$ element loses its main counter in favor of a “tail item”, i.e., an item that is not within the top-$k$. Since $k$ is the least-frequent element within the top-$k$, it is enough to consider whether it is guaranteed to have a counter. The key point of our analysis is observing that if all top-$k$ are allocated with counters, they have a positive probability of forever being allocated with these counters if and only if the tail counters increase in a rate smaller than $f_k$. Observe that if the top-$k$ elements reside within the main counters, the expected increment rate of the sum of tail counters is $\sum_{i=k+1}^{D} f_i = 1 - F_k$. This means that the average increment rate for a tail counter is $\frac{1-F_k}{m-k}$, and therefore the increment rate for the minimum among all tail counter is at most $\frac{1-F_k}{m-k}$. On the other hand, the counter allocated for item $k$ increases with a rate of $f_k$. Next, we claim that having $f_k > \frac{1-F_k}{m-k}$ is a necessary and sufficient condition for Space Saving to identify the top-$k$. Illustration of the claim appears in Figure 4.8.

![Figure 4.8: The counters allocated for the top-$k$ elements are increased with rates $f_1, \ldots, f_k$. The increase rate of the minimal tail counter is at most $\frac{1-F_k}{m-k}$.](image)

**Theorem 4.6.1.** Space Saving successfully identifies the top-$k$ using $m$ counters if and only if $f_k > \frac{1-F_k}{m-k}$.

**Proof.** We start by noting that with probability 1, if a top-$k$ element is not allocated with a main counter, it will get such a counter in the future. This happens because we assumed $f_k > f_k+1$, which means that after a tail element takes over a main counter, the counter is incremented with a frequency of at most $f_k+1$, while the counter allocated with the missing top-$k$ element (or the smallest counter if it is not currently allocated with one) increases at a rate of at least $f_k > f_k+1$.

We are left with showing that $f_k > \frac{1-F_k}{m-k}$ is a characterization of the ability of the top-$k$ elements to seize the main counters without being evicted. As mentioned above, it is enough to argue that item $k$ will be eventually allocated within the main counters to ensure all top-$k$ items are successfully identified. Assume that $k$ is allocated with a
counter with value $c_k$, and let $c_t$ be the value of the minimal tail counter. Notice that $c_k$ is increased with rate $f_k$ while $c_t$’s rate is at most $\frac{1-F_k}{m-k}$. Consider the infinite Markov chain whose states represent the difference between $c_k$ and $c_t$, i.e., state $i$ represents the case of $c_k - c_t = i$. At any time an element arrives, it increments $c_k$ with probability $f_k$, increments $c_t$ with probability of at most $1 - \frac{F_k}{m-k}$, and increments other counters otherwise. Therefore if we ignore other counters, the transition probabilities do not depend on the current state $i$ and can be expressed as $\forall i : \Pr[i + 1 \mid i] = \lambda \triangleq \frac{f_k}{1 - \frac{F_k}{m-k} + f_k}$ and $\forall i : \Pr[i - 1 \mid i] = 1 - \Pr[i + 1 \mid i] = \mu \triangleq \frac{1-F_k}{m-k} + f_k$. Notice that 
\[ \Pr[i + 1 \mid i] > \frac{1}{2} \iff f_k > \frac{1 - F_k}{m-k}. \]
The stochastic process is illustrated in Figure 4.9.

Figure 4.9: The probability of moving to a larger index state is $\lambda$ and is not dependent on the process made so far or the current state index.

Since we know that top-$k$ elements will obtain a main counter infinitely often, the question of successful identification narrows to the question “is there a positive probability that given a positive integer $n > 1$, such that if $c_k \geq c_t + n$, the process will never return to state 0?” (as then the main counter allocated for $k$ may become minimal and lead to $k$ being evicted). It is a known fact that a 1D random walk over the non-negative integers that goes left with probability $\mu$ and right with probability $\lambda$ will return to 0 starting at state $n$ is $\left(\frac{\mu}{\lambda}\right)^n$, which is strictly smaller than 1 for $\lambda > \mu$. As we are guaranteed to reach a positive difference infinitely often, when $\mu < 1/2$ we will eventually guarantee that item $k$ will not be evicted if and only if
\[ f_k > \frac{1 - F_k}{m-k}. \] (4.1)
This can further be expressed as a bound on the number of counters required by Space Saving:
\[ m > k + \frac{1 - F_k}{f_k}. \] (4.2)

4.6.2 Conditions for Successful RAP’ Top-$k$ Identification

In this subsection, we present a variant of the Randomized Admission Policy algorithm (see Algorithm 4), called RAP’, that aims to minimize the number of counters required for
top-$k$ identification. RAP’ takes advantage of the fact that we can “slow” the frequency in which the smallest counter is incremented for achieving better identification results. Formally, RAP’ acts similarly to RAP, except that upon arrival of a non-monitored element, the probability in which it will be allocated with the minimal counter is a constant $P$, and do not depend on the value of the minimal counter. Notice that these probabilistic allocations has both positive and negative effects on the possibility of successful identification. On the positive side, since the minimal counter is incremented slower than in Space Saving, we can relax the $f_k > \frac{1-F_k}{m}$ constraint a bit. Alas, RAP’ is not guaranteed that a top-$k$ element will obtain a counter infinitely often. This means that for RAP’ to successfully identify the top-$k$ we need to impose two constraints:

1. A top-$k$ item which is not currently allocated with a counter will get one (w.p. 1). Notice that if a top-$k$ element is not allocated with a counter (recall that now we have $m-1$ allocated counters), then the minimal, non-allocated counter increases with rate of at least $P \cdot (f_k + 1 - F_m)$. In contrast, the slowest allocated counter rate cannot exceed $f_m$. Thus, the imposed constraint is

$$f_m < P \cdot (f_k + 1 - F_m).$$

This gives us a lower bound on the value of $P$:

$$P > \frac{f_m}{f_k + 1 - F_m}. \quad (4.3)$$

For the sake of simplifying the calculations, we impose a stronger restriction on the admission probability and consider $P$ to satisfy

$$P > \frac{f_m}{f_k}. \quad (4.4)$$

2. When all top-$k$ elements are allocated with counters, there exists a positive probability that they will never be evicted from the table. This is the front in which RAP’ has advantage over Space Saving. Since the minimal counter is only incremented with sampling rate $P$, the rate in which the smallest counter within the tail increases is at most

$$\frac{\sum_{i=k+1}^{m} f_i + P \cdot \sum_{i=m+1}^{D} f_i}{m - k} = \frac{F_m - F_k + P \cdot (1 - F_m)}{m - k}.$$

Meanwhile, the counter associated with item $k$ is incremented in rate of $f_k$. This means that our constraint is:

$$f_k > \frac{F_m - F_k + P \cdot (1 - F_m)}{m - k}. \quad (4.6)$$

---

3If there exists more than a single minimum-value counter, then the arriving element is admitted with probability 1.
Notice that for $P = 1$, Inequality (4.5) trivially holds (and thus Space Saving is guaranteed to get counters allocated for top-$k$ elements infinitely often), while Inequality (4.6) degenerates into Inequality (4.1). In the following subsection, we analyze how a smart choice of the increment probability $P$ reduces the required number of counters.

4.6.3 Performance Comparison for Zipf Distributed Streams

In many of the previous works, Zipf distributed streams served as a popular benchmark for algorithms comparison due to its nice mathematical properties. Here, we continue this line and compare the performance of Space Saving and RAP’ on i.i.d Zipf distributed streams with varying skews. We start with a formal definition of a Zipf stream.

**Definition** Denote $\Gamma_\alpha(D) = \sum_{i=1}^{D} i^{-\alpha}$. A stream will be called an i.i.d Zipf stream with skew $\alpha$ over domain $D$ if all of its elements are sampled independently and follow the distribution in which item $i \in \{1, 2, \ldots, D\}$ appears with probability $f_i = \frac{i^{-\alpha}}{\Gamma_\alpha(D)}$.

Traditionally, streams with skew $\alpha \in (0, 1]$ are called “mildly skewed” or “heavy tailed”, while larger skews grants the streams the title “highly skewed”. Streams in which every item is selected with uniform probability (skew=0) are called “uniform”. These names differentiate highly skewed streams, in which a small number of elements consists most of the stream, and heavy tailed ones, where most of the arriving elements are tail items. This property is also observable from the behavior of $\Gamma_\alpha(D)$; for $\alpha > 1$, $\Gamma_\alpha(D)$ convergence to a constant as $D$ grows (e.g., $\Gamma_2(\infty) \approx 1.645$); for $\alpha = 1$, $\Gamma_1(D) \approx \ln(1.78D)$; lastly, for $\alpha < 1$, we have $\Gamma_\alpha(D) = \frac{D^{1-\alpha}}{1-\alpha} + O(1)$.

For heavily skewed streams, Space Saving is known to be optimal [30]. For more mildly skewed streams, we show that RAP’ could asymptotically improve the number of counters required for identifying the top-$k$ elements. This also provides theoretical grounds to the poor empirical performance of Space Saving when evaluated on heavy tailed workloads in Section 4.5.

Assuming a Zipf distributed stream, the condition for Space Saving to converge, as appears in (4.2), then becomes:

$$m > k + \frac{1 - F_k}{f_k} = \frac{1}{\Gamma_\alpha(k)} - \frac{1}{\Gamma_\alpha(D)}$$

$$\implies m > k + k^\alpha \left( \Gamma_\alpha(D) - \Gamma_\alpha(k) \right) \quad (4.7)$$

For analyzing RAP’s performance, we will select the value of $P$ based on the skewness of the stream, as discussed below. When plugging the Zipf distribution into the first RAP’ constraint (see (4.5)), we get

$$P > \frac{f_m}{f_k} = \left( \frac{m}{k} \right)^{-\alpha}$$

$$\implies m > k \cdot P^{-\frac{1}{\alpha}}. \quad (4.8)$$
Similarly, the second constraint (see (4.6)) is now:

\[ f_k > \frac{F_m - F_k + P \cdot (1 - F_m)}{m - k} \]

\[ \iff m > k + k^\alpha \left( \Gamma_\alpha(m) - \Gamma_\alpha(k) 
+ P \cdot (\Gamma_\alpha(D) - \Gamma_\alpha(m)) \right) \]

In order to simplify the right hand side of the inequality, we impose a stronger bound on \( m \) and require it to satisfy:

\[ m > k + k^\alpha \cdot (\Gamma_\alpha(m) - \Gamma_\alpha(k) + P \cdot \Gamma_\alpha(D)) \]  

(4.9)

**Heavy Tailed Streams**

In this section, we assume that \( \alpha \in (0,1) \) is fixed and that \( k = o(D^{\frac{1}{1+\alpha}}) \) and analyze the number of counters required for Space Saving and RAP’ for successfully identifying the top-\( k \) items. We start by using the explicit formula of \( \Gamma_\alpha(\cdot) \) for (4.7):

\[ m_{SS} = k + k^\alpha (\Gamma_\alpha(D) - \Gamma_\alpha(k)) \]

\[ = k + k^\alpha \frac{D^{1-\alpha} - k^{1-\alpha} + \Theta(1)}{1 - \alpha} \]

\[ = \frac{k^\alpha \cdot D^{1-\alpha} - \alpha \cdot k + \Theta(k^\alpha)}{1 - \alpha} = \Theta(D^{1-\alpha}) \]

Thus, we established that the number of counters required for Space Saving is \( m_{SS} = \Omega(D^{1-\alpha}) \).

For RAP’, we choose the admission probability to be

\[ P \triangleq D^{\frac{\alpha^2}{1+\alpha}}. \]  

(4.10)

Notice that \( P \in (0,1) \) is a valid probability. Next, we will show that using

\[ m_{RAP'} \triangleq c \cdot k \cdot D^{\frac{1-\alpha}{1+\alpha}} \]  

(4.11)

counters, where \( c \) is a (large enough) constant, we can satisfy both (4.8) and (4.9), thus \( m_{RAP'} \) counters are enough for successful identification of the top-\( k \) elements.

Constraint (4.8) requires that \( m > k \cdot P^{-\frac{1}{\alpha}} \). Plugging in (4.10) and (4.11), we get:

\[ m_{RAP'} = c k \cdot D^{\frac{1-\alpha}{1+\alpha}} > k \cdot P^{-\frac{1}{\alpha}}, \]

as required. Next, we show an inequality that will be useful later:

\[ (m_{RAP'})^{1-\alpha} = (ck \cdot D^{\frac{1-\alpha}{1+\alpha}})^{1-\alpha} \]

\[ = \left(D^{\frac{1}{1+\alpha}} \cdot \frac{ck}{D^{\frac{1-\alpha}{1+\alpha}}} \right)^{1-\alpha} < D^{\frac{1-\alpha}{1+\alpha}}, \]  

(4.12)

where the last inequality holds for large enough \( c \).

---

\(^4\)In practice, values of \( k \) are typically very small and may be considered sub-polynomial in \( D \).
Finally, we show that our choice of $P$ and $m_{\text{RAP}'}$ also satisfies (4.6):

$$k + k^\alpha \cdot (\Gamma_\alpha(m) - \Gamma_\alpha(k) + P \cdot \Gamma_\alpha(D))$$

$$= k + \frac{k^\alpha \cdot \left(m^{1-\alpha} - k^{1-\alpha} + D \frac{\alpha^2 - \alpha}{1+\alpha} \cdot D^{1-\alpha} + \Theta(1)\right)}{1 - \alpha}$$

$$= \frac{k^\alpha \cdot \left(m^{1-\alpha} + D \frac{1}{1+\alpha}\right) - \alpha k + \Theta(k^\alpha)}{1 - \alpha}$$

$$< \frac{2k^\alpha \cdot D \frac{1}{1+\alpha} - \alpha k + \Theta(k^\alpha)}{1 - \alpha}$$

$$= m_{\text{RAP}'} \cdot \frac{2}{c(1 - \alpha)} + O(k) < m_{\text{RAP}'}$$

(4.13)

Since we have shown that our selection of admission probability and number of counters satisfies both constraints, we have proved that RAP’ requires only $O(D^{1-\alpha})$ counters to successfully identify the top-$k$ hitters on a Zipf stream with skew $\alpha$.

We conclude that for the problem of identifying top-$k$ over i.i.d. heavy tailed streams, Space Saving requires $\Theta(D^{-\alpha})$ while RAP’ requires $\Theta(D^{1-\alpha})$. Notice that for values of $\alpha$ that are close to 1, this is nearly a quadratic space reduction. For example, consider trying to find the top-32 flows on a backbone router whose traffic is approximately Zipf0.8 with domain of $D = 2^{64}$ elements. Space Saving requires about 570K counters; in contrast, RAP’ could allocate roughly 44K counters to achieve the same. The admission probability for these input parameters is slightly less than 2%.

**Skew=1 Streams**

Heavy tailed streams are usually not analyzed in the literature, perhaps because the existing algorithm cannot find the top-$k$ elements in these using a reasonable amount of counters (see Section 4.5). However, skew 1 Zipf streams were analyzed in some previous works for both Space Saving and Count Sketch [27, 30]. In this section, we show that by introducing an admission probability, RAP’ is able to achieve asymptotic space improvement on i.i.d. Zipf streams. The convergence condition for Space Saving, which appears in (4.7) is now:

$$m > k + k (\Gamma_1(D) - \Gamma_1(k))$$

$$\approx k + k \ln \left(1.78 \frac{D}{k}\right) = \Theta(\log D).$$

(4.14)

For RAP’, we choose the admission probability to be

$$P \triangleq \sqrt{\frac{1}{\ln D}}.$$ 

(4.15)

We show that using probabilistic admission, we reduce the number of required counters to $m_{\text{RAP}'} \triangleq c \cdot k \sqrt{\ln D} = O(\sqrt{\log D})$, where $c$ is a positive constant. We start by showing
that this choice of $m_{RAP'}$ and $P$ satisfies (4.8):

$$m_{RAP'} = c \cdot k \sqrt{\ln D} > k \cdot P^{-1}$$ (4.16)

Next, we consider (4.9):

$$k + k \cdot (\Gamma_1(m) - \Gamma_1(k) + P \cdot \Gamma_1(D))$$

$$\approx k + k \cdot \left( \ln \left( 1.78 \frac{m}{k} \right) + \sqrt{\frac{1}{\ln D}} \ln (1.78D) \right)$$

$$= k + k \cdot \left( \ln \left( c \cdot \sqrt{\ln D} \right) + \sqrt{\ln D + O(1)} \right) < m_{RAP'},$$

where the last inequality holds for large enough $c$.

We conclude that by introducing a probabilistic admission filter, one can reduce the number of counters required for successful top-$k$ identification over skew=1 Zipf streams from $O(\log D)$ to $O(\sqrt{\log D})$.

### 4.6.4 Section Summary

In this section, we have shown how the concept of admission filters, introduced in Section 4.3 can be adapted for reducing the number of counters needed for top-$k$ identification. Nevertheless, our analysis has a few drawbacks; first, we are only able to provide theoretical analysis for i.i.d. streams, which may not truly represent the all practical settings; second, our analysis assumes that the stream may be arbitrarily long, and only considers eventual convergence. While this may fit massive data streams, in cases where we wish to process smaller streams, perhaps because we reset the process every once in a while to provide freshness, our analysis does not hold. Lastly, we have assumed a prior knowledge of the data skew. In practice, one can estimate the skew from the data, but this will require the admission probability to be adaptive.

We plan to evaluate RAP’ on real data traces and compare it with existing techniques, and specifically with RAP (see Algorithm 4) whose admission probability was optimized for frequency estimation but proved effective also for top-$k$. We also wish to find a method for dynamically adapting the admission probability without assuming knowledge about the stream length or skew.

### 4.7 Conclusion and Discussion

In this chapter, we have presented Randomized Admission Policy (RAP), a novel algorithm for approximate frequency estimation and top-$k$ identification. We have also introduced $d$-Way Randomized Admission Policy ($dW$-RAP), a hardware friendly variant of RAP. We have extensively evaluated RAP and $dW$-RAP for both problems under two packet traces and a YouTube video trace as well as multiple synthetic Zipf traces. These
experiments exhibited significant reductions in the memory requirements of RAP and dW-RAP compared to state of the art alternatives for obtaining the same error. In top-k, we showed that our algorithms achieve superior precision/recall than the alternatives in any tested situation. In the case of frequency estimation, the only exception is the highly skewed UCLA trace [59], and even there, it is only when all schemes are allocated a very large number of counters compared to the trace. Notice that for this case, all algorithms are precise since with many counters on such a skewed trace, the problem becomes almost trivial. In contrast, RAP and dW-RAP are the only algorithms that performed well on heavy-tailed distributions, which are common in Internet services.

Another benefit of RAP and dW-RAP is that they incur fewer updates to memory since they do not replace a counter with each untracked item. This is especially true in heavy-tailed workloads. We have not included the evaluation and quantification of this property for lack of space.

Since dW-RAP can be implemented as a simple cache policy, in the future we would like to integrate it into real networking devices. Interestingly, we believe that dW-RAP may also offer benefits for software implementations. For example, it can probably be parallelized efficiently since each operation only computes the minimum over a small counter set.
Chapter 5

Fast Flow Volume Estimation

5.1 Introduction

The increasing popularity of jumbo frames means growing variance in the size of packets transmitted in modern networks. Consequently, network monitoring tools must maintain explicit traffic volume statistics rather than settle for packet counting as before. We present constant time algorithms for volume estimations in streams and sliding windows, which are faster than previous work. Our solutions are formally analyzed and are extensively evaluated over multiple real-world packet traces as well as synthetic ones. For streams, we demonstrate a run-time improvement of up to 2.4X compared to the state of the art. On sliding windows, we exhibit a memory reduction of over 100X on all traces and an asymptotic runtime improvement to a constant. Finally, we apply our approach to hierarchical heavy hitters and achieve an empirical 2.4-7X speedup.

5.1.1 Background

Traffic measurement is vital for many network algorithms such as routing, load balancing, quality of service, caching and anomaly/intrusion detection [47, 48, 49, 50, 51, 52]. Typically, networking devices handle millions of concurrent flows [53, 54, 55]. Often, monitoring applications track the most frequently appearing flows, known as heavy hitters, as their impact is most significant.

Most works on heavy hitters identification have focused on packet counting [14, 22, 26, 69]. However, in recent years jumbo frames and large TCP packets are becoming increasingly popular and so the variability in packet sizes grows. Consequently, plain packet counting may no longer serve as a good approximation for bandwidth utilization. For example, in data collected by [70] in 2014, less than 1% of the packets account for over 25% of the total traffic. Here, packet count based heavy hitters algorithms might fail to identify some heavy hitter flows in terms of bandwidth consumption.

Hence, in this chapter we explicitly address monitoring of flow volume rather than plain packet counting. Further, given the rapid line rates and the high volume of accumulating
data, an aging mechanism such as a sliding window is essential for ensuring data freshness and the volume estimation’s relevance. Hence, we study estimations of flow volumes in both streams and sliding windows.

Finally, per flow measurements are not enough for certain functionalities like anomaly detection and Distributed Denial of Service (DDoS) attack detection \[71, 72\]. In such attacks, each attacking device only generates a small portion of the traffic and is not a heavy hitter. Yet, their combined traffic volume is overwhelming. Hierarchical heavy hitters (HHH) aggregates traffic from IP addresses that share some common prefix \[38\]. In a DDoS, when attacking devices share common IP prefixes, HHH can discover the attack. To that end, we consider volume based HHH detection as well.

Before explaining our contribution, let us first motivate why packet counting solutions are not easily adaptable to volume estimation. Counter algorithms typically maintain a fixed set of counters \[30, 35, 33, 73, 29, 14\] that is considerably smaller than the number of flows. Ideally, counters are allocated to the heavy hitters. When a packet from an unmonitored flow arrives, the corresponding flow is allocated the minimal counter \[30\] or a counter whose value has dropped below a dynamic threshold \[35\].

We refer to a stream in which each packet is associated with a weight as a weighted stream. Similarly, we refer to streams without weights, or when all packets receive the same weight, as unweighted. For unweighted streams, ordered data structures allow constant time updates and queries \[30, 14\], since when a counter is incremented, its relative order among all counters changes by at most one. Unfortunately, maintaining the counters sorted after a counter increment in a weighted stream either requires to search for its new location, which incurs a logarithmic cost, or resorting to logarithmic time data structures like heaps. The reason is that if the counter is incremented by some value \(w\), its relative position might change by up to \(w\) positions. This difficulty motivates our work\[\footnote{The most naive approach treats a packet of size \(w\) as \(w\) consecutive arrivals of the same packet in the unweighted case, resulting in linear update times, which is even worse.}\].

5.1.2 Contributions

We contribute to the following network traffic measurement problems: (i) stream heavy hitters, (ii) sliding window heavy hitters, (iii) stream hierarchical heavy hitters. Specifically, our first contribution is Frequent items Algorithm with a Semi-structured Table (FAST), a novel algorithm for monitoring flow volumes and finding heavy hitters. FAST processes elements in worst case \(O(1)\) time using asymptotically optimal space. A major part of our contribution lies in the detailed formal analysis we perform, which proves the above properties, as well as in and the accompanying performance study. We evaluate FAST on five real Internet packet traces from a data center and backbone networks. We demonstrate a speedup of up to a factor of 2.4X compared to previous works.
Our second contribution is **Windowed Frequent items Algorithm with a Semi-structured Table (WFAST)**, a novel algorithm for monitoring flow volumes and finding heavy hitters in sliding windows. We evaluate WFAST on five Internet traces and show that its runtime is reasonably fast, and that it requires as little as 1% of the memory of previous work [71]. We analyze WFAST and show that it operates in constant time and is space optimal, which asymptotically improves both the runtime and the space consumption of previous work. We believe that such a dramatic improvement makes volume estimation over a sliding window practical!

Our third contribution is **Hierarchical Frequent items Algorithm with a Semi-structured Table (HFAST)**, which finds hierarchical heavy hitters. HFAST is created by replacing the underlying HH algorithm in [38] (Space Saving) with FAST. We evaluate HFAST and demonstrate an asymptotic update time improvement as well as an empirical 2.4-7X speedup on real Internet traces.

### 5.2 Related Work

As mentioned above, our work addresses three related problems, which we survey below.

#### 5.2.1 Streams

**Probabilistic short counters**, or **estimators**, represent large numbers using small counters by degrading precision [26] [22] [74]. By shrinking counters’ size, more flows can be monitored in SRAM. But these methods still require maintaining a flow-to-counter mapping that often requires more space than the counters themselves. Sampling is also an attractive approach when space is scarce [65, 75, 66] despite the resulting sampling error.

Sketches such as **Count Sketch (CS)** [27] and **Count Min Sketch (CMS)** [28] are attractive as they enable counter sharing and need not maintain a flow to counter mapping for all flows. Sketches typically only provide a probabilistic estimation, and often do not store flow identifiers. Thus, they cannot find the heavy hitters, but only focus on the volume estimation problem. Advanced sketches, such as **Counter Braids** [55], **Randomized Counter Sharing** [62] and **Counter Tree** [61], improve accuracy but their queries require complicated decoding procedures that can only be done off-line.

In **counter based** algorithms, a flow table is maintained, but only a small number of flows are monitored. These algorithms differ from each other in the size and maintenance policy of the flow table, e.g., **Lossy Counting** [35] and its extensions [33] [73], **Frequent** [29] and **Space Saving** [30]. Given ideal conditions, counter algorithms are considered superior to sketch based techniques. Particularly, Space Saving was empirically shown to be the most accurate [37] [56] [57]. Many counter based algorithms were developed by the databases community and are mostly suitable for software implementation. In [14] we suggested a compact static memory implementation of Space Saving that may be more
accessible for hardware design. Yet, software implementations are becoming increasingly relevant in networking as emerging technologies such as NFVs become popular.

Alas, most previous works rely on sorted data structures such as Stream Summary [30] or SAIL [14] that only operate in constant time for unweighted updates. As mentioned, existing sorted data structures cannot be maintained in constant time in the weighted updates case. Thus, a logarithmic time heap based implementation of Space Saving was suggested [56] for the more general volume counting problem. IM-SUM, DIM-SUM [14] and BUS-SS [76] are very recent algorithms developed for the volume heavy-hitters problem (only for streams, with no sliding windows support). BUS offers a randomized algorithm that operates in constant time. IM-SUM operates in amortized $O(1)$ time and DIM-SUM in worst case constant time. Empirically, DIM-SUM is slower than FAST. Additionally, DIM-SUM requires $2^{\frac{\gamma}{\epsilon}}$ counters, for some $\gamma > 0$, to guarantee $N \cdot M \cdot \epsilon$ error and operates in $O(\gamma^{-1})$ time. FAST only needs half as many counters for the same time and error guarantees. The very recent work of [77], introduces a mergeable algorithm that operates in amortized constant time.

5.2.2 Sliding Windows

Heavy hitters on sliding windows were first studied by [12]. Given an accuracy parameter ($\epsilon$), a window size ($W$) and a maximal increment size ($M$), such algorithms estimate flows’ volume on the sliding window with an additive error that is at most $W \cdot M \cdot \epsilon$.

Their algorithm requires $O\left(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon}\right)$ counters and performs queries and updates in $O\left(\frac{1}{\epsilon} \log \frac{1}{\epsilon}\right)$ time. The work of [6] reduces the space requirements and update time to $O\left(\frac{1}{\epsilon}\right)$. An improved algorithm with a constant update time is given in [78]. Further, [14] provided an algorithm that requires $O\left(\frac{1}{\epsilon}\right)$ for queries and supports constant time updates and item frequency queries.

The weighted variant of the problem was only studied by [79], whose algorithm operates in $O\left(\frac{1}{\epsilon}\right)$ time and requires $O\left(\frac{1}{\epsilon}\right)$ space for a $W \cdot M \cdot \epsilon$ approximation; here, $A \in [1, M]$ is the average packet size in the window. In this work, we suggest an algorithm for the weighted problem that (i) uses optimal $O\left(\frac{1}{\epsilon}\right)$ space, (ii) performs heavy hitters queries in optimal $O\left(\frac{1}{\epsilon}\right)$ time, and (iii) performs volume queries and updates in constant time.

5.2.3 Hierarchical Heavy Hitters

Hierarchical Heavy Hitters (HHH) were first defined by [80], and then extended to multiple dimensions in [81, 40, 52, 71, 38]. HHH algorithms monitor aggregates of flows that share a common prefix. To do so, HHH algorithms treat flows identifiers as a hierarchical domain. We denote by $H$ the size of this domain.

A single dimension algorithm requiring $O\left(\frac{H^2}{\epsilon}\right)$ space was introduced in [83]. Later, [84] showed a two dimensions algorithm requiring $O\left(\frac{H^{3/2}}{\epsilon}\right)$ space and update time. The full
and partial ancestry algorithms \[40\] are trie based algorithms that require \(O \left( \frac{H}{\epsilon} \log \epsilon N \right)\) space and operate at \(O \left( H \log \epsilon N \right)\) time. The state of the art \[38\] algorithm requires \(O \left( \frac{H}{\epsilon} \right)\) space and its update time for weighted inputs is \(O \left( H \log \left( \frac{1}{\epsilon} \right) \right)\).

The algorithm of \[38\] solves the approximate HHH problem by dividing it into multiple simpler heavy hitters problems. In our work, we replace the underlying heavy hitters algorithm of \[38\] with FAST, which yields a space complexity of \(O \left( \frac{H}{\epsilon} \right)\) and an update complexity of \(O(H)\). That is, we improve the update complexity from \(O \left( H \log \left( \frac{1}{\epsilon} \right) \right)\) to \(O(H)\). Alternatively, the recent work of \[35\] suggests a novel HHH algorithm that takes linear space but optimizes the update time.

5.3 Preliminaries

Given a set \(\mathcal{U}\) and a positive integer \(M \in \mathbb{N}^+\), we say that \(\mathcal{S}\) is a \((\mathcal{U}, M)\)-weighted stream if it contains a sequence of \((id, weight)\) pairs. Specifically: \(\mathcal{S} = (p_1, \ldots, p_N)\), where \(\forall i \in 1, \ldots, N\): \(p_i \in \mathcal{U} \times \{1, \ldots, M\}\). Given a packet \(p_i = (d_i, w_i)\), we say that \(d_i\) is \(p_i\)'s id while \(w_i\) is its weight; \(N\) is the stream length, and \(M\) is the maximal packet size. Notice that the same packet id may possibly appear multiple times in the stream, and each such occurrence may potentially be associated with a different weight. Given a \((\mathcal{U}, M)\)-weighted stream \(\mathcal{S}\), we denote \(v_x\), the volume of id \(x\), as the total weight of all packets with id \(x\). That is: \(v_x \triangleq \sum_{i \in \{1, \ldots, N\}}: w_i\). For a window size \(W \in \mathbb{N}^+\), we denote the window volume of id \(x\) as its total weight of packets with id \(x\) within the last \(W\) packets, that is: \(v^W_x \triangleq \sum_{i \in \{N-W+1, \ldots, N\}}: w_i\). We seek algorithms that support the operations:

- **ADD**\((\langle x, w \rangle)\): append a packet with identifier \(x\) and weight \(w\) to \(\mathcal{S}\).
- **Query**\((x)\): return an estimate \(\widehat{v}_x\) of \(v_x\).
- **WinQuery**\((x)\): return an estimate \(\widehat{v}^W_x\) of \(v^W_x\).

We now formally define the main problems in this work:

- **\((\epsilon, M)\)-Volume Estimation**: \(\text{QUERY}()\) returns an estimation \((\widehat{v}_x)\) that satisfies \(v_x \leq \widehat{v}_x \leq v_x + N \cdot M \cdot \epsilon\).

- **\((W, \epsilon, M)\)-Volume Estimation**: \(\text{WINQUERY}(x)\) returns an estimation \((\widehat{v}^W_x)\) that satisfies \(v^W_x \leq \widehat{v}^W_x \leq v^W_x + W \cdot M \cdot \epsilon\).

- **\((\theta, \epsilon, M)\)-Approximate Weighted Heavy Hitters**: returns a set \(H \subseteq \mathcal{U}\) such that: \(\forall x \in \mathcal{U} \colon (v_x > N \cdot M \cdot \theta \Rightarrow x \in H) \land (v_x < N \cdot M \cdot (\theta - \epsilon) \Rightarrow x \notin H)\).

- **\((W, \theta, \epsilon, M)\)-Approximate Weighted Heavy Hitters**: returns a set \(H \subseteq \mathcal{U}\) such that: \(\forall x \in \mathcal{U} \colon (v^W_x > W \cdot M \cdot \theta \Rightarrow x \in H) \land (v^W_x < W \cdot M \cdot (\theta - \epsilon) \Rightarrow x \notin H)\).

Our heavy hitter definitions are asymmetric. That is, they require that flows whose frequency is above the threshold of \(N \cdot M \cdot \theta\) (or \(W \cdot M \cdot \theta\)) are included in the list, but flows whose volume is slightly less than the threshold can be either included or excluded from the list. This relaxation is necessary as it enables reducing the required amount of space to sub linear. Let us emphasize that the identities of the heavy hitter flows are not
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>stream</td>
</tr>
<tr>
<td>$N$</td>
<td>number of elements in the stream</td>
</tr>
<tr>
<td>$M$</td>
<td>maximal value of an element in the stream</td>
</tr>
<tr>
<td>$W$</td>
<td>window size</td>
</tr>
<tr>
<td>$\mathcal{U}$</td>
<td>the universe of elements</td>
</tr>
<tr>
<td>$[r]$</td>
<td>the set ${0, 1, \ldots, r - 1}$</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>FAST performance parameter.</td>
</tr>
<tr>
<td>$v_x$</td>
<td>the volume of an element $x$ in $S$</td>
</tr>
<tr>
<td>$\hat{v}_x$</td>
<td>an estimation of $v_x$</td>
</tr>
<tr>
<td>$v^W_x$</td>
<td>the volume of element $x$ in the last $W$ elements of $S$</td>
</tr>
<tr>
<td>$\hat{v}^W_x$</td>
<td>an estimation of $v^W_x$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>estimation accuracy parameter</td>
</tr>
<tr>
<td>$\theta$</td>
<td>heavy hitters threshold parameter</td>
</tr>
</tbody>
</table>

Table 5.1: List of Symbols

known in advance. Hence, it is impossible to a-priori allocate counters only to these flows. The basic notations used in this work are listed in Table 5.1.

5.4 Frequent items Algorithm with a Semi-structured Table (FAST)

In this section, we present Frequent items Algorithm with a Semi-structured Table (FAST), a novel algorithm that achieves constant time weighted updates. FAST uses a data structure called Semi Ordered Summary (SOS), which maintains flow entries in a semi ordered manner. That is, similarly to previous works, SOS groups flows according to their volume, each of which is called a volume group. The volume groups are maintained in an ordered list. Each volume group is associated with a value $C$ that determines the volume of its nodes. Unlike existing data structures, counters within each volume group are kept unordered.

Unlike previous works, the grouping is done at coarse granularity. Each node (inside a group) includes a variable called Remainder (denoted $R$). The volume estimate of a flow is $C + R$ where $R$ is the remainder of its volume node and $C$ is the value of its volume group.

This semi-ordered structure is unique to SOS and enables it to serve weighted updates in $O(1)$. Volume queries are satisfied in $O(1)$ time using a separate aggregate hash table which maps each flow identifier to its SOS node. FAST then uses SOS to find a near-minimum flow when needed.
Figure 5.1: An example of how FAST utilizes the SOS structure. Here, flows are partially ordered according to the third digit (100’s), and each flow maintains its own remainder; e.g., the estimated volume of $D$ is $\hat{v}_D = 583$.

Figure 5.1 provides an intuitive example for the case $M = 1,000$. Here, the volume of an item is calculated by both its group counter ($C$) and the item’s remainder ($R$), e.g., the volume of $A$ is $400 + 32 = 432$. Flows are partially ordered according to their third digit, i.e., in multiples of 100, or $M/10$. Within a specific group, however, items are unordered, e.g., $A$, $B$ and $J$ are unordered but all appear before items with volume of at least 500. As the number of lists to skip prior to an addition is $O(1)$, the update complexity is also $O(1)$. Specifically, we need to traverse at most 10 linked lists when updating an item.

Intuitively, flows are only ordered according to volume groups and if we make sure that the maximal weight can only advance a flow a constant number of flow groups then SOS operates in constant time. Alas, keeping the flows only partially ordered increases the error. We compensate for such an increase by requiring a larger number of SOS entries compared to previously suggested fully ordered structures. The main challenge in realizing this idea is to analyze the accuracy impact and provide strong estimation guarantees.

5.4.1 FAST - Accurate Description

FAST employs $\left\lceil \frac{1+\gamma}{\epsilon} \right\rceil$ counters, for some non-negative constant $\gamma \geq 0$. $\gamma$ determines how ordered SOS is: for $\gamma = 0$, we get full order, while for $\gamma > 0$, it is only ordered up to $M \cdot \gamma/2$ (all flows that fall into the same volume group are unordered, and each group holds a range of $M \cdot \gamma/2$ values). The runtime is, however, $O(1/\gamma)$ and is therefore constant for any fixed $\gamma$. We note that an $\Omega(\epsilon^{-1})$ counters lower bound is known \cite{30}. Thus, FAST is asymptotically optimal for constant $\gamma$. The pseudo code of FAST appears in Algorithm 5.

5.4.2 FAST Analysis

We start by a simple useful observation

**Observation 5.4.1.** Let $a, b \in \mathbb{N}$ : $a = b \cdot \left\lfloor \frac{a}{b} \right\rfloor + (a \mod b)$.
Algorithm 5 FAST \((M, \epsilon, \gamma)\)

\[\begin{align*}
\text{Initialization:} & \quad C \leftarrow \emptyset, \forall x : c_x \leftarrow 0, r_x \leftarrow 0, s \leftarrow \left\lfloor \frac{M \gamma}{0.5} + 1 \right\rfloor, \mathcal{C} \leftarrow \left\lfloor \frac{1 + \gamma}{\epsilon} \right\rfloor.
1: & \quad \text{function } \text{Add}(\text{Item } x, \text{Weight } w) \\
2: & \quad \text{if } x \in C \text{ or } |C| < \mathcal{C} \text{ then}
3: & \quad c_x \leftarrow c_x + \left\lfloor \frac{r_x + w}{s} \right\rfloor
4: & \quad r_x \leftarrow (r_x + w) \mod s
5: & \quad C \leftarrow C \cup \{x\}
6: & \quad \text{else}
7: & \quad \text{Let } m \in \arg\min_{y \in C} (c_y) \quad > \text{arbitrary minimal item}
8: & \quad c_x \leftarrow c_m + \left\lfloor \frac{r_x + 1 + w}{s} \right\rfloor
9: & \quad r_x \leftarrow (s - 1 + w) \mod s
10: & \quad C \leftarrow C \setminus \{m\} \cup \{x\}
11: & \quad \text{end if}
12: & \quad \text{end function}
13: & \quad \text{function } \text{Query}(x) \\
14: & \quad \text{if } x \in C \text{ or } |C| < \mathcal{C} \text{ then}
15: & \quad \text{return } r_x + s \cdot c_x
16: & \quad \text{else}
17: & \quad \text{return } s - 1 + s \cdot \min_{y \in C} c_y
18: & \quad \text{end if}
19: & \quad \text{end function}
\end{align*}\]

For the analysis, we use the following notations: for every item \(x \in \mathcal{U}\) and stream length \(t\), we denote by \(q_t(x)\) the value of QUERY() after seeing \(t\) elements. We slightly abuse the notation and refer to \(t\) also as the \(t\) time at which the \(t\) element arrived, where time here is discrete. We denote by \(C_t\) the set of elements with an allocated counter at time \(t\), by \(r_{x,t}\) the value of \(r_x\) and by \(c_{x,t}\) the value of \(c_x\). Also, we denote the volume at time \(t\) as \(v_{x,t} \triangleq \sum_{i \in \{1,\ldots,t\}} d_{i,x} = r_{x,t} + s \cdot c_{x,t}\).

We now show that FAST has a one-sided error.

**Lemma 5.4.2.** For any \(t \in \mathbb{N}\), after seeing any \((\mathcal{U}, M)\)-weighted stream \(S\) of length \(t\), for any \(x \in \mathcal{U} : v_x \leq \tilde{v}_x\).

**Proof.** We prove \(v_{x,t} \leq q_t(x)\) by induction over \(t\).

**Basis:** \(t = 0\). Here, we have \(v_{x,t} = 0 = q_t(x)\).

**Hypothesis:** \(v_{x,t-1} \leq q_{t-1}(x)\).

**Step:** \((x_t, w_t)\) arrives at time \(t\). By case analysis:

Consider the case where the queried item \(x\) is not the arriving one (i.e., \(x \neq x_t\)). In this case, we have \(v_{x,t} = v_{x,t-1}\). If \(x \in C_{t-1}\) but was evicted (Line 10) then \(c_x \in \arg\min_{y \in C_{t-1}} (c_{y,t-1})\). This means that:

\[q_{t-1}(x) = r_{x,t-1} + s \cdot \min_{y \in C_{t-1}} (c_{y,t-1}) \leq s - 1 + s \cdot \min_{y \in C_t} (c_{y,t}) = q_t(x),\]
where the last equation follows from the query for \( x \notin C_t \) (Line 17). Next, if \( x \in C_{t-1} \) and \( x \in C_t \), its estimated volume is determined by Line 15 and we get \( q_t(x) = q_{t-1}(x) \geq v_{x,t-1} = v_{x,t} \). If \( x \notin C_{t-1} \) then \( x \notin C_t \), so the values of \( q_t(x), q_{t-1}(x) \) are determined by line 17. Since the value of \( \min_{y \in C} c_y \) can only increase over time, we have \( q_t(x) \geq q_{t-1}(x) \geq v_{x,t} \) and the claim holds.

On the other hand, assume that we are queried about the last item, i.e., \( x = x_t \). In this case, we get \( v_{x,t} = v_{x,t-1} + w_t \). We consider the following cases: First, if \( x \in C_{t-1} \), then \( q_t(x) = q_{t-1}(x) + w_t \). Using the hypothesis, we conclude that \( v_{x,t} = v_{x,t-1} + w_t \leq q_{t-1}(x) + w_t = q_t(x) \) as required. Next, if \( |C_{t-1}| < \mathcal{C} \), we also have \( q_t(x) = q_{t-1}(x) + w_t \) and the above analysis holds. Finally, if \( x \notin C_{t-1} \) and \( |C_{t-1}| = \mathcal{C} \), then

\[
q_{t-1}(x) = s - 1 + s \cdot \min_{y \in C_{t-1}} c_{y,t-1}.
\]  

On the other hand, when \( x \) arrives, the condition of Line 2 was not satisfied, and thus

\[
q_t(x) = r_{x,t} + s \cdot c_{x,t} = (s - 1 + w) \mod s \\
+ s \cdot \left( \min_{y \in C_{t-1}} c_{y,t-1} + \left\lfloor \frac{s - 1 + w}{s} \right\rfloor \right)
\]

(Observation 5.4.1) = \( s \cdot \min_{y \in C_{t-1}} c_{y,t-1} + s - 1 + w \)

(5.1) = \( q_{t-1}(x) + w \)

\[
(\text{induction hypothesis}) \geq v_{x,t-1} + w = v_{x,t}.
\]

We continue by showing that FAST is accurate if there are only a few distinct items.

**Lemma 5.4.3.** If the stream contains at most \( \left\lceil \frac{1+4\gamma}{\epsilon} \right\rceil \) distinct elements then FAST provides an exact estimation of an items volume upon query.

**Proof.** Since \( |C| \leq \mathcal{C} \), we get that the conditions in Line 2 and Line 15 are always satisfied. Before the queried element \( x \) first appeared, we have \( r_x = c_x = 0 \) and thus \( \text{QUERY}() = 0 \). Once \( x \) appears once, it gets a counter and upon every arrival with value \( w \), the estimation for \( x \) exactly increases by \( w \), since \( x \) never gets evicted (which can only happen in Line 7).

We now analyze the sum of counters in \( C \).

**Lemma 5.4.4.** For any \( t \in \mathbb{N} \), after seeing any \((\mathcal{U}, M)\)-weighted stream \( S \) of length \( t \), FAST satisfies:

\[
\sum_{x \in C_t} \text{QUERY}() \leq t \cdot M \cdot (1 + \gamma/2).
\]

**Proof.** We prove the claim by induction on the stream length \( t \).

**Basis:** \( t = 0 \). In this case, all counters have value of 0 and thus \( \sum_{x \in C_t} q_t(x) = 0 = t \cdot (M \cdot (1 + \gamma/2)) \).

**Hypothesis:** \( \sum_{x \in C_{t-1}} q_{t-1}(x) \leq (t - 1) \cdot M \cdot (1 + \gamma/2) \).

**Step:** \( \angle x_t, w_t \) arrives at time \( t \). We consider the following cases:
1. \( x \in C_{t-1} \) or \( |C_{t-1}| < \left\lfloor \frac{1+\gamma}{\epsilon} \right\rfloor \). In this case, the condition in Line 2 is satisfied and thus \( c_{x,t} = c_{x,t-1} + \left\lfloor \frac{r_{x,t-1}+w}{s} \right\rfloor \) (Line 3) and \( r_{x,t} = (r_{x,t-1} + w) \mod s \) (Line 4). By Observation 5.4.1 we get

\[
q_t(x) = (\text{by line } 10) \ r_{x,t} + s \cdot c_{x,t}
\]

\[
= c_{x,t-1} + \left\lfloor \frac{r_{x,t-1}+w}{s} \right\rfloor + (r_{x,t-1} + w) \mod s
\]

\[
= w + c_{x,t-1} + r_{x,t-1} = q_{t-1}(x) + w. \tag{5.2}
\]

Since the value of a query for every \( y \in C_t \setminus \{x\} \) remains unchanged, we get that

\[
\sum_{y \in C_t} q_t(y) = q_t(x) + \sum_{y \in C_{t-1}} q_{t-1}(y) \tag{by (5.2)}
\]

\[
= w + q_{t-1}(x) + \sum_{y \in C_{t-1}} q_{t-1}(y)
\]

\[
= w + \sum_{y \in C_{t-1}} q_{t-1}(y) \tag{induction hypothesis}
\]

\[
\leq w + (t-1) \cdot (M \cdot (1 + \gamma/2))
\]

\[
\leq M + (t-1) \cdot (M \cdot (1 + \gamma/2)) \tag{\( \gamma \geq 0 \)}
\]

\[
\leq t \cdot (M \cdot (1 + \gamma/2)).
\]

2. \( x \notin C_{t-1} \) and \( |C_{t-1}| = \left\lfloor \frac{1+\gamma}{\epsilon} \right\rfloor \). In this case, the condition of Line 2 is false and therefore \( c_{x,t} = c_{m,t-1} + \left\lfloor \frac{s-1+w}{s} \right\rfloor \) (Line 8) and \( r_{x,t} \leftarrow (s-1+w) \mod s \) (Line 9). From Observation 5.4.1 we get that

\[
q_t(x) = (\text{by Line } 10) \ r_{x,t} + s \cdot c_{x,t}
\]

\[
= c_{m,t-1} + \left\lfloor \frac{s-1+w}{s} \right\rfloor + (s-1+w) \mod s
\]

\[
= w + c_{m,t-1} + s - 1
\]

\[
= q_{t-1}(m) - r_{m,t-1} + \left\lfloor \frac{M \gamma}{2} \right\rfloor + w
\]

\[
\leq q_{t-1}(m) + \left\lfloor \frac{M \gamma}{2} \right\rfloor + w. \tag{5.3}
\]
As before, the value of a query for every \( y \in C_t \setminus \{x\} \) is unchanged, and since \( C_{t-1} \setminus C_t = \{m\} \),

\[
\sum_{y \in C_t} q_t(y) = q_t(x) - q_{t-1}(m) + \sum_{y \in C_{t-1}} q_t(y)
\]

(by (5.3)) \leq \left\lfloor \frac{M \gamma}{2} \right\rfloor + w + \sum_{y \in C_{t-1}} q_t(y)

(\text{induction hypothesis}) \leq \left\lfloor \frac{M \gamma}{2} \right\rfloor + w + (t-1) \cdot (M \cdot (1 + \gamma/2))

\leq \left\lfloor \frac{M \gamma}{2} \right\rfloor + M + (t-1) \cdot (M \cdot (1 + \gamma/2))

(\gamma \geq 0) \leq t \cdot (M \cdot (1 + \gamma/2)). \tag*{\Box}

Next, we show a bound on FAST’s estimation error.

**Lemma 5.4.5.** For any \( t \in \mathbb{N} \), after seeing any \((U, M)\)-weighted stream \( S \) of length \( t \), for any \( x \in U : \hat{v}_x \leq v_x + t \cdot M \cdot \epsilon \).

**Proof.** First, consider the case where the stream contains at most \( \left\lfloor \frac{1+\gamma}{\epsilon} \right\rfloor \) distinct elements. By Lemma 5.4.3, \( \hat{v}_x \leq v_x \) and the claim holds. Otherwise, we have seen more than \( \left\lfloor \frac{1+\gamma}{\epsilon} \right\rfloor \) distinct elements, and specifically

\[
t > \left\lceil \frac{1 + \gamma}{\epsilon} \right\rceil. \tag{5.4}
\]

From Lemma 5.4.4, it follows that

\[
\min_{y \in C_t} \text{Query}(y) \leq \frac{t \cdot M \cdot (1 + \gamma/2)}{\left\lceil \frac{1+\gamma}{\epsilon} \right\rceil} \leq \frac{t \cdot M \cdot \epsilon \cdot (1 + \gamma/2)}{1 + \gamma}. \tag{5.5}
\]

Notice that \( \forall x \in C_t \), the value of \( \text{QUERY()} \) is determined in Line 15, that is, \( q_t(x) = r_{x,t} + s \cdot c_{x,t} \). Next, observe that an item’s remainder value is bounded by \( s - 1 \) (Line 4 and Line 9). Thus,

\[
\forall x, y \in C_t : q_t(x) \geq s + q_t(y) \implies c_{x,t} > c_{y,t}. \tag{5.6}
\]

By choosing \( y \in \arg\min_{y \in C_t} q_t(y) \), we get that if \( v_{x,t} \geq q_t(y) + s \), then \( q_t(x) \geq q_t(y) + s \) and thus \( c_{x,t} > c_{y,t} \). Next, we show that if \( v_{x,t} \geq t \cdot M \cdot \epsilon \), then \( c_x > \min_{y \in C_t} c_y \) and thus \( x \) will never be the “victim” in Line 7.

\[
q_t(x) \geq v_{x,t} \geq t \cdot M \cdot \epsilon = t \cdot M \cdot \epsilon \cdot \frac{1 + \gamma/2}{1 + \gamma} + M \gamma/2 \cdot \frac{t}{1 + \gamma} \tag{5.3} \geq q_t(y) + M \gamma/2 \cdot \frac{t}{1 + \gamma} \tag{5.4} \geq q_t(y) + M \gamma/2.
\]

Next, since \( q_t(x) \) and \( q_t(y) \) are integers, it follows that

\[
q_t(x) \geq q_t(y) + \left\lfloor \frac{M \cdot \gamma}{2} + 1 \right\rfloor = q_t(y) + s.
\]

Finally, we apply (5.6) to conclude that once \( x \) arrives with a cumulative volume of \( t \cdot M \cdot \epsilon \), it will never be evicted (Line 7) and from that moment on its volume will be measured exactly. \( \Box \)
Next, we prove a bound on the run time of FAST.

**Lemma 5.4.6.** let $\gamma > 0$, FAST adds in $O\left(\frac{1}{\gamma}\right)$ time.

**Proof.** As mentioned before, FAST utilizes the SOS data structure that answers queries in $O(1)$. Updates are a bit more complex as we need to handle weights and thus may be required to move the flow more than once, upon a counter increase. Whenever we wish to increase the value of a counter (Line 3 and Line 8), we need to remove the item from its current group and place it in a group that has the increased $c$ value. This means that for increasing a counter by $n \in \mathbb{N}$, we have to traverse at most $n$ groups until we find the correct location. Since the remainder value is at most $s - 1$ (Line 4 and Line 9), we get that at any time point, a counter is increased by no more than $\lfloor s - 1 + w \rfloor / s$ (Line 3 and Line 8). Finally, since $s = \lfloor M \cdot \gamma^2 + 1 \rfloor$, we get that the counter increase is bounded by

$$\frac{\lfloor M \cdot \gamma^2 + 1 \rfloor - 1 + w}{M \cdot \gamma^2 + 1} < 1 + \frac{w}{M \cdot \gamma^2} \leq 1 + \frac{2}{\gamma} = O\left(\frac{1}{\gamma}\right).$$

Next, we combine Lemma 5.4.2, Lemma 5.4.5 and Lemma 5.4.6 to conclude the correctness of the FAST algorithm.

**Theorem 5.4.7.** For any fixed $\gamma > 0$, when allocated with $C \triangleq \lceil 1 + \frac{1}{e} \rceil$ counters, FAST performs updates and queries in constant time, and solves the $(\epsilon, M)$-Volume Estimation problem.

Finally, FAST also solves the heavy hitters problem:

**Theorem 5.4.8.** For any fixed $\gamma > 0$, when allocated with $C \triangleq \lceil 1 + \frac{1}{e} \rceil$ counters, by returning $\{x \in U \mid \hat{v}_x \geq N \cdot M \cdot \theta\}$, FAST solves the $(W, \theta, \epsilon)$-Heavy Hitters problem.

### 5.5 Windowed FAST (WFAST)

We now present Windowed Frequent items Algorithm with a Semi-structured Table (WFAST), an efficient algorithm for the $(W, \epsilon, M)$-Volume Estimation and $(W, \theta, \epsilon, M)$-Weighted Heavy Hitters problems.

We partition the stream into consecutive sequences of size $W$ called frames. Each frame is further divided into $k \triangleq \lceil 4 \epsilon \rceil$ blocks, each of size $T$, which we assume is an integer for simplicity. Figure 5.2 illustrates the setting.

WFAST uses a FAST instance $y$ to estimate the volume of each flow within the current frame. Once a frame ends (the stream length is divisible by $W$), we “flush” the instance, i.e., reset all counters and remainders to 0. Yet, we do not “forget” all information in a flush, as high volume flows are stored in a dedicated data structure. Specifically, we say that an element $x$ overflowed at time $t$ if $\left\lfloor \frac{q_{x,t}}{MW/k} \right\rfloor > \left\lfloor \frac{q_{x,t-1}}{MW/k} \right\rfloor$. We use a queue of queues structure $b$ to keep track of which elements have overflowed in each block. That is,
Figure 5.2: The stream is divided into intervals of size $W$ called frames and each frame is partitioned into $k$ equal-sized blocks. The window of interest is also of size $W$, and overlaps with at most 2 frames and $k + 1$ blocks.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>A constant $k \triangleq \lceil 4/\varepsilon \rceil$</td>
</tr>
<tr>
<td>$y$</td>
<td>A FAST instance using $k(1 + \gamma)$ counters.</td>
</tr>
<tr>
<td>$b$</td>
<td>A queue of $k + 1$ queues.</td>
</tr>
<tr>
<td></td>
<td>An efficient implementation appears in [14].</td>
</tr>
<tr>
<td>$B$</td>
<td>The histogram of $b$, implemented using a hash table.</td>
</tr>
<tr>
<td>$M$</td>
<td>The offset within the current frame.</td>
</tr>
</tbody>
</table>

Table 5.2: Variables used by the WFAST algorithm.

Each node of the main queue represents a block and contains a queue of all elements that overflowed in its block. Particularly, the secondary queues maintain the ids of overflowing elements. Once a block ends, we remove the oldest block’s node (queue) from the main queue, and initialize a new queue for the starting block. Finally, we answer queries about the window volume of an item $x$ by multiplying its overflows count by $MW/k$, adding the residual count from $y$ (i.e., the part that is not recorded in $b$), plus $2MW/k$ to ensure an overestimation.

For $O(1)$ time queries, we also maintain a hash table $B$ that tracks the overflow count for each item. That is, for each element $x$, $B[x]$ contains the number of times $x$ is recorded in $b$. Since multiple items may overflow in the same block, we cannot update $B$ once a block ends in constant time. We address this issue by deamortizing $B$’s update, and on each arrival we remove a single item from the queue of the oldest block (if such exists). The pseudo code of WFAST appears in Algorithm 6 and a list containing its variables description appears in Table 5.2. An efficient implementation of the queue of queues $b$ is described in [14].

### 5.5.1 WFAST Analysis

We start by introducing several notations to be used in this section. We mark the queried element by $x$, the current time by $W + M$, and assume that item $W$ is the first element of the current frame. For convenience, denote $v_x(t_1, t_2) \triangleq \sum_{i \in \{t_1, \ldots, t_2\}} w_i$, i.e., the volume of $x$ between $t_1$ and $t_2$. The goal is then to approximate the window volume of $x$, which is defined as

$$v_x^w \triangleq v(M + 1, W + M) = v(M + 1, W - 1) + v(W, W + M), \quad (5.7)$$
Algorithm 6 WFAST \((W, M, \gamma)\)

Initialization: \(y \leftarrow \text{FAST}(M, 1/k, \gamma), M \leftarrow 0, B \leftarrow \text{Empty hash table}, b \leftarrow \text{Queue of } k + 1 \text{ empty queues.}\)

1: \textbf{function} ADD(Item \(x\), Weight \(w\))
2: \(M = M + 1 \mod W\)
3: \textbf{if} \(M = 0\) \textbf{then} \(\triangleright \text{new frame starts}\)
4: \(y.\text{FLUSH}()\)
5: \textbf{end if}
6: \textbf{if} \(M \mod T = 0\) \textbf{then} \(\triangleright \text{new block}\)
7: \(b.\text{POP}()\)
8: \textbf{end if}
9: \textbf{if} \(b.\text{tail is not empty}\) \(\triangleright \text{remove oldest item}\)
10: \(\text{oldID} \leftarrow b.\text{tail}.\text{POP}()\)
11: \(B[\text{oldID}] = B[\text{oldID}] - 1\)
12: \textbf{if} \(B[\text{oldID}] = 0\) \textbf{then}
13: \(B.\text{REMOVE}()\)
14: \textbf{end if}
15: \textbf{end if}
16: \textbf{end if}
17: \(\text{prevOverflowCount} \leftarrow \left\lfloor \frac{y.\text{QUERY}(x)}{MW/k} \right\rfloor\)
18: \(y.\text{ADD}(x, w)\) \(\triangleright \text{add item}\)
19: \textbf{if} \(\left\lceil \frac{y.\text{QUERY}(x)}{MW/k} \right\rceil > \text{prevOverflowCount}\) \(\triangleright \text{overflow}\)
20: \(b.\text{HEAD}.\text{PUSH}(x)\)
21: \textbf{if} \(B.\text{CONTAINS}(x)\) \textbf{then}
22: \(B[x] = B[x] + 1\)
23: \textbf{else}
24: \(B[x] \leftarrow 1\) \(\triangleright \text{adding } x \text{ to } B\)
25: \textbf{end if}
26: \textbf{end if}
27: \textbf{end function}
28: \textbf{function} WINQUERY(Item \(x\))
29: \textbf{if} \(B.\text{CONTAINS}(x)\) \textbf{then}
30: \(\text{return} \ MW/k \cdot (B[x] + 2) + (y.\text{QUERY}(x) \mod MW/k)\)
31: \textbf{else} \(\triangleright x \text{ has no overflows}\)
32: \(\text{return} 2MW/k + y.\text{QUERY}(x)\)
33: \textbf{end if}
34: \textbf{end function}

i.e., the sum of weights in the timestamps within \(\langle M + 1, M + 2, \ldots, W + M \rangle\) in which \(x\) arrived. We denote the value returned from \(y.\text{QUERY}(x)\) after the \(t\)'th item was added by \(y_t\). Similarly, \(u_t\) represents whether \(x\) arrived at time \(t\) \((x = x_t)\) \text{ and overflows, i.e., the condition of Line 19 was satisfied. We assume that if } x \text{ is not allocated a FAST counter}
at time $t$, then $B[x] = 0$, which allows us to consider Line 30 for queries. For simplicity, we mark $B[x] = 0$ for $x \notin B$.

We proceed with a useful lemma that bounds WFAST’s error on arrivals happening before the flush (Line 4).

**Lemma 5.5.1.** Let $t_1, t_2 \in \{M + 1, \ldots + W - 1\}$ be two timestamps within the previous frame, then

$$\left\lfloor \frac{v_x(t_1, t_2)}{MW/k} \right\rfloor \leq \sum_{t=t_1}^{t_2} u_t \leq \left\lceil \frac{v_x(t_1, t_2)}{MW/k} \right\rceil.$$  

**Proof.** Since $y$, initialized with $\epsilon = \frac{1}{k}$, is flushed every $W$ elements (Line 4), any element $z$ that satisfies $y.QUERY(z) > MW/k$ is guaranteed not to lose its counter (see Lemma 5.4.5’s proof for a similar analysis). This means that once an item overflows, it never loses its counter. Further, being an over-estimator, an element with a volume of $MW/k$ (or more) is guaranteed to have a counter since the minimal counter cannot exceed $MW/k$. Thus, if $v_x(t_1, t_2) < MW/k$ then the claim holds, since after overflowing for the first time, an item has to arrive with a weight of $MW/k$ to overflow again. On the other hand, if $v_x(t_1, t_2) > MW/k$ then $x$ may overflow for the first time before its volume reached $MW/k$, but from that point on only arrivals of $x$ increase the counter and can cause an overflow. \hfill \Box

We continue with proving the algorithm’s correctness.

**Theorem 5.5.2.** Algorithm 6 solves $(W, \epsilon, M)$-VOLUME ESTIMATION.

**Proof.** We prove the theorem in two steps. We first analyze the volume of $x$ within the previous frame, and then consider the current one. We continue by bounding the error introduced by the deamortization and factor FAST being an approximation algorithm in the first place. Finally, we add up the different error types and show that WFAST provides a decent approximation.

We start with the number of times $x$ has overflowed before the flush. By applying Lemma 5.5.1 we get that

$$\left\lfloor \frac{v_x(M + 1, W - 1)}{MW/k} \right\rfloor \leq \sum_{t=M+1}^{W-1} u_t \leq \left\lceil \frac{v_x(M + 1, W - 1)}{MW/k} \right\rceil.$$  

We continue with analyzing the current frame, which started after $y$ was last flushed (Line 4). As discussed above, an element whose volume is larger than $MW/k$ overflows and will not lose its counter in the flush, thus:

$$yw+M = MW/k \cdot \sum_{t=W}^{W+M} u_t + (yw+M \mod MW/k).$$  

Notice that if $x$ does not have a counter, it did not overflow and the equation still holds.

Next, we consider the number of overflows recorded in $B[x]$ and the number of actual overflows. We have deamortized (Line 11) the process of updating the overflow count (in $B$). This means that $B[x]$ is not guaranteed to have the exact count of the number of times $x$ overflowed within the blocks overlapping with the current window. Luckily, since
x cannot overflow twice in the same block, and specifically in the oldest block \((b_. \text{tail})\), we get that we underestimate the number of overflows by at most one, and specifically:

\[ \sum_{t=M+1}^{W+M} u_t - 1 \leq B[x] \leq \sum_{t=M+1}^{W+M} u_t. \]  

(5.10)

Since \(y\) is a FAST instance with parameters \((M, \frac{1}{k}, \gamma)\), it solves the \((\epsilon, M)\)-VOLUME ESTIMATION problem, thus

\[ v(W, W + M) \leq y_{W+M} \leq v(W, W + M) + MW/k. \]  

(5.11)

When queried for \(x\), the algorithm returns

\[ f^W_x = MW/k \cdot (B[x] + 2) + (y_{W+M} \mod MW/k) \]

\[ = MW/k \cdot (B[x] + 2 - \sum_{t=W}^{W+M} u_t) + y_{W+M}. \]

We combine the above inequalities to bound the overestimation:

\[ f^W_x = MW/k \cdot (B[x] + 2 - \sum_{t=W}^{W+M} u_t) + y_{W+M} \]

\[ \leq (5.11) \quad MW/k \cdot (B[x] + 3 - \sum_{t=W}^{W+M} u_t) + v(W, W + M) \]

\[ \leq (5.10) \quad MW/k \cdot \left( \sum_{t=M+1}^{W+M} u_t + 3 - \sum_{t=W}^{W+M} u_t \right) + v(W, W + M) \]

\[ = MW/k \cdot \left( \sum_{t=M+1}^{W-1} u_t + 3 \right) + v(W, W + M) \]

\[ \leq (\text{Lemma} \ 5.5.1) \quad \frac{v_x(M + 1, W - 1)}{MW/k} + 3 + v(W, W + M) \]

\[ \leq (5.7) \quad v(M + 1, W + M) + 4MW/k \leq f^W_x + WM\epsilon. \]

Similarly, we bound the query value from below:

\[ f^W_x = MW/k \cdot (B[x] + 2 - \sum_{t=W}^{W+M} u_t) + y_{W+M} \]

\[ \geq (5.11) \quad MW/k \cdot (B[x] + 2 - \sum_{t=W}^{W+M} u_t) + v(W, W + M) \]

\[ \geq (5.10) \quad MW/k \cdot \left( \sum_{t=M+1}^{W+M} u_t + 1 - \sum_{t=W}^{W+M} u_t \right) + v(W, W + M) \]

\[ = MW/k \cdot \left( \sum_{t=M+1}^{W-1} u_t + 1 \right) + v(W, W + M) \]

\[ \geq (\text{Lemma} \ 5.5.1) \quad \frac{v_x(M + 1, W - 1)}{MW/k} + 1 + v(W, W + M) \]

\[ \geq (5.7) \quad v(M + 1, W + M) = f^W_x. \]

Showing both bounds, we established that WFAST solves the \((W, \epsilon, M)\)-FREQUENCY ESTIMATION problem.

As a corollary, Algorithm \(\Theta\) can find heavy hitters.

**Theorem 5.5.3.** By returning all items \(x \in \mathcal{U}\) for which \(f^W_x \geq MW\theta\), Algorithm \(\Theta\) solves \((W, \theta, \epsilon, M)\)-WEIGHTED HEAVY HITTERS.
Figure 5.3: The effect of parameter $\gamma$ on operation speed for different error guarantees ($\epsilon$). $\gamma$ influences the space requirement as the algorithm is allocated with $\lceil \frac{1+\gamma}{\epsilon} \rceil$ counters.

**WFAST runtime analysis:**

As listed in the pseudo code of WFAST (see Algorithm 6) and the description above, processing new elements requires adding them to the FAST instance $y$, which takes $O\left(\frac{1}{\gamma}\right)$ time, and another $O(1)$ operations. The query processing includes $O(1)$ operations and hash tables accesses. For returning the heavy hitters, we go over all of the items with allocated counters in time $O\left(\frac{1+\gamma}{\epsilon}\right)$. In summary, we get the following theorem:

**Theorem 5.5.4.** For any fixed $\gamma > 0$, WFAST processes new elements and answers window-volume queries in constant time, while finding the window’s weighted heavy hitters in $O(\epsilon^{-1})$ time.

### 5.6 Hierarchical Heavy Hitters

*Hierarchical heavy hitters (HHH)* algorithms treat IP addresses as a hierarchical domain. At the bottom are fully specified IP addresses such as $p_0 = 101.102.103.104$. Higher layers include shorter and shorter prefixes of the fully specified addresses. For example, $p_1 = 101.102.103.$ and $p_2 = 101.102.$ are level 1 and level 2 prefixes of $p_0$, respectively. Such prefixes generalize an IP address. In this example, $p_0 \prec p_1 \prec p_2$, indicating that
Figure 5.4: Runtime comparison for a given error guarantee ($\epsilon = 2^{-8}$). All algorithms provide the same guarantees and FAST uses different $\gamma$ values to show the speedup gained from allocating additional counters.

$p_0$ satisfies the pattern of $p_1$, and any IP address that satisfies $p_1$ also satisfies $p_2$. The above example refers to a single dimension (e.g., the source IP), and can be generalized to multiple dimensions (e.g., pairs of source IP and destination IP). HHH algorithms need to find the heavy hitter prefixes at each level of the induced hierarchy. For example, this enables identifying heavy hitters subnets, which may be suspected of generating a DDoS attack. The problem is formally defined in [38, 40].

**Hierarchical Fast (HFAST)**

Hierarchical FAST (HFAST) is derived from the algorithm of [38]. Specifically, the work of [38] suggests Hierarchical Space Saving with a Heap (HSSH). In their work, the HHH prefixes are distilled from multiple solutions of plain heavy hitter problems. That is, each prefix pattern has its own separate heavy hitters algorithm that is updated on each packet arrival. For example, consider a packet whose source IP address is 101.102.103.104 where the (one dimensional) HHH measurements are carried according to source addresses. In this case, the packet arrival is translated into the following five heavy hitters update operations: 101.102.103.104, 101.102.103.*, 101.102.*, 101.*, and *. Finally, HHHs are identified by calculating the heavy hitters of each separate heavy hitters algorithm.
Figure 5.5: Runtime as a function of accuracy guarantee ($\epsilon$) provided by the algorithms.

Figure 5.6: Space overheads of WFAST compared to previous works. Note that WFAST operates in constant time while the other algorithm requires linear scanning of all counters.
HFAST is derived by replacing the underlying heavy hitters algorithm in [38] from Space Saving with heap [30] to FAST. This asymptotically improves the update complexity from $O \left( H \log \left( \frac{1}{\varepsilon} \right) \right)$ to $O(H)$, where $H$ is the size of the hierarchy. Since the analysis of [38] is indifferent to the internal implementation of the heavy hitters algorithm, no further analysis is required for HFAST.

Finally, we note that a hierarchical heavy hitters algorithm on sliding windows can be constructed using the work of [38] by replacing each space saving instance with our
<table>
<thead>
<tr>
<th>Trace</th>
<th>Chicago16</th>
<th>Chicago15</th>
<th>SanJose14</th>
<th>SanJose13</th>
<th>DC1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date(Y/M/D)</td>
<td>2016/02/18</td>
<td>2015/12/17</td>
<td>2014/06/19</td>
<td>2013/12/19</td>
<td>2010</td>
</tr>
<tr>
<td>#Packets</td>
<td>97M</td>
<td>85M</td>
<td>112M</td>
<td>97M</td>
<td>2010</td>
</tr>
<tr>
<td>Total Volume</td>
<td>94GB</td>
<td>80GB</td>
<td>149GB</td>
<td>110GB</td>
<td>6.1GB</td>
</tr>
<tr>
<td>Mean Size</td>
<td>1046B</td>
<td>1013B</td>
<td>1424B</td>
<td>1255B</td>
<td>894B</td>
</tr>
<tr>
<td>Max Size</td>
<td>49458B</td>
<td>64134B</td>
<td>65535B</td>
<td>65528B</td>
<td>1476B</td>
</tr>
<tr>
<td>Large Packets</td>
<td>0.34%</td>
<td>0.22%</td>
<td>0.78%</td>
<td>0.49%</td>
<td>0%</td>
</tr>
<tr>
<td>Large Pkt Traffic</td>
<td>0.5%</td>
<td>0.34%</td>
<td>25.02%</td>
<td>18.81%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 5.3: A summary of key characteristics of the real Internet traces used in this work.

WFAST. The complexity of the proposed algorithm is $O\left(\frac{H}{\varepsilon}\right)$ space and $O(H)$ update time. To our knowledge, there is no prior work for this problem.

## 5.7 Evaluation

Our evaluation is performed on an Intel i7-5500U CPU with a clock speed of 2.4GHz, 16 GB RAM and a Windows 8.1 operating system. We compare our C++ prototypes to the following alternatives:

**Count Min Sketch (CMS)** [28] – a sketch based solution that can only solve the volume estimation problem.

**Space Saving Heap (SSH)** – a heap based implementation [56] of Space Saving [30] that has a logarithmic runtime complexity.

**Hierarchical Space Saving Heap (HSSH)** – an HHH algorithm [38] that uses SSH as a building block and operates in $O(H \log(\frac{1}{\varepsilon}))$ complexity.

**Full Ancestry** – a trie based HHH algorithm suggested by [40], which operates in $O(H \log \epsilon N)$ complexity.

**Partial Ancestry** – a trie based HHH algorithm suggested by [40], which operates in $O(H \log \epsilon N)$ complexity and is faster than Full Ancestry.

Related work implementations were taken from open source libraries released by [37] for streams and by [38] for hierarchical heavy hitters. As we have no access to a concrete implementation of a competing sliding window protocol, we compare WFAST to Hung and Ting’s algorithm [79] by conservatively estimating the space needed by their approach. Each data point we report here is the average of 10 runs.

### 5.7.1 Datasets

Our evaluation includes the following datasets:

1. The CAIDA backbone Internet traces that monitor links in Chicago [86, 87] and San Jose [88, 70]. The data includes a mix of UDP/TCP and ICMP packets. The
Chicago16 [87] and Chicago15 [86] data sets were taken from the ‘equinix-chicago’ high-speed monitor in 2016 and 2015, respectively. Similarly, the SanJose14 [70] and SanJose13 [88] were taken from the ‘equinix-sanjose’ monitor.

2. A datacenter trace from a large university [89].

3. A trace of 436K YouTube video accesses [90]. The weight of a video is its length in seconds.

4. Self generated synthetic traces following a Zipfian distribution with varying skews. A trace of skew X is denoted ZIPfX. Each trace is unweighed (each element has weight 1) and contains 10M elements.

A summary of key characteristics for CAIDA traces is given in Table 5.3. As can be seen, the impact of jumbo frames varies between backbone links. Yet, the weight of large packets increases over time in both. In the San Jose link, the number and volume of large packets have increased by 50% within a period of 6 months. For Chicago, large packets are still insignificant, but their number and volume have increased by 50% in two months.

5.7.2 Effect of $\gamma$ on Runtime

We begin the evaluation by exploring our trade-off parameter $\gamma$. Recall that smaller $\gamma$ yields space efficiency while the runtime is proportional to $\frac{1}{\gamma}$, i.e, smaller $\gamma$ is expected to cause a slower runtime. Figure 5.3 shows runtime performance of FAST as a function of $\gamma$ for three different $\epsilon$ values ($2^{-8}$, $2^{-10}$, $2^{-12}$). As can be observed, in practice, we indeed get speedup with larger $\gamma$ values. But, we reach a saturation point and increasing $\gamma$ beyond a certain threshold has little impact on performance. It is encouraging that even with small values of $\gamma$ such as $2^{-7}$, FAST is still reasonably fast. For the rest of our evaluation, we focus on $\gamma = 0.25$ that offers attractive space/time trade off, as well as on $\gamma = 4$ that yields higher performance at the expense of more space.

5.7.3 Speed vs. Space Tradeoff

To explain the tradeoff proposed by FAST, we measured the runtime of the various algorithms for a fixed error guarantee. Here, SSH and CMS are fully determined by the error guarantees (set to $\epsilon = 2^{-8}$) represented by a single measurement point. CMS requires more counters as it uses 10 rows of $\lceil e/\epsilon \rceil$ counters each, while SSH only requires $1/\epsilon$. FAST can provide the same error guarantee for different $\gamma$ values, which affects both runtime and the number of counters. Hence, FAST is represented by a curve. As Figure 5.4 shows, in all traces, allocating a few additional counters to the $1/\epsilon$ required by SSH allows FAST to achieve higher throughput. Further, on all traces, FAST provides faster throughput than CMS with far fewer counters. While FAST has larger per counter
overheads than CMS, its ID to counter mapping allows it to solve the Weighted Heavy Hitters problem that CMS cannot.

5.7.4 Operation Speed Comparison

Figure 5.5 presents a comparative analysis of the operation speed of previous approaches. Recall that CMS is a probabilistic scheme; we configured it with a failure probability of 0.1%. For FAST, we used two configurations: \( \gamma = 4 \) (4FAST) and \( \gamma = 0.25 \) (0.25FAST).

As can be observed, 4FAST and 0.25FAST are considerably faster than the alternatives in Chicago16 and YouTube. In SanJose14 and SanJose13, SSH is as fast as 4FAST for a large \( \epsilon \) (small number of counters). Yet, as \( \epsilon \) decreases and the number of counters increases, SSH becomes slower due to its logarithmic complexity. In contrast, CMS is almost workload independent. When considering only previous work, in some workloads CMS is faster than SSH, mainly because SSH’s performance is workload dependent. The bottom 3 figures (g,h,i) show results for synthetic unweighted Zipf traces with skew parameters of 0.7, 1, 1.3, respectively. As can be observed, for mildly skewed distributions, CMS is faster than SSH, while for skewed distributions such as when the skew is 1.3, SSH is faster. In all these measurements, 4FAST is faster than the alternatives.

5.7.5 Sliding Window

We evaluate WFAST compared to Hung and Ting’s algorithm \[79\], which is the only one that supports weighted updates on sliding windows. Figure 5.6 shows the memory consumption of WFAST with parameters \( \gamma = 4 \) and \( \gamma = 0.25 \) (4WFAST, 0.25FAST) compared to Hung and Ting’s algorithm. All algorithms are configured to provide the same worst case error guarantee. As shown, WFAST is up to 100 times more space efficient than Hung and Ting’s algorithm. Sadly, we could not obtain an implementation of Hung and Ting’s algorithm and thus do not compare its runtime to WFAST. However, WFAST improves their update complexity from \( O(\epsilon) \), where \( A \) is the average packet size, to \( O(1) \).

Figure 5.7 shows the operation speed of WFAST for different window sizes and different \( \epsilon \) values. As seen, WFAST achieves over 15 million updates per second using a single thread. It is about half as fast as FAST for streams and still within the range of acceptable parameters. There is little dependence in window size and \( \epsilon \) with the exception of the DC1 dataset. In this dataset, since the average and maximal packet sizes are similar, the inner working of WFAST causes overflows to be more frequent when \( \epsilon \) is close to the window size. Thus, to achieve similar performance as the other traces one needs a sufficiently large window sized.
5.7.6 Hierarchical Heavy Hitters

In Figure 5.8, we evaluate the speed of our HFAST compared to the algorithm of [38], which is denoted by HSSH, as well as the Partial Ancestry and Full Ancestry algorithms by [40]. We used the library of [38] for their own HSSH implementation as well as for the Partial Ancestry and Full Ancestry implementations. Since the library was released for Linux, we used a different machine for our HFAST evaluation. Specifically, we used a Dell 730 server running Ubuntu 16.04.01 release. The server has 128GB of RAM and an Intel(R) Xeon(R) CPU E5-2667 v4 @ 3.20GHz processor.

We used two dimensional source/destination hierarchies in byte granularity, where networks IDs are assumed to be 8, 16 or 24 bits long. The weight of each packet is its byte volume, including both the payload size and the header size. As depicted, HFAST is up to 7 times faster than the best alternative and at least 2.4 times faster in every data point.

5.8 Discussion

In this chapter, we presented algorithms for estimating per flow traffic volume in streams, sliding windows and hierarchical domains. We achieved asymptotic and empirical improvements.

For streams, FAST processes packets in constant time while being asymptotically space optimal. This is enabled by our novel approach of maintaining only a partial order between counters. An evaluation over real-world traffic traces as well as synthetic ones has yielded a speed improvement of up to 2.4X compared to previous work. This is significant since the combination of fast line rates with NFV trends imposes strict timing constraints.

In the sliding window case, we showed that WFAST works reasonably fast and offers 100x reduction in required space, bringing sliding windows to the realm of possibility. For a given error of $W \cdot M \cdot \epsilon$, WFAST requires $O(H)$ counters while previous work uses $O \left( \frac{A}{\epsilon} \right)$, where $A$ is the average packet size. Moreover, it performs updates in constant time whereas previous works do so in $O \left( \frac{A}{\epsilon} \right)$ time.

For hierarchical domains, we presented HFAST that requires $O(H \epsilon)$ space and has $O(H)$ update complexity. This asymptotically improves previous works. Additionally, we demonstrated a speedup of 2.4X-7X on real Internet traces. To our knowledge, there is no prior work on that problem and we plan to examine its possible applications in the future.

The code of FAST is available as open source [91].
Chapter 6

Hierarchical Heavy Hitters

Monitoring tasks, such as anomaly and DDoS detection, require identifying frequent flow aggregates based on common IP prefixes. These are known as hierarchical heavy hitters (HHH), where the hierarchy is determined based on the type of prefixes of interest in a given application. The per packet complexity of existing HHH algorithms is proportional to the size of the hierarchy, imposing significant overheads.

In this chapter, we propose a randomized constant time algorithm for HHH. We prove probabilistic precision bounds backed by an empirical evaluation. Using four real Internet packet traces, we demonstrate that our algorithm indeed obtains comparable accuracy and recall as previous works, while running up to 62 times faster. Finally, we extended Open vSwitch (OVS) with our algorithm and showed it is able to handle 13.8 million packets per second. In contrast, incorporating previous works in OVS only obtained 2.5 times lower throughput.

6.1 Background

Network measurements are essential for a variety of network functionalities such as traffic engineering, load balancing, quality of service, caching, anomaly and intrusion detection \[92, 93, 47, 49, 50, 94, 95, 52\]. A major challenge in performing and maintaining network measurements comes from rapid line rates and the large number of active flows. Previous works suggested identifying Heavy Hitter (HH) flows \[96\] that account for a large portion of the traffic. Indeed, approximate HH are used in many functionalities and can be captured quickly and efficiently \[69, 76, 14, 44\]. However, applications such as anomaly detection and Distributed Denial of Service (DDoS) attack detection require more sophisticated measurements \[71, 72\]. In such attacks, each device generates a small portion of the traffic but their combined volume is overwhelming. HH measurement is therefore insufficient as each individual device is not a heavy hitter.

Hierarchical Heavy Hitters (HHH) account aggregates of flows that share certain IP prefixes. The structure of IP addresses implies a prefix based hierarchy as defined
Figure 6.1: A high level overview of this work. Previous algorithms’ update requires $\Omega(H)$ run time, while we perform at most a single $O(1)$ update.

more precisely below. In the DDoS example, HHH can identify IP prefixes that are suddenly responsible for a large portion of traffic and such an anomaly may very well be a manifesting attack. Further, HHH can be collected in one dimension, e.g., a single source IP prefix hierarchy, or in multiple dimensions, e.g., a hierarchy based on both source and destination IP prefixes.

Previous works $[38,40]$ suggested deterministic algorithms whose update complexity is proportional to the hierarchy’s size. These algorithms are currently too slow to cope with line speeds. For example, a 100 Gbit link may deliver over 10 million packets per second, but previous HHH algorithms cannot cope with this line speed on existing hardware. The transition to IPv6 is expected to increase hierarchies’ sizes and render existing approaches even slower.

Emerging networking trends such as Network Function Virtualization (NFV) enable virtual deployment of network functionalities. These are run on top of commodity servers rather than on custom made hardware, thereby improving the network’s flexibility and reducing operation costs. These trends further motivate fast software based measurement algorithms.

6.2 Contributions

First, we define a probabilistic relaxation of the HHH problem. Second, we introduce Randomized HHH (a.k.a. RHHH), a novel randomized algorithm that solves probabilistic
HHH over single and multi dimensional hierarchical domains. Third, we evaluate RHHH on four different real Internet traces and demonstrate a speedup of up to X62 while delivering similar accuracy and recall ratios. Fourth, we integrate RHHH with Open vSwitch (OVS) and demonstrate a capability of monitoring HHH at line speed, achieving a throughput of up to 13.8M packets per second. Our algorithm also achieves X2.5 better throughput than previous approaches. To the best of our knowledge, our work is the first to perform OVS multi dimensional HHH analysis in line speed.

Intuitively, our RHHH algorithm operates in the following way, as illustrated in Figure 6.1: We maintain an instance of a heavy-hitters detection algorithm for each level in the hierarchy, as is done in [38]. However, whenever a packet arrives, we randomly select only a single level to update using its respective instance of heavy-hitters rather than updating all levels (as was done in [38]). Since the update time of each individual level is $O(1)$, we obtain an $O(1)$ worst case update time. The main challenges that we address in this chapter are in formally analyzing the accuracy of this scheme and exploring how well it works in practice with a concrete implementation.

The update time of previous approaches is $O(H)$, where $H$ is the size of the hierarchy. An alternative idea could have been to simply sample each packet with probability $\frac{1}{H}$, and feed the sampled packets to previous solutions. However, such a solution only provides an $O(1)$ amortized running time. Bounding the worst case behavior to $O(1)$ is important when the counters are updated inside the data path. In such cases, performing an occasional very long operation could both delay the corresponding “victim” packet, and possibly cause buffers to overflow during the relevant long processing. Even in off-path processing, such as in an NFV setting, occasional very long processing creates an unbalanced workload, challenging schedulers and resource allocation schemes.

Roadmap The rest of this chapter is organized as follows: We survey related work on HHH in Section 6.3. We introduce the problem and our probabilistic algorithm in Section 6.4. For presentational reasons, we immediately move on to the performance evaluation in Section 6.5 followed by describing the implementation in OVS in Section 6.6. We then prove our algorithm and analyze its formal guarantees in Section 6.7. Finally, we conclude with a discussion in Section 6.8.

6.3 Related Work

In one dimension, HHH were first defined by [80], which also introduced the first streaming algorithm to approximate them. Additionally, [39] offered a TCAM approximate HHH algorithm for one dimension. The HHH problem was also extended to multiple dimensions [81, 40, 82, 71, 38].

The work of [33] introduced a single dimension algorithm that requires $O\left(\frac{H^2}{\epsilon}\right)$ space, where the symbol $H$ denotes the size of the hierarchy and $\epsilon$ is the allowed relative error.
estimation error for each single flow’s frequency. Later, \cite{31} introduced a two dimensions algorithm that requires $O\left(\frac{H^3}{\epsilon^2}\right)$ space and update time. In \cite{40}, the trie based Full Ancestry and Partial Ancestry algorithms were proposed. These use $O\left(\frac{H \log(N \epsilon)}{\epsilon}\right)$ space and requires $O\left(\frac{H}{\epsilon}\right)$ time per update.

The seminal work of \cite{38} introduced and evaluated a simple multi dimensional HHH algorithm. Their algorithm uses a separate copy of Space Saving \cite{30} for each lattice node and upon packet arrival, all lattice nodes are updated. Intuitively, the problem of finding hierarchical heavy hitters can be reduced to solving multiple non hierarchical heavy hitters problems, one for each possible query. This algorithm provides strong error and space guarantees and its update time does not depend on the stream length. Their algorithm requires $O\left(\frac{H}{\epsilon}\right)$ space and its update time for unitary inputs is $O\left(\frac{H}{\epsilon}\right)$ while for weighted inputs it is $O\left(\frac{H \log \frac{1}{\epsilon}}{\epsilon}\right)$.

The update time of existing methods is too slow to cope with modern line speeds and the problem escalates in NFV environments that require efficient software implementations. This limitation is both empirical and asymptotic as some settings require large hierarchies.

Our work describes a novel algorithm that solves a probabilistic version of the hierarchical heavy hitters problem. We argue that in practice, our solution’s quality is similar to previously suggested deterministic approaches while the runtime is dramatically improved. Formally, we improve the update time to $O(1)$, but require a minimal number of packets to provide accuracy guarantees. We argue that this trade off is attractive for many modern networks that route a continuously increasing number of packets.

### 6.4 Randomized HHH (RHHH)

We start with an intuitive introductory to the field as well as preliminary definitions and notations. Table \ref{tab:notations} summarizes notations used in this work.

\footnote{Notice that in two dimensions, $H$ is a square of its counter-part in one dimension.}
6.4.1 Basic terminology

We consider IP addresses to form a hierarchical domain with either bit or byte size granularity. Fully specified IP addresses are the lowest level of the hierarchy and can be generalized. We use \( U \) to denote the domain of fully specified items. For example, 181.7.20.6 is a fully specified IP address and 181.7.20.* generalizes it by a single byte. Similarly, 181.7.* generalizes it by two bytes and formally, a fully specified IP address is generalized by any of its prefixes. The parent of an item is the longest prefix that generalizes it.

In two dimensions, we consider a tuple containing source and destination IP addresses. A fully specified item is fully specified in both dimensions. For example, \((181.7.20.6)\rightarrow(208.67.222.222)\) is fully specified. In two dimensional hierarchies, each item has two parents, e.g., \((181.7.20.*)\rightarrow(208.67.222.222))\) and \((181.7.20.6)\rightarrow(208.67.222.*)\) are both parents to \((181.7.20.6)\rightarrow(208.67.222.222))\).

Generalization For two prefixes \(p, q\), we denote \(p \preceq q\) if in any dimension it is either a prefix of \(q\) or is equal to \(q\). We also denote the set of elements that are generalized by \(p\) with \(H_p \triangleq \{e \in U \mid e \preceq p\}\), and those generalized by a set of prefixes \(P\) by \(H_P \triangleq \bigcup_{p \in P} H_p\). If \(p \preceq q\) and \(p \neq q\), we denote \(p \prec q\).

In a single dimension, the generalization relation defines a vector going from fully generalized to fully specified. In two dimensions, the relation defines a lattice where each item has two parents. A byte granularity two dimensional lattice is illustrated in Table 6.2. In the table, each lattice node is generalized by all nodes that are upper or more to the left. The most generalized node \((*,*)\) is called fully general and the most specified node \((s1.s2.s3.s4,d1.d2.d3.d4)\) is called fully specified. We denote \(H\) the hierarchy’s size as the number of nodes in the lattice. For example, in IPv4, byte level one dimensional hierarchies imply \(H = 5\) as each IP address is divided into four bytes and we also allow querying *.

Definition Given a prefix \(p\) and a set of prefixes \(P\), we define \(G(p|P)\) as the set of prefixes:

\[
\{h : h \in P, h \prec p, \nexists h' \in P \text{ s.t. } h \prec h' \prec p\}.
\]

Intuitively, \(G(p|P)\) are the prefixes in \(P\) that are most closely generalized by \(p\). E.g., let \(p = <142.14.*>\) and the set \(P = \{<142.14.13.*>,<142.14.13.14>\}\), then \(G(p|P)\) only contains \(<142.14.13.*>\).

We consider a stream \(S\), where at each step a packet of an item \(e\) arrives. Packets belong to a hierarchical domain of size \(H\), and can be generalized by multiple prefixes as explained above. Given a fully specified item \(e\), \(f_e\) is the number of occurrences \(e\) has in \(S\). Definition 6.4.1 extends this notion to prefixes.
Definition (Frequency) Given a prefix $p$, the frequency of $p$ is:

$$f_p \triangleq \sum_{e \in H_p} f_e.$$ 

Our implementation utilizes Space Saving [30], a popular (non hierarchical) heavy hitters algorithm, but other algorithms can also be used. Specifically, we can use any counter algorithm that satisfies Definition 6.4.1 below and can also find heavy hitters, such as [30, 29, 35]. We use Space Saving because it is believed to have an empirical edge over other algorithms [37, 57, 56].

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>Stream</td>
</tr>
<tr>
<td>$N$</td>
<td>Current number of packets (in all flows)</td>
</tr>
<tr>
<td>$H$</td>
<td>Size of Hierarchy</td>
</tr>
<tr>
<td>$V$</td>
<td>Performance parameter, $V \geq H$</td>
</tr>
<tr>
<td>$S^i_x$</td>
<td>Variable for the $i$’th appearance of a prefix $x$.</td>
</tr>
<tr>
<td>$S_x$</td>
<td>Sampled prefixes with id $x$.</td>
</tr>
<tr>
<td>$S$</td>
<td>Sampled prefixes from all ids.</td>
</tr>
<tr>
<td>$\mathcal{U}$</td>
<td>Domain of fully specified items.</td>
</tr>
<tr>
<td>$\epsilon, \epsilon_s, \epsilon_a$</td>
<td>Overall, sample, algorithm’s error guarantee.</td>
</tr>
<tr>
<td>$\delta, \delta_s, \delta_a$</td>
<td>Overall, sample, algorithm confidence.</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Threshold parameter.</td>
</tr>
<tr>
<td>$C_{q</td>
<td>P}$</td>
</tr>
<tr>
<td>$G(q</td>
<td>P)$</td>
</tr>
<tr>
<td>$f_q$</td>
<td>Frequency of prefix $q$</td>
</tr>
<tr>
<td>$f_q^+, f_q^-$</td>
<td>Upper, lower bound for $f_q$</td>
</tr>
</tbody>
</table>

Table 6.1: List of Symbols

The minimal requirements from an algorithm to be applicable to our work are defined in Definition 6.4.1. This is a weak definition and most counter algorithms satisfy it with $\delta = 0$. Sketches [27, 28, 63] can also be applicable here, but to use them, each sketch should also maintain a list of heavy hitter items (Definition 6.4.1).

Definition An algorithm solves the $(\epsilon, \delta)$ - FREQ UENCY ESTIMATION problem if for any prefix $(x)$, it provides $\hat{f}_x$ s.t.:

$$\Pr \left[ |f_x - \hat{f}_x| \leq \epsilon N \right] \geq 1 - \delta.$$

Heavy hitter (HH) Given a threshold $(\theta)$, a fully specified item $(e)$ is a heavy hitter if its frequency $(f_e)$ is above the threshold: $\theta \cdot N$, i.e., $f_e \geq \theta \cdot N$.

Our goal is to identify the hierarchical heavy hitter prefixes whose frequency is above the threshold $(\theta \cdot N)$. However, if the frequency of a prefix exceeds the threshold then so
is the frequency of all its ancestors. For compactness, we are interested in prefixes whose frequency is above the threshold due to non HHH siblings. This motivates the definition of conditioned frequency \((C_p|P)\). Intuitively, \(C_p|P\) measures the additional traffic prefix \(p\) adds to a set of previously selected HHHs \((P)\), and it is defined as follows.

**Definition** (Conditioned frequency) The conditioned frequency of a prefix \(p\) with respect to a prefix set \(P\) is:

\[
C_p|P \triangleq \sum_{e \in H(P \cup \{p\}) \setminus H_P} f_e.
\]

\(C_p|P\) is derived by subtracting the frequency of fully specified items that are already generalized by items in \(P\) from \(p\)'s frequency \((f_p)\). In two dimensions, exclusion inclusion principles are used to avoid double counting.

We now continue and describe how exact hierarchical heavy hitters (with respect to \(C_p|P\)) are found. To that end, partition the hierarchy to levels as explained in Definition 6.4.1.

**Hierarchy Depth** Define \(L\), the depth of a hierarchy, as follows: Given a fully specified element \(e\), we consider a set of prefixes such that: \(e \prec p_1 \prec p_2 \prec \ldots \prec p_L\) where \(e \neq p_1 \neq p_2 \neq \ldots \neq p_L\) and \(L\) is the maximal size of that set. We also define the function \(\text{level}(p)\) that given a prefix \(p\) returns \(p\)'s maximal location in the chain, i.e., the maximal chain of generalizations that ends in \(p\).

To calculate exact heavy hitters, we go over fully specified items (\(\text{level}(0)\)) and add their heavy hitters to the set \(HHH_0\). Using \(HHH_0\), we calculate conditioned frequency for prefixes in \(\text{level}(1)\) and if \(C_p|HHH_0 \geq \theta \cdot N\) we add \(p\) to \(HHH_1\). We continue this process until the last level \((L)\) and the exact heavy hitters are the set \(HHH_L\). Next, we define \(HHH\) formally.

**Hierarchical HH (HHH)** The set \(HHH_0\) contains the fully specified items \(e\) s.t. \(f_e \geq \theta \cdot N\). Given a prefix \(p\) from \(\text{level}(l)\), \(0 \leq l \leq L\), we define:

\[
HHH_l = HHH_{l-1} \cup \{p : (p \in \text{level}(l) \land C_p|HHH_{l-1} \geq \theta \cdot N)\}.
\]

The set of exact hierarchical heavy hitters \(HHH\) is defined as the set \(HHH_L\).

For example, consider the case where \(\theta N = 100\) and assume that the following prefixes with their frequencies are the only ones above \(\theta N\). \(p_1 = (\langle 101.*, \rangle, 108)\) and \(p_2 = (\langle 101.102.*, \rangle, 102)\). Clearly, both prefixes are heavy hitters according to Definition 6.4.1. However, the conditioned frequency of \(p1\) is \(108 - 102 = 6\) and that of \(p2\) is \(102\). Thus only \(p2\) is an HHH prefix.

Finding exact hierarchical heavy hitters requires plenty of space. Indeed, even finding exact (non hierarchical) heavy hitters requires linear space [97]. Such a memory requirement is prohibitively expensive and motivates finding approximate HHHs.
\((\epsilon, \theta)\)–approximate HHH  An algorithm solves \((\epsilon, \theta)\) - \textsc{Approximate Hierarchical Heavy Hitters} if after processing any stream \(S\) of length \(N\), it returns a set of prefixes \((P)\) that satisfies the following conditions:

- **Accuracy**: for every prefix \(p \in P\), \(|f_p - \hat{f}_p| \leq \epsilon N\).
- **Coverage**: for every prefix \(q \notin P\): \(C_q|P < \theta N\).

Approximate HHH are a set of prefixes \((P)\) that satisfies accuracy and coverage; there are many possible sets that satisfy both these properties. Unlike exact HHH, we do no require that for \(p \in P\), \(C_p|P \geq \theta N\). Unfortunately, if we add such a requirement then \cite{82} proved a lower bound of \(\Omega\left(\frac{d}{(\Theta \epsilon)^{1}}\right)\) space, where \(d\) is the number of dimensions. This is considerably more space than is used in our work \((\mathcal{H}^\epsilon)\) that when \(\theta \propto \epsilon\) is also \(\mathcal{H}^\frac{\epsilon}{\theta}\).

Finally, Definition \ref{def:approx:hh:prob} defines the probabilistic approximate HHH problem that is solved in this chapter.

\((\delta, \epsilon, \theta)\)–approximate HHHs  An algorithm \(A\) solves \((\delta, \epsilon, \theta)\) - \textsc{Approximate Hierarchical Heavy Hitters} if after processing any stream \(S\) of length \(N\), it returns a set of prefixes \(P\) that, for an arbitrary run of the algorithm, satisfies the following:

- **Accuracy**: for every prefix \(p \in P\),
  \[
  \Pr\left(\left|f_p - \hat{f}_p\right| \leq \epsilon N\right) \geq 1 - \delta.
  \]
- **Coverage**: given a prefix \(q \notin P\),
  \[
  \Pr\left(C_q|P < \theta N\right) \geq 1 - \delta.
  \]

Notice that this is a simple probabilistic relaxation of Definition \ref{def:approx:hh:prob}. Our next step is to show how it enables the development of faster algorithms.

### 6.4.2 Randomized HHH

Our work employs the data structures of \cite{38}. That is, we use a matrix of \(H\) independent HH algorithms, and each node is responsible for a single prefix pattern.

Our solution, Randomized HHH (RHHH), updates at most a single randomly selected HH instance that operates in \(O(1)\). In contrast, \cite{38} updates every HH algorithm for each packet and thus operates in \(O(H)\).

Specifically, for each packet, we randomize a number between 0 and \(V\) and if it is smaller than \(H\), we update the corresponding HH algorithm. Otherwise, we ignore the packet. Clearly, \(V\) is a performance parameter: when \(V = H\), every packet updates one of the HH algorithms whereas when \(V \gg H\), most packets are ignored. Intuitively, each HH algorithm receives a \textit{sample} of the stream. We need to prove that given enough traffic, hierarchical heavy hitters can still be extracted.
Algorithm 7 Randomized HHH algorithm

Initialization: $\forall d \in [L] : HH[d] = HH_{\text{Alg}}(\epsilon^{-1})$

1: function UPDATE($x$)
2:     $d = \text{randomInt}(0, V)$
3:     if $d < H$ then
4:         Prefix $p = x \& HH[d].mask$ \hspace{2cm} \triangleright \text{Bitwise AND}
5:         $HH[d].INCREMENT(p)$
6:     end if
7: end function

8: function OUTPUT($\theta$)
9:     $P = \emptyset$
10:    for Level $l = |H|$ down to 0. do
11:        for each $p$ in level $l$ do
12:            $\hat{C}_{p|P} = \hat{f}_p^+ + \text{calcPred}(p, P)$
13:            $\tilde{C}_{p|P} = \hat{C}_{p|P} + 2Z_{1-\delta}\sqrt{NV}$
14:            if $\tilde{C}_{p|P} \geq \theta N$ then
15:                $P = P \cup \{p\}$ \hspace{2cm} \triangleright p is an HHH candidate
16:                print($\langle p, \hat{f}_p^-, \hat{f}_p^+ \rangle$)
17:            end if
18:        end for
19:    end for
20:    return $P$
21: end function

Pseudocode of RHHH is given in Algorithm[7] RHHH uses the same algorithm for both one and two dimensions. The differences between them are manifested in the calcPred method. Pseudocode of this method is found in Algorithm[8] for one dimension and in Algorithm[9] for two dimensions.

Definition The underlying estimation provides us with upper and lower estimates for the number of times prefix $p$ was updated ($X_p$). We denote: $\hat{X}_p^+$ to be an upper bound for $X_p$ and $\hat{X}_p^-$ to be a lower bound. For simplicity of notations, we define the following:

$\hat{f}_p \triangleq \hat{X}_p V$ – an estimator for $p$’s frequency.

$\hat{f}_p^+ \triangleq \hat{X}_p^+ V$ – an upper bound for $p$’s frequency.

$\hat{f}_p^- \triangleq \hat{X}_p^- V$ – a lower bound for $p$’s frequency.

Note these bounds ignore the sample error that is accounted separately in the analysis.

The output method of RHHH starts with fully specified items and if their frequency is above $\theta N$, it adds them to $P$. Then, RHHH iterates over their parent items and calculates a conservative estimation of their conditioned frequency with respect to $P$. Conditioned frequency is calculated by an upper estimate to ($f_p^+$) amended by the output
Algorithm 8 calcPred for one dimension
1: function CALCPRED(prefix p, set P)
2: \[ R = 0 \]
3: for each \( h \in G(p|P) \) do
4: \[ R = R - \hat{f}_h^- \]
5: end for
6: return \( R \)
7: end function

Algorithm 9 calcPred for two dimensions
1: function CALCPRED(prefix p, set P)
2: \[ R = 0 \]
3: for each \( h \in G(p|P) \) do
4: \[ R = R - \hat{f}_h^- \]
5: end for
6: for each pair \( h, h' \in G(p|P) \) do
7: \[ q = \text{glb}(h, h') \]
8: if \( \exists h_3 \neq h, h' \in G(p|P), q \preceq h_3 \) then
9: \[ R = R + \hat{f}_q^+ \]
10: end if
11: end for
12: return \( R \)
13: end function

of the calcPred method. In a single dimension, we reduce the lower bounds of \( p \)'s closest predecessor \( \text{HHH} \)'s. In two dimensions, we use inclusion and exclusion principles to avoid double counting. In addition, Algorithm 9 uses the notation of \textit{greater lower bound (glb)} that is formally defined in Definition 6.4.2. Finally, we add a constant to the conditioned frequency to account for the sampling error.

Definition Denote \( \text{glb}(h, h') \) the greatest lower bound of \( h \) and \( h' \). \( \text{glb}(h, h') \) is a unique common descendant of \( h \) and \( h' \) s.t. \( \forall p : (q \preceq p) \land (p \preceq h) \land (p \preceq h') \Rightarrow p = q \). When \( h \) and \( h' \) have no common descendants, define \( \text{glb}(h, h') \) as an item with count 0.

In two dimensions, \( C_{p|P} \) is first set to be the upper bound on \( p \)'s frequency (Line 12 Algorithm 7). Then, we remove previously selected descendant heavy hitters (Line 4 Algorithm 9). Finally, we add back the common descendant (Line 9 Algorithm 9).

Note that the work of [38] showed that their structure extends to higher dimensions, with only a slight modification to the Output method to ensure that it conservatively estimates the conditioned count of each prefix. As we use the same general structure, their extension applies in our case as well.
Figure 6.3: Accuracy error ratio – HHH candidates whose frequency estimation error is larger than $N\epsilon$ ($\epsilon = 0.001$).

Figure 6.4: The percentage of Coverage errors – elements $q$ such that $q \notin P$ and $C_{q,P} \geq N\theta$ (false negatives).

6.5 Evaluation

Our evaluation includes MST [38], the Partial and Full Ancestry [40] algorithms and two configurations of RHHH, one with $V = H$ (RHHH) and the other with $V = 10 \cdot H$ (10-RHHH). RHHH performs a single update operation per packet while 10-RHHH performs such an operation only for 10% of the packets. Thus, 10-RHHH is considerably faster than RHHH but requires more traffic to converge.

The evaluation was performed on a single Dell 730 server running Ubuntu 16.04.01 release. The server has 128GB of RAM and an Intel(R) Xeon(R) CPU E5-2667 v4 @ 3.20GHz processor.

Our evaluation includes four datasets, each containing a mix of 1 billion UDP/TCP and ICMP packets collected from major backbone routers in both Chicago [86, 87] and San Jose [88, 70] during the years 2014-2016. We considered source hierarchies in byte (1D Bytes) and bit (1D Bits) granularities, as well as a source/destination byte hierarchy (2D Bytes). Such hierarchies were also used by [38, 40]. We ran each data point 5 times and used two-sided Student’s t-test to determine 95% confidence intervals.

6.5.1 Accuracy and Coverage Errors

RHHH has a small probability of both accuracy and coverage errors that are not present in previous algorithms. Figure 6.3 quantifies the accuracy errors and Figure 6.4 quantifies the coverage errors. As can be seen, RHHH becomes more accurate as the trace progresses.
Figure 6.5: False Positive Rate for different stream lengths.

Figure 6.6: Update speed comparison for different hierarchical structures and workloads.

Our theoretic bound ($\psi$ as derived in Section 6.7 below) for these parameters is about 100 million packets for RHHH and about 1 billion packets for 10-RHHH. Indeed, these algorithms converge once they reach their theoretical bounds (see Theorem 6.7.17).
6.5.2 False Positives

Approximate HHH algorithms find all the HHH prefixes but they also return non HHH prefixes. False positives measure the ratio non HHH prefixes pose out of the returned HHH set. Figure 6.5 shows a comparative measurement of false positive ratios in the Chicago 16 and San Jose 14 traces. Every point was measured for $\epsilon = 0.1\%$ and $\theta = 1\%$. As shown, for RHHH and 10-RHHH the false positive ratio is reduced as the trace progresses. Once the algorithms reach their theoretic grantees ($\psi$), the false positives are comparable to these of previous works. In some cases, RHHH and 10-RHHH even perform slightly better than the alternatives.

6.5.3 Operation Speed

Figure 6.6 shows a comparative evaluation of operation speed. Figure 6.6(a), Figure 6.6(b) and Figure 6.6(c) show the results of the San Jose 14 trace for 1D byte hierarchy ($H = 5$), 1D bit hierarchy ($H = 33$) and 2D byte hierarchy ($H = 25$), respectively. Similarly, Figure 6.6(d), Figure 6.6(e) and Figure 6.6(f) show results for the Chicago 16 trace on the same hierarchical domains. Each point is computed for $250M$ long packet traces. Clearly, the performance of RHHH and 10-RHHH is relatively similar for a wide range of $\epsilon$ values and for different data sets. Existing works depend on $H$ and indeed run considerably slower for large $H$ values.

Another interesting observation is that the Partial and Full Ancestry [40] algorithms improve when $\epsilon$ is small. This is because in that case there are few replacements in their trie based structure, as is directly evident by their $O(H \log(N\epsilon))$ update time, which is decreasing with $\epsilon$. However, the effect is significantly lessened when $H$ is large.

RHHH and 10-RHHH achieve speedup for a wide range of $\epsilon$ values, while 10-RHHH is the fastest algorithm overall. For one dimensional byte level hierarchies, the achieved speedup is up to X3.5 for RHHH and up to X10 for 10-RHHH. For one dimensional bit level hierarchies, the achieved speedup is up to X21 for RHHH and up to X62 for 10-RHHH. Finally, for 2 dimensional byte hierarchies, the achieved speedup is up to X20 for RHHH and up to X60 for 10-RHHH. Evaluation on Chicago15 and SanJose13 yielded similar results, which are omitted due to lack of space.

6.6 Virtual Switch Integration

This section describes how we extended Open vSwitch (OVS) to include approximate HHH monitoring capabilities. For completeness, we start with a short overview of OVS and then continue with our evaluation.
6.6.1 Open vSwitch Overview

Virtual switching is a key building block in NFV environments, as it enables interconnecting multiple Virtual Network Functions (VNFs) in service chains and enables the use of other routing technologies such as SDN. In practice, virtual switches rely on sophisticated optimizations to cope with the line rate.

Specifically, we target the DPDK version of OVS that enables the entire packet processing to be performed in user space. It mitigates overheads such as interrupts required to move from user space to kernel space. In addition, DPDK enables user space packet processing and provides direct access to NIC buffers without unnecessary memory copy. The DPDK library received significant engagement from the NFV industry [98].

The architectural design of OVS is composed of two main components: ovs-vswitchd and ovsdb-server. Due to space constraints, we only describe the vswitchd component. The interested reader is referred to [99] for additional information. The DPDK-version of the vswitchd module implements control and data planes in user space. Network packets ingress the datapath (dpif or dpif-netdev) either from a physical port connected to the physical NIC or from a virtual port connected to a remote host (e.g., a VNF). The datapath then parses the headers and determines the set of actions to be applied (e.g., forwarding or rewrite a specific header).

6.6.2 Open vSwitch Evaluation

We examined two integration methods: First, HHH measurement can be performed as part of the OVS dataplane. That is, OVS updates each packet as part of its processing stage. Second, HHH measurement can be performed in a separate virtual machine. In that case, OVS forwards the relevant traffic to the virtual machine. When RHHH operates with $V > H$, we only forward the sampled packets and thus reduce overheads.

OVS Environment Setup

Our evaluation settings consist of two identical HP ProLiant servers with an Intel Xeon E3-1220v2 processor running at 3.1 Ghz with 8 GB RAM, an Intel 82599ES 10 Gbit/s network card and CentOS 7.2.1511 with Linux kernel 3.10.0 operating system. The servers are directly connected through two physical interfaces. We used Open vSwitch 2.5 with Intel DPDK 2.02, where NIC physical ports are attached using dpdk ports.

One server is used as traffic generator while the other is used as Design Under Test (DUT). Placed on the DUT, OVS receives packets on one network interface and then forwards them to the second one. Traffic is generated using MoonGen traffic generator [100], and we generate 1 billion UDP packets but preserve the source and destination IP as in the original dataset. We also adjust the payload size to 64 bytes and reach 14.88 million packets per second (Mpps).
Figure 6.7: Throughput of dataplane implementations ($\varepsilon = 0.001, \delta = 0.001, 2$D Bytes, Chicago 16).

**OVS Throughput Evaluation**

Figure 6.7 exhibits the throughput of OVS for dataplane implementations. It includes our own 10-RHHH (with $V = 10H$) and RHHH (with $V = H$), as well as MST and Partial Ancestry. Since we only have 10 Gbit/s links, the maximum achievable packet rate is 14.88 Mpps.

As can be seen, 10-RHHH processes 13.8 Mpps, only 4% lower than unmodified OVS. RHHH achieves 10.6 Mpps, while the fastest competition is Partial Ancestry that delivers 5.6 Mpps. Note that a 100 Gbit/s link delivering packets whose average size is 1KB only delivers $\approx 8.33$ Mpps. Thus, 10-RHHH and RHHH can cope with the line speed.

In Figure 6.8, we evaluate the throughput for different $V$ values, from $V = H = 25$ (RHHH) to $V = 10 \cdot H = 250$ (10-RHHH). Figure 6.8(a) evaluates the dataplane implementation while Figure 6.8(b) evaluates the distributed implementation. In both figures, performance improves for larger $V$ value. In the distributed implementation, this speedup means that fewer packets are forwarded to the VM whereas in the dataplane implementation, it is linked to fewer processed packets.

Note that while the distributed implementation is somewhat slower, it enables the measurement machine to process traffic from multiple sources.
6.7 Analysis

This section aims to prove that RHHH solves the \((\delta, \epsilon, \theta)\)-APPROXIMATE HHH problem (Definition 6.4.1) for one and two dimensional hierarchies. Toward that end, Section 6.7.1 proves the accuracy requirement while Section 6.7.2 proves coverage. Section 6.7.3 proves that RHHH solves the \((\delta, \epsilon, \theta)\)-APPROXIMATE HHH problem as well as its memory and update complexity.

We model the update procedure of RHHH as a balls and bins experiment where there are \(V\) bins and \(N\) balls. Prior to each packet arrival, we place the ball in a bin that is selected uniformly at random. The first \(H\) bins contain an HH update action while the next \(V-H\) bins are void. When a ball is assigned to a bin, we either update the underlying HH algorithm with a prefix obtained from the packet’s headers or ignore the packet if the bin is void. Our first goal is to derive confidence intervals around the number of balls in a bin.

**Definition** We define \(X^K_i\) to be the random variable representing the number of balls from set \(K\) in bin \(i\), e.g., \(K\) can be all packets that share a certain prefix, or a combination of multiple prefixes with a certain characteristic. When the set \(K\) contains all packets, we use the notation \(X_i\).

Random variables representing the number of balls in a bin are dependent on each other. Therefore, we cannot apply common methods to create confidence intervals. Formally, the dependence is manifested as:
\[
\sum_i^V X_i = N.
\]
This means that the number of balls in a certain bin is determined by the number of balls in all other bins.

Our approach is to approximate the balls and bins experiment with the corresponding Poisson one. That is, analyze the Poisson case and derive confidence intervals and then use Lemma 6.7.1 to derive a (weaker) result for the original balls and bins case.
We now formally define the corresponding Poisson model. Let $Y^K_1, ..., Y^K_V$ s.t. $\{ Y^K_i \} \sim Poisson \left( \frac{K}{V} \right)$ be independent Poisson random variables representing the number of balls in each bin from a set of balls $K$. That is: $\{ Y^K_i \} \sim Poisson \left( \frac{K}{V} \right)$.

**Lemma 6.7.1 (Corollary 5.11, page 103 of [101]).** Let $\mathcal{E}$ be an event whose probability is either monotonically increasing or decreasing with the number of balls. If $\mathcal{E}$ has probability $p$ in the Poisson case then $\mathcal{E}$ has probability at most $2p$ in the exact case.

### 6.7.1 Accuracy Analysis

We now tackle the accuracy requirement from Definition 6.4.1. That is, for every HHH prefix ($p$), we need to prove:

$$\Pr \left( \left| f_p - \hat{f}_p \right| \leq \varepsilon N \right) \geq 1 - \delta.$$

In RHHH, there are two distinct origins of error. Some of the error comes from fluctuations in the number of balls per bin while the approximate HH algorithm is another source of error.

We start by quantifying the balls and bins error. Let $Y^p_i$ be the Poisson variable corresponding to prefix $p$. That is, the set $p$ contains all packets that are generalized by prefix $p$. Recall that $f_p$ is the number of packets generalized by $p$ and therefore: $E(Y^p_i) = \frac{f_p}{V}$.

We need to show that with probability $1 - \delta_s$, $Y^p_i$ is within $\varepsilon_s N$ from $E(Y^p_i)$. Fortunately, confidence intervals for Poisson variables are well studied [102] and we use the method of [103] that is quoted in Lemma 6.7.2.

**Lemma 6.7.2.** Let $X$ be a Poisson random variable, then

$$\Pr \left( \left| X - E(X) \right| \geq Z_{1-\delta} \sqrt{E(X)} \right) \leq \delta,$$

where $Z_\alpha$ is the $z$ value that satisfies $\phi(z) = \alpha$ and $\phi(z)$ is the density function of the normal distribution with mean 0 and standard deviation of 1.

Lemma 6.7.2 provides us with a confidence interval for Poisson variables, and enables us to tackle the main accuracy result.

**Theorem 6.7.3.** If $N \geq Z_{1-\frac{\delta_s}{2}} V \varepsilon_s^{-2}$ then

$$\Pr \left( \left| Y^p_i H - f_p \right| \geq \varepsilon_s N \right) \leq \delta_s.$$
Proof. We use Lemma 6.7.2 for $\frac{\delta_s^2}{2}$ and get:

$$\Pr \left( \left| Y_i^p - \frac{f_p}{V} \right| \geq Z_{1 - \frac{\delta_s^2}{2}} \sqrt{\frac{f_p}{V}} \right) \leq \frac{\delta_s^2}{2}. $$

To make this useful, we trivially bind $f_p \leq N$ and get

$$\Pr \left( \left| Y_i^p - \frac{f_p}{V} \right| \geq Z_{1 - \frac{\delta_s^2}{2}} \sqrt{\frac{N}{V}} \right) \leq \frac{\delta_s^2}{2}. $$

However, we require error of the form $\frac{\varepsilon_s N}{V}$.

$$\varepsilon_s N V^{-1} \geq Z_{1 - \frac{\delta_s^2}{2}} V^{-0.5} N^{0.5}$$

$$N^{0.5} \geq Z_{1 - \frac{\delta_s^2}{2}} V^{0.5} \varepsilon_s^{-1}$$

$$N \geq Z_{1 - \frac{\delta_s^2}{2}} V \varepsilon_s^{-2}.$$  

Therefore, when $N \geq Z_{1 - \frac{\delta_s^2}{2}} V \varepsilon_s^{-2}$, we have that:

$$\Pr \left( \left| Y_i^p - \frac{f_p}{V} \right| \geq \frac{\varepsilon_s N}{V} \right) \leq \frac{\delta_s^2}{2}. $$

We multiply by $V$ and get:

$$\Pr \left( \left| Y_i^p V - f_p \right| \geq \varepsilon_s N \right) \leq \frac{\delta_s^2}{2}. $$

Finally, since $Y_i^p$ is monotonically increasing with the number of balls ($f_p$), we apply Lemma 6.7.1 to conclude that

$$\Pr \left( \left| X_i^p V - f_p \right| \geq \varepsilon_s N \right) \leq \delta_s. $$

To reduce clutter, we denote $\psi \triangleq Z_{1 - \frac{\delta_s^2}{2}} V \varepsilon_s^{-2}$. Theorem 6.7.3 proves that the desired sample accuracy is achieved once $N > \psi$.

It is sometimes useful to know what happens when $N < \psi$. For this case, we have Corollary 6.7.4 which is easily derived from Theorem 6.7.3. We use the notation $\varepsilon_s(N)$ to define the actual sampling error after $N$ packets. Thus, it assures us that when $N < \psi$, $\varepsilon_s(N) > \varepsilon_s$. It also shows that $\varepsilon_s(N) < \varepsilon_s$ when $N > \psi$. Another application of Corollary 6.7.4 is that given a measurement interval $N$, we can derive a value for $\varepsilon_s$ that assures correctness. For simplicity, we continue with the notion of $\varepsilon_s$.

**Corollary 6.7.4.** $\varepsilon_s(N) \geq \sqrt{\frac{Z_{1 - \frac{\delta_s^2}{2}} V}{N}}$.

The error of approximate HH algorithms is proportional to the number of updates. Therefore, our next step is to provide a bound on the number of updates of an arbitrary HH algorithm. Given such a bound, we configure the algorithm to compensate so that the accumulated error remains within the guarantee even if the number of updates is larger than average.

100

Technion - Computer Science Department - Ph.D. Thesis PHD-2018-08 - 2018
Corollary 6.7.5. Consider the number of updates for a certain lattice node \((X_i)\). If \(N > \psi\), then

\[
\Pr\left( X_i \leq \frac{N}{V} (1 + \varepsilon_s) \right) \geq 1 - \delta_s.
\]

Proof. We use Theorem 6.7.3 and get:
\[
\Pr \left( \left| X_i - \frac{N}{V} \right| \geq \varepsilon_s N \right) \leq \delta_s.
\]
This implies that:
\[
\Pr \left( X_i \leq \frac{N}{V} (1 + \varepsilon_s) \right) \geq 1 - \delta_s,
\]
completing the proof.

We explain now how to configure our algorithm to defend against situations in which a given approximate HH algorithm might get too many updates, a phenomenon we call over sample. Corollary 6.7.5 bounds the probability for such an occurrence, and hence we can slightly increase the accuracy so that in the case of an over sample, we are still within the desired limit. We use an algorithm (A) that solves the \((\varepsilon_a, \delta_a)\) - FREQUENCY ESTIMATION problem. We define \(\varepsilon'_a = \frac{\varepsilon_a}{1 + \varepsilon_s}\). According to Corollary 6.7.5, with probability \(1 - \delta_s\), the number of sampled packets is at most \((1 + \varepsilon_s) \frac{N}{V}\). By using the union bound and with probability \(1 - \delta_a - \delta_s\) we get:

\[
\left| X^p - \hat{X}^p \right| \leq \varepsilon'_a (1 + \varepsilon_s) \frac{N}{V} = \varepsilon_a \frac{(1 + \varepsilon_s) N}{1 + \varepsilon_s} = \varepsilon_a \frac{N}{V}.
\]

For example, Space Saving requires 1,000 counters for \(\varepsilon_a = 0.001\). If we set \(\varepsilon_s = 0.001\), we now require 1001 counters. Hereafter, we assume that the algorithm is configured to accommodate these over samples.

Theorem 6.7.6. Consider an algorithm (A) that solves the \((\varepsilon_a, \delta_a)\) - FREQUENCY ESTIMATION problem. If \(N > \psi\), then for \(\delta \geq \delta_a + 2 \cdot \delta_s\) and \(\epsilon \geq \epsilon_a + \epsilon_s\), A solves \((\epsilon, \delta)\) - FREQUENCY ESTIMATION.

Proof. As \(N > \psi\), we use Theorem 6.7.3. That is, the input solves \((\epsilon, \delta)\) - FREQUENCY ESTIMATION.

\[
\Pr \left[ \left| f_p - X^p V \right| \geq \varepsilon_s N \right] \leq \delta_s.
\] (6.1)

A solves the \((\varepsilon_a, \delta_a)\) - FREQUENCY ESTIMATION problem and provides us with an estimator \(\hat{X}^p\) that approximates \(X^p\) – the number of updates for prefix \(p\). According to Corollary 6.7.5

\[
\Pr \left( \left| X^p - \hat{X}^p \right| \leq \frac{\varepsilon_a N}{V} \right) \geq 1 - \delta_a - \delta_s,
\]
and multiplying both sides by \(V\) gives us:

\[
\Pr \left( \left| X^p V - \hat{X}^p V \right| \geq \varepsilon_a N \right) \leq \delta_a + \delta_s.
\] (6.2)

We need to prove that: \(\Pr \left( \left| f_p - \hat{X}^p V \right| \leq \varepsilon N \right) \geq 1 - \delta\). Recall that: \(f_p = E(X^p) V\) and that \(\hat{f}_p = \hat{X}^p V\) is the estimated frequency of \(p\). Thus,

\[
\Pr \left( \left| f_p - \hat{f}_p \right| \geq \varepsilon N \right) = \Pr \left( \left| f_p - \hat{X}^p V \right| \geq \varepsilon N \right)
\]
\[
= \Pr \left( \left| f_p + (X^p V - \hat{X}^p V) - V \hat{X}^p \right| \geq (\varepsilon_a + \varepsilon_s) N \right)
\]
\[
\leq \Pr \left( \left| f_p - X^p V \right| \geq \varepsilon_s N \right) \lor \left[ \left| X^p V - \hat{X}^p V \right| \geq \varepsilon_a N \right],
\]
101
Technion - Computer Science Department - Ph.D. Thesis PHD-2018-08 - 2018
where the last inequality follows from the fact that in order for the error of (6.3) to exceed $\epsilon N$, at least one of the events has to occur. We bound this expression using the Union bound.

$$\Pr \left( \left| f_p - \hat{f}_p \right| \geq \epsilon N \right) \leq \Pr \left( |f_p - X_pV| \geq \epsilon_s N \right) + \Pr \left( |X_pV - \hat{X}_p H| \geq \epsilon_a N \right) \leq \delta_a + 2\delta_s,$$

where the last inequality is due to equations 6.1 and 6.2.

An immediate observation is that Theorem 6.7.6 implies accuracy, as it guarantees that with probability $1 - \delta$ the estimated frequency of any prefix is within $\epsilon N$ of the real frequency while the accuracy requirement only requires it for prefixes that are selected as HHH.

**Lemma 6.7.7.** If $N > \psi$, then Algorithm $\mathcal{A}$ satisfies the accuracy constraint for $\delta = \delta_a + 2\delta_s$ and $\epsilon = \epsilon_a + \epsilon_s$.

**Proof.** The proof follows from Theorem 6.7.6 as the frequency estimation of a prefix depends on a single HH algorithm.

**Multiple Updates**

One might consider how RHHH behaves if instead of updating at most 1 HH instance, we update $r$ independent instances. This implies that we may update the same instance more than once per packet. Such an extension is easy to do and still provides the required guarantees. Intuitively, this variant of the algorithm is what one would get if each packet is duplicated $r$ times. The following corollary shows that this makes RHHH converge $r$ times faster.

**Corollary 6.7.8.** Consider an algorithm similar to RHHH with $V = H$, but for each packet we perform $r$ independent update operations. If $N > \psi r$, then this algorithm satisfies the accuracy constraint for $\delta = \delta_a + 2\delta_s$ and $\epsilon = \epsilon_a + \epsilon_s$.

**Proof.** Observe that the new algorithm is identical to running RHHH on a stream ($S'$) where each packet in $S$ is replaced by $r$ consecutive packets. Thus, Lemma 6.7.7 guarantees that accuracy is achieved for $S'$ after $\psi$ packets are processed. That is, it is achieved for the original stream ($S$) after $N > \frac{\psi}{r}$ packets.

6.7.2 Coverage Analysis

Our goal is to prove the coverage property of Definition 6.4.1. That is:

$$\Pr \left( \hat{C}_{q|P} \geq C_{q|P} \right) \geq 1 - \delta.$$ Conditioned frequencies are calculated in a different manner for one and two dimensions. Thus, Section 6.7.2 deals with one dimension and Section 6.7.2 with two.

We now present a common definition of the best generalized prefixes in a set.
**Best generalization** Define \( G(q|P) \) as the set \( \{ p : p \in P, p \prec q, \neg \exists p' \in P : q \prec p' \prec p \} \). Intuitively, \( G(q|P) \) is the set of prefixes that are best generalized by \( q \). That is, \( q \) does not generalize any prefix that generalizes one of the prefixes in \( G(q|P) \).

**One Dimension**

We use the following lemma for bounding the error of our conditioned count estimates.

**Lemma 6.7.9.** ([38]) In one dimension,

\[
C_{q|P} = \hat{f}_q - \sum_{h \in G(q|P)} \hat{f}_h.
\]

Using Lemma 6.7.9 it is easier to establish that the conditioned frequency estimates calculated by Algorithm 7 are conservative.

**Lemma 6.7.10.** The conditioned frequency estimation of Algorithm 7 is:

\[
\hat{C}_{q|P} = \hat{f}_q^+ - \sum_{h \in G(q|P)} \hat{f}_h^- + 2Z_{1-\delta}\sqrt{NV}.
\]

**Proof.** Looking at Line 12 in Algorithm 7 we get that:

\[
\hat{C}_{q|P} = \hat{f}_q^+ + \text{calcPred}(q,P).
\]

That is, we need to verify that the return value \( \text{calcPred}(q,P) \) in one dimension (Algorithm 8) is \( \sum_{h \in G(q|P)} \hat{f}_h^- \). This follows naturally from that algorithm. Finally, the addition of \( 2Z_{1-\delta}\sqrt{NV} \) is due to line 13.

In deterministic settings, \( \hat{f}_q^+ - \sum_{h \in G(q|P)} \hat{f}_h^- \) is a conservative estimate since \( \hat{f}_q^+ \geq f_q \) and \( f_h < \hat{f}_h^- \). In our case, these are only true with regard to the sampled sub-stream and the addition of \( 2Z_{1-\delta}\sqrt{NV} \) is intended to compensate for the randomized process.

Our goal is to show that \( \Pr(\hat{C}_{q|P} > C_{q|P}) \geq 1 - \delta \). That is, the conditioned frequency estimation of Algorithm 7 is probabilistically conservative.

**Theorem 6.7.11.** \( \Pr(\hat{C}_{q|P} \geq C_{q|P}) \geq 1 - \delta \).

**Proof.** Recall that:

\[
\hat{C}_{q|P} = \hat{f}_q^+ - \sum_{h \in G(q|P)} \hat{f}_h^- + 2Z_{1-\delta}\sqrt{NV}.
\]

We denote by \( K \) the set of packets that may affect \( \hat{C}_{q|P} \). We split \( K \) into two sets: \( K^+ \) contains the packets that may positively impact \( \hat{C}_{q|P} \) and \( K^- \) contains the packets that may negatively impact it.

We use \( K^+ \) to estimate the sample error in \( \hat{f}_q \) and \( K^- \) to estimate the sample error in \( \sum_{h \in G(q|P)} \hat{f}_h^- \). The positive part is easy to estimate. In the negative, we do not know exactly how many bins affect the sum. However, we know for sure that there are at most \( N \). We define the random variable \( Y^+_K \) that indicates the number of balls included in the positive
sum. We invoke Lemma 6.7.2 on $Y^+_K$. For the negative part, the conditioned frequency is positive so $E(Y^-_K)$ is at most $\frac{N}{V}$. Hence, $\Pr\left(\left|Y^+_K - E(Y^+_K)\right| \geq Z_1 \sqrt{\frac{N}{V}}\right) \leq \frac{\delta}{4}$. Similarly, we use Lemma 6.7.2 to bound the error of $Y^-_K$:

$$\Pr\left(\left|Y^-_K - E(Y^-_K)\right| \geq Z_1 \sqrt{\frac{N}{V}}\right) \leq \frac{\delta}{4}.$$  

$Y^+_K$ is monotonically increasing with any ball and $Y^-_K$ is monotonically decreasing with any ball. Therefore, we can apply Lemma 6.7.1 on each of them and conclude:

$$\Pr\left(C_{q|P} \geq C_{q|P}\right) \leq 2 \Pr\left(H(Y^-_K + Y^+_K) \geq V E(Y^-_K + Y^+_K) + 2Z_1 \sqrt{N V}\right) \leq 1 - 2\frac{\delta}{2} = 1 - \delta.$$

**Theorem 6.7.12.** If $N > \psi$, Algorithm 7 solves the $(\delta,\varepsilon,\theta)$-APPROXIMATE HHH problem for $\delta = \delta_a + 2\delta_s$ and $\varepsilon = \varepsilon_s + \varepsilon_a$.

**Proof.** We need to show that the accuracy and coverage guarantees hold. Accuracy follows from Lemma 6.7.7 and coverage follows from Theorem 6.7.11 that implies that for every non heavy hitter prefix $(q)$, $C_{q|P} < \theta N$ and thus:

$$\Pr\left(C_{q|P} < \theta N\right) \geq 1 - \delta.$$  

**Two Dimensions**

Conditioned frequency is calculated differently for two dimensions, as we use inclusion/exclusion principles and we need to show that these calculations are sound too. We start by stating the following lemma:

**Lemma 6.7.13.** (38) In two dimensions,

$$C_{q|P} = f_q - \sum_{h \in G(q|P)} f_h + \sum_{h, h' \in G(q|P)} f_{glb(h,h')}.$$  

In contrast, Algorithm 7 estimates the conditioned frequency as:

**Lemma 6.7.14.** In two dimensions, Algorithm 7 calculates conditioned frequency in the following manner:

$$\widehat{C_{q|P}} = \widehat{f_q} - \sum_{h \in G(q|P)} \widehat{f_h} + \sum_{h, h' \in G(q|P)} \widehat{f_{glb(h,h')}} + 2Z_1 \sqrt{N V}.$$  

**Proof.** The proof follows from Algorithm 7. Line 12 is responsible for the first element $\widehat{f_q}$ while Line 13 is responsible for the last element. The rest is due to the function calcPredecessors in Algorithm 9. 

104
Theorem 6.7.15. \( \Pr \left( C_{q|P} \geq C_{q|\hat{P}} \right) \geq 1 - \delta. \)

Proof. Observe Lemma 6.7.13 and notice that in deterministic settings, as shown in [38],
\[ \hat{f}_q + \sum_{h \in G(q|P)} \hat{f}_h + \sum_{h, h' \in G(q|P)} \hat{f}_{glb(h, h')} \]
is a conservative estimate for \( C_{q|P}. \) Therefore, we need to account for the randomization error and verify that with probability \( 1 - \delta \) it is less than \( 2 \left( \frac{1}{8} \sqrt{N V} \right) \).

We denote by \( K \) the packets that may affect \( C_{q|P}. \) Since the expression of \( \hat{C}_{q|P} \) is not monotonic, we split it into two sets: \( K^+ \) are packets that affect \( C_{q|P} \) positively and \( K^- \) affect it negatively. Similarly, we define \( \{ Y^+_K \} \) to be Poisson random variables that represent how many of the packets of \( K \) are in each bin.

We do not know how many bins affect the sum, but we know for sure that there are no more than \( \tilde{N} \) balls. We define the random variable \( Y^+_K \) that defines the number of packets from \( K \) that fell in the corresponding bins to have a positive impact on \( \hat{C}_{q|P}. \) Invoking Lemma 6.7.2 on \( Y^+_K \) yields that:
\[ \Pr \left( |Y^+_K - E(Y^+_K)| \geq Z_{1-\frac{\delta}{8}} \sqrt{V} \right) \leq \frac{\delta}{4}. \]

Similarly, we define \( Y^-_K \) to be the number of packets from \( K \) that fell into the corresponding buckets to create a negative impact on \( \hat{C}_{q|P} \) and Lemma 6.7.2 results in:
\[ \Pr \left( |Y^-_K - E(Y^-_K)| \geq Z_{1-\frac{\delta}{8}} \sqrt{V} \right) \leq \frac{\delta}{4}. \]

\( Y^+_K \) is monotonically increasing with the number of balls and \( Y^-_K \) is monotonically decreasing with the number of balls. We can apply Lemma 6.7.1 and conclude that:
\[
\begin{align*}
\Pr \left( \hat{C}_{q|P} \geq C_{q|P} \right) & \leq 2 \Pr \left( V \left(Y^-_K + Y^+_K\right) \geq (VE(Y^-_K + Y^+_K) + 2Z_{1-\frac{\delta}{8}}\sqrt{NV}) \right) \\
& \leq 1 - 2\frac{\delta}{2} = 1 - \delta,
\end{align*}
\]
completing the proof.

Putting It All Together

We can now prove the coverage property for one and two dimensions.

Corollary 6.7.16. If \( N > \psi \) then RHHH satisfies coverage. That is, given a prefix \( q \notin P, \) where \( P \) is the set of HHH returned by RHHH,
\[ \Pr \left( C_{q|P} < \theta N \right) > 1 - \delta. \]

Proof. The proof follows form Theorem 6.7.11 in one dimension, or Theorem 6.7.15 in two, that guarantee that in both cases: \( \Pr \left( C_{q|P} < \hat{C}_{q|P} \right) > 1 - \delta. \)

The only case where \( q \notin P \) is if \( \hat{C}_{q|P} < \theta N. \) Otherwise, Algorithm 7 would have added it to \( P. \) However, with probability \( 1 - \delta, \) \( C_{q|P} < \hat{C}_{q|P}, \) and therefore \( C_{q|P} < \theta N \) as well.
6.7.3 RHHH Properties Analysis

Finally, we can prove the main result of our analysis. It establishes that if the number of packets is large enough, RHHH is correct.

**Theorem 6.7.17.** If $N > \psi$, then RHHH solves $(\delta, \epsilon, \theta)$ - Approximate Hierarchical Heavy Hitters.

**Proof.** The theorem is proved by combining Lemma 6.7.7 and Corollary 6.7.16.

Note that $\psi \triangleq Z_{1-\frac{\delta}{2}} V \varepsilon^2$ contains the parameter $V$ in it. When the minimal measurement interval is known in advance, the parameter $V$ can be set to satisfy correctness at the end of the measurement. For short measurements, we may need to use $V = H$, while longer measurements justify using $V \gg H$ and achieve better performance. When considering modern line speed and emerging new transmission technologies, this speedup capability is crucial because faster lines deliver more packets in a given amount of time and thus justify a larger value of $V$ for the same measurement interval.

For completeness, we prove the following.

**Theorem 6.7.18.** RHHH’s update complexity is $O(1)$.

**Proof.** Observe Algorithm 7. For each update, we randomize a number between 0 and $V - 1$, which can be done in $O(1)$. Then, if the number is smaller than $H$, we also update a Space Saving instance, which can be done in $O(1)$ as well [30].

Finally, we note that our space requirement is similar to that of [38].

**Theorem 6.7.19.** The space complexity of RHHH is $O \left( \frac{H}{\epsilon a} \right)$ flow table entries.

**Proof.** RHHH utilizes $H$ separate instances of Space Saving, each using $\frac{1}{\epsilon a}$ table entries. There are no other space significant data structures.

6.8 Discussion

This work is about realizing hierarchical heavy hitters measurement in virtual network devices. Existing HHH algorithms are too slow to cope with current improvements in network technology. Therefore, we define a probabilistic relaxation of the problem and introduce a matching randomized algorithm called RHHH. Our algorithm leverages the massive traffic in modern networks to perform simpler update operations. Intuitively, the algorithm replaces the traditional approach of computing all prefixes for each incoming packets by sampling (if $V > H$) and then choosing one random prefix to be updated. While similar convergence guarantees can be derived for the simpler approach of updating all prefixes for each sampled packet, our solution has the clear advantage of processing elements in $O(1)$ worst case time.
We evaluated RHHH on four real Internet packet traces, consisting over 1 billion packets each and achieved a speedup of up to X62 compared to previous works. Additionally, we showed that the solution quality of RHHH is comparable to that of previous work. RHHH performs updates in constant time, an asymptotic improvement from previous works whose complexity is proportional to the hierarchy’s size. This is especially important in the two dimensional case as well as for IPv6 traffic that requires larger hierarchies.

Finally, we integrated RHHH into a DPDK enabled Open vSwitch and evaluated its performance as well as the alternative algorithms. We provided a dataplane implementation where HHH measurement is performed as part of the per packet routing tasks. In a dataplane implementation, RHHH is capable of handling up to 13.8 Mpps, 4% less than an unmodified DPDK OVS (that does not perform HHH measurement). We showed a throughput improvement of X2.5 compared to the fastest dataplane implementations of previous works.

Alternatively, we evaluated a distributed implementation where RHHH is realized in a virtual machine that can be deployed in the cloud and the virtual switch only sends the sampled traffic to RHHH. Our distributed implementation can process up to 12.3 Mpps. It is less intrusive to the switch, and offers greater flexibility in virtual machine placement. Most importantly, our distributed implementation is capable of analyzing data from multiple network devices.

Notice the performance improvement gap between our direct implementation – X62, compared to the performance improvement when running over OVS – X2.5. In the case of the OVS experiments, we were running over a 10Gbps link, and were bound by that line speed – the throughput obtained by our implementation was only 4% lower than the unmodified OVS baseline (that does nothing). In contrast, previous works were clearly bounded by their computational overhead. Thus, one can anticipate that once we deploy the OVS implementation on faster links, or in a setting that combines traffic from multiple links, the performance boost compared to previous work will be closer to the improvement we obtained in the direct implementation.

A downside of RHHH is that it requires some minimal number of packets in order to converge to the desired formal accuracy guarantees. In practice, this is a minor limitation as busy links deliver many millions of packets every second. For example, in the settings reported in Section 6.3.1, RHHH requires up to 100 millions packets to fully converge, yet even after as little as 8 millions packets, the error reduces to around 1%. With a modern switch that can serve 10 million packets per second, this translates into a 10 seconds delay for complete convergence and around 1% error after 1 second. As line rates will continue to improve, these delays would become even shorter accordingly. The code used in this work is open sourced [104].
Chapter 7

Additional Sliding Window Measurements

For many networking applications, recent data is more significant than older data, motivating the need for sliding window solutions. Various capabilities, such as DDoS detection and load balancing, require insights about multiple metrics including Bloom filters, per-flow counting, count distinct and entropy estimation.

In this work, we present a unified construction that solves all the above problems in the sliding window model. Our single solution offers a better space to accuracy tradeoff than the state-of-the-art for each of these individual problems! We show this both analytically and by running multiple real Internet backbone and datacenter packet traces.

7.1 Background

Network measurements are at the core of many applications, such as load balancing, quality of service and anomaly/intrusion detection [95, 93, 50, 47]. Measurement algorithms are required to cope with the throughput demands of modern links, forcing them to rely on fast SRAM memory. Such memory is limited in size [55], which motivates approximate solutions that conserve space.

Network algorithms often find recent data useful. For example, anomaly detection solutions attempt to detect manifesting anomalies and a load balancer needs to balance the current load rather than the historical one. The sliding window model captures this recency notion and is therefore an active research field [14, 18, 19, 105, 106].

The desired measurement types differ from one application to the other. For example, a load balancer may be interested in the heavy hitter flows [95], which are responsible for a large portion of the traffic. Additionally, anomaly detection systems often monitor the number of distinct elements [50] and entropy [107] or use Bloom filters [108]. Yet, existing algorithms usually provide just a single utility at a time, e.g., approximate set membership (Bloom filters) [109], per-flow counting [63], count distinct [110, 111, 112]
Table 7.1: Summary of SWAMP’s accuracy guarantees.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Estimator</th>
<th>Guarantee</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(W, \epsilon)$-Approximate Set Membership</td>
<td>IsMember()</td>
<td>$\Pr(\text{true}</td>
<td>x \in S^W) = 1$</td>
</tr>
<tr>
<td>$(W, \epsilon)$-Approximate Set Multiplicity</td>
<td>FREQUENCY ($\hat{f}_L$)</td>
<td>$\Pr(f_L \leq \hat{f}_L) = 1$</td>
<td>Theorem 7.5.2</td>
</tr>
<tr>
<td>$(W, \epsilon, \delta)$-Approximate Count Distinct</td>
<td>DISTINCT LB ($Z$)</td>
<td>$\Pr(D \geq Z) = 1$</td>
<td>Theorem 7.5.9</td>
</tr>
<tr>
<td>$(W, \epsilon, \delta)$-Entropy Estimation</td>
<td>ENTROPY ($\hat{H}$)</td>
<td>$\Pr(H \geq \hat{H}) = 1$</td>
<td>Theorem 7.5.17</td>
</tr>
</tbody>
</table>

Table 7.1: Summary of SWAMP’s accuracy guarantees.

and entropy [107]. Therefore, as networks complexity grows, multiple measurement types may be required. However, employing multiple stand-alone solutions incurs the additive combined cost of each of them, which is inefficient in both memory and computation.

In this work, we suggest Sliding Window Approximate Measurement Protocol (SWAMP), an algorithm that bundles together four commonly used measurement types. Specifically, it approximates set membership, per flow counting, distinct elements and entropy in the sliding window model. As illustrated in Figure 7.1, SWAMP stores flows fingerprints in a cyclic buffer while their frequencies are maintained in a compact fingerprint hash table named TinyTable [63]. On each packet arrival, its corresponding fingerprint replaces the oldest one in the buffer. We then update the table, decrementing the departing fingerprint’s frequency and incrementing that of the arriving one. An additional counter $Z$ maintains the number of distinct fingerprints in the window and is updated every time a fingerprint’s frequency is reduced to 0 or increased to 1. Intuitively, the number of distinct fingerprints provides a good estimation of the number of distinct elements. Additionally, the scalar $\hat{H}$ (not illustrated) maintains the fingerprints distribution entropy and approximates the real entropy.

### 7.1.1 Contribution

We present SWAMP, a sliding window algorithm for approximate set membership (Bloom filters), per-flow counting, distinct elements and entropy measurements. We prove that SWAMP operates in constant time and provides accuracy guarantees for each of the supported problems. Despite its versatility, SWAMP improves the state of the art for each.

For approximate set membership, SWAMP is memory succinct when the false positive rate is constant and requires up to 40% less space than [18]. SWAMP is also succinct for per-flow counting and is more accurate than [14] on real packet traces. When compared with $1 + \epsilon$ count distinct approximation algorithms [113, 114], SWAMP asymptotically improves the query time from $O(\epsilon^{-2})$ to a constant. It is also up to $x1000$ times more

---

1 A fingerprint is a short random string obtained by hashing an ID.
Figure 7.1: An overview of SWAMP: Fingerprints are stored in a cyclic fingerprint buffer (CFB), and their frequencies are maintained by TinyTable. Upon item $x_n$’s arrival, we update CFB and the table by removing the oldest item’s ($x_n-W$) fingerprint (in black) and adding that of $x_n$ (in red). We also maintain an estimate for the number of distinct fingerprints ($Z$). Since the black fingerprints count is now zero, we decrement $Z$.

accurate on real packet traces. For entropy, SWAMP asymptotically improves the runtime to a constant and provides accurate estimations in practice.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Space</th>
<th>Time</th>
<th>Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWAMP</td>
<td>$(1 + o(1)) \cdot W \log_2 W$</td>
<td>$O(1)$</td>
<td>✓</td>
</tr>
<tr>
<td>SWBF [18]</td>
<td>$(2 + o(1)) \cdot W \log_2 W$</td>
<td>$O(1)$</td>
<td>x</td>
</tr>
<tr>
<td>TBF [106]</td>
<td>$O(W \log_2 W \log_2 \epsilon^{-1})$</td>
<td>$O(\log_2 \epsilon^{-1})$</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 7.2: Comparison of sliding window set membership algorithms for $\epsilon = W^{-o(1)}$.

the notations we use can be found in Table 7.3

7.1.2 Chapter organization

Related work on the problems covered by this work is found in Section 7.2. Section 7.3 provides formal definitions and introduces SWAMP. Section 7.4 describes an empirical evaluation of SWAMP and previously suggested algorithms. Section 7.5 includes a formal analysis of SWAMP which is briefly summarized in Table 7.1. Finally, we conclude with a short discussion in Section 7.6.
7.2 Related work

7.2.1 Set Membership and Counting

A Bloom filter [109] is an efficient data structure that encodes an approximate set. Given an item, a Bloom filter can be queried if that item is a part of the set. An answer of ‘no’ is always correct, while an answer of ‘yes’ may be false with a certain probability. This case is called False Positive.

Plain Bloom filters do not support removals or counting and thus many algorithms fill this gap. For example, some alternatives support removals [115, 63, 116, 117, 118, 119] and others support multiplicity queries [32, 63]. Additionally, some works use aging [105] and others compute the approximate set with regard to a sliding windows [106, 18].

SWBF [18] uses a Cuckoo hash table to build a sliding Bloom filter, which is more space efficient than previously suggested Timing Bloom filters (TBF) [106].

The Cuckoo table is allocated with 2W entries such that each entry stores a fingerprint and a timestamp. Cuckoo tables require that W entries remain empty to avoid circles and this is done implicitly by treating cells containing outdated items as ‘empty’. Finally, a cleanup process is used to remove outdated items and allow timestamps to be wrapped around. A comparison of SWAMP, TBF and SWBF appears in Table 7.2.

7.2.2 Count Distinct

The number of distinct elements provides a useful indicator for anomaly detection algorithms. Accurate count distinct is impractical due to the massive scale of the data [120] and thus most approaches resort to approximate solutions [121, 122, 110].

Approximate algorithms typically use a hash function \( H : \mathbb{D} \to \{0, 1\}^\infty \) that maps ids to infinite bit strings. In practice, finite bit strings are used and 32 bit integers suffice to reach estimations of over \( 10^9 \) [120]. These algorithms look for certain observables in the hashes. For example, some algorithms [121, 123] treat the minimal observed hash value as a real number in \([0, 1]\) and exploit the fact that \( \mathbb{E}(\min(H(M))) = \frac{1}{D+1} \), where \( D \) is the real number of distinct items in the multi-set \( M \). Alternatively, one can seek patterns of the form \( 0^{3-1} \) [110, 120] and exploit the fact that such a pattern is encountered on average once per every \( 2^3 \) unique elements.

Monitoring observables reduces the required amount of space as we only need to maintain a single memory word. In practice, the variance of such methods is large and hence multiple observables are maintained.

In principle, one could repeat the process and perform \( m \) independent experiments but this has significant computational overheads. Instead, stochastic averaging [112] is used to mimic the effects of multiple experiments with a single hash calculation. At any case, using \( m \) repetitions reduces the standard deviation by a factor of \( \frac{1}{\sqrt{m}} \).


112
Technion - Computer Science Department - Ph.D. Thesis PHD-2018-08 - 2018
The state of the art count distinct algorithm is HyperLogLog (HLL) [120], which is used in multiple Google projects [124]. HLL requires \( m \) bytes and its standard deviation is \( \sigma \approx 0.04 \sqrt{m} \). SWHLL extends HLL to sliding windows [113, 114] and was used to detect attacks such as port scans [125]. SWAMP’s space requirement is proportional to \( W \) and thus, it is only comparable in space to HLL when \( \varepsilon^{-2} = O(W) \). However, when multiple functionalities are required the residual space overhead of SWAMP is only \( \log(W) \) bits, which is considerably less than any standalone alternative.

7.2.3 Entropy Detection

Entropy is commonly used as a signal for anomaly detection [107]. Intuitively, it can be viewed as a summary of the entire traffic histogram. The benefit of entropy based approaches is that they require no exact understanding of the attack’s mechanism. Instead, such a solution assumes that a sharp change in the entropy is caused by anomalies.

An \( \epsilon, \delta \) approximation of the entropy of a stream can be calculated in \( O(\varepsilon^{-2} \log \delta^{-1}) \) space [126], an algorithm that was also extended to sliding window using priority sampling [127]. That sliding window algorithm is improved by [128] whose algorithm requires \( O(\varepsilon^{-2} \log \delta^{-1} \log(N)) \) memory.

7.2.4 Preliminaries – Compact Set Multiplicity

Our work requires a compact set multiplicity structure that supports both set membership and multiplicity queries. TinyTable [63] and CQF [129] fit the description while other structures [115, 117] can be extended for multiplicity queries at the expense of additional space. We choose TinyTable [63] as its code is publicly available in open source.

TinyTable encodes \( W \) fingerprints of size \( L \) using \( (1 + \alpha) W (L - \log_2(W) + 3) + o(W) \) bits, where \( \alpha \) is a small constant that affects update speed; when \( \alpha \) grows, TinyTable becomes faster but also consumes more space.

7.3 SWAMP Algorithm

7.3.1 Model

We consider a stream \( (S) \) of IDs where at each step an ID is added to \( S \). The last \( W \) elements in \( S \) are denoted \( S^W \). Given an ID \( y \), the notation \( f^W_y \) represents the frequency of \( y \) in \( S^W \). Similarly, \( \hat{f}^W_y \) is an approximation of \( f^W_y \). For ease of reference, notations are summarized in Table 7.3.

7.3.2 Problems definitions

We start by formally defining the approximate set membership problem.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W$</td>
<td>Sliding window size</td>
</tr>
<tr>
<td>$S$</td>
<td>Stream.</td>
</tr>
<tr>
<td>$S^W$</td>
<td>Last $W$ elements in the stream.</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>Accuracy parameter for sets membership.</td>
</tr>
<tr>
<td>$\varepsilon_D$</td>
<td>Accuracy parameter for count distinct.</td>
</tr>
<tr>
<td>$\varepsilon_H$</td>
<td>Accuracy parameter for entropy estimation.</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Confidence for count distinct.</td>
</tr>
<tr>
<td>$L$</td>
<td>Fingerprint length in bits, $L \equiv \log_2 (W\varepsilon^{-1})$.</td>
</tr>
<tr>
<td>$f_y^W$</td>
<td>Frequency of ID $y$ in $S^W$.</td>
</tr>
<tr>
<td>$\hat{f}_y^W$</td>
<td>An approximation of $f_y^W$.</td>
</tr>
<tr>
<td>$D$</td>
<td>Number of distinct elements in $S^W$.</td>
</tr>
<tr>
<td>$Z$</td>
<td>Number of distinct fingerprints.</td>
</tr>
<tr>
<td>$\hat{D}$</td>
<td>Estimation provided by DISTINCTMLE function.</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>TinyTable’s parameter ($\alpha = 1.2$).</td>
</tr>
<tr>
<td>$h$</td>
<td>A pairwise independent hash function.</td>
</tr>
<tr>
<td>$F$</td>
<td>A set of fingerprints stored in CFB.</td>
</tr>
</tbody>
</table>

Table 7.3: List of symbols

**Definition** We say that an algorithm solves the $(W, \varepsilon)$-APPROXIMATE SET MEMBERSHIP problem if given an ID $y$, it returns true if $y \in S^W$ and if $y \notin S^W$, it returns false with probability of at least $1 - \varepsilon$.

The above problem is solved by SWBF [19] and by *Timing Bloom filter (TBF)* [106]. In practice, SWAMP solves the stronger $(W, \varepsilon)$-APPROXIMATE SET MULTIPLICITY problem, as defined below:

**Definition** We say that an algorithm solves the $(W, \varepsilon)$-APPROXIMATE SET MULTIPLICITY problem if given an ID $y$, it returns an estimation $\hat{f}_y^W$ s.t. $\hat{f}_y^W \geq f_y^W$ and with probability of at least $1 - \varepsilon$: $f_y^W = \hat{f}_y^W$.

Intuitively, the $(W, \varepsilon)$-APPROXIMATE SET MULTIPLICITY problem guarantees that we always get an over approximation of the frequency and that with probability of at least $1 - \varepsilon$ we get the exact window frequency. A simple observation shows that any algorithm that solves the $(W, \varepsilon)$-APPROXIMATE SET MULTIPLICITY problem also solves the $(W, \varepsilon)$-APPROXIMATE SET MEMBERSHIP problem. Specifically, if $y \in S^W$, then $\hat{f}_y^W \geq f_y^W$ implies that $f_y^W \geq 1$ and we can return true. On the other hand, if $y \notin S^W$, then $f_y^W = 0$ and with probability of at least $1 - \varepsilon$, we get: $f_y^W = 0 = \hat{f}_y^W$. Thus, the isMember estimator simply returns true if $f_y^W > 0$ and false otherwise. We later show that this estimator solves the $(W, \varepsilon)$-APPROXIMATE SET MEMBERSHIP problem.

The goal of the $(W, \varepsilon, \delta)$-APPROXIMATE COUNT DISTINCT problem is to maintain an estimation of the number of distinct elements in $S^W$. We denote their number by $D$. 

Technion - Computer Science Department - Ph.D. Thesis PHD-2018-08 - 2018
Algorithm 10 SWAMP

Require: TinyTable $TT$, Fingerprint Array $CFB$, integer $curr$, integer $Z$

initialization
$CFB \leftarrow \emptyset$
$curr \leftarrow 0$
$Z \leftarrow 0$

$TT \leftarrow TinyTable$
$\bar{H} \leftarrow 0$

1: function $Update(ID x)$
2: $prevFreq \leftarrow TT.frequency(CFB[curr])$
3: $TT.remove(CFB[curr])$
4: $CFB[curr] \leftarrow h(x)$
5: $TT.add(CFB[curr])$
6: $xFreq \leftarrow TT.frequency(CFB[curr])$
7: $UpdateCD(prevFreq, xFreq)$
8: $UpdateEntropy(prevFreq, xFreq)$
9: $curr \leftarrow (curr + 1) \mod W$
end function

11: procedure $UpdateCD(prevFreq, xFreq)$
12: if $prevFreq = 1$ then
13: $Z \leftarrow Z - 1$
end if
15: if $xFreq = 1$ then
16: $Z \leftarrow Z + 1$
end if
end procedure

19: procedure $UpdateEntropy(prevFreq, xFreq)$
20: $PP \leftarrow \frac{prevFreq - 1}{W}$
21: $CP \leftarrow \frac{prevFreq - 1}{W}$
22: $\bar{H} \leftarrow \bar{H} + PP \log(PP) - CP \log CP$
23: $xPP \leftarrow \frac{prevFreq - 1}{W}$
24: $xCP \leftarrow \frac{prevFreq - 1}{W}$
25: $\bar{H} \leftarrow \bar{H} + xPP \log(xPP) - xCP \log(xCP)$
end procedure

27: function $IsMember(ID x)$
28: if $TT.frequency(h(x)) > 0$ then
29: return true
end if
31: return false
end function

33: function $Frequency(ID x)$
34: return $TT.frequency(h(x))$
end function

36: function $DistinctLB()$
37: return $Z$
end function

39: function $DistinctMLE()$
40: $\hat{D} \leftarrow \frac{\log(1 - \frac{Z}{W})}{\log(1 - \frac{Z}{W})}$
41: return $\hat{D}$
end function

Definition We say that an algorithm solves the $(W, \epsilon, \delta)$-Approximate Count Distinct problem if it returns an estimation $\hat{D}$ such that $D \geq \hat{D}$ and with probability $1 - \delta$: $\hat{D} \geq (1 - \epsilon) D$. 

115
Intuitively, an algorithm that solves the $(W, \epsilon, \delta)$-APPROXIMATE COUNT DISTINCT problem is able to conservatively estimate the number of distinct elements in the window and with probability of $1 - \delta$, this estimate is close to the real number of distinct elements.

The entropy of a window is defined as:

$$H \triangleq - \sum_{i=1}^{D} \frac{f_i}{W} \log \left( \frac{f_i}{W} \right),$$

where $D$ is the number of distinct elements in the window, $W$ is the total number of packets, and $f_i$ is the frequency of flow $i$. We define the window entropy estimation problem as:

**Definition** An algorithm solves the $(W, \epsilon, \delta)$-ENTROPY ESTIMATION problem, if it provides an estimator $\hat{H}$ so that, $H \geq \hat{H}$ and $\Pr \left( H - \hat{H} \geq \epsilon \right) \leq \delta$.

### 7.3.3 SWAMP algorithm

We now present *Sliding Window Approximate Measurement Protocol (SWAMP)*. SWAMP uses a single hash function ($h$), which given an ID ($y$), generates $L \triangleq \lceil \log_2(W \epsilon^{-1}) \rceil$ random bits $h(y)$ that are called its *fingerprint*. We note that $h$ only needs to be pairwise-independent and can thus be efficiently implemented using only $O(\log W)$ space. Fingerprints are then stored in a cyclic fingerprint buffer of length $W$ that is denoted $CFB$. The variable $curr$ always points to the oldest entry in the buffer. Fingerprints are also stored in TinyTable [63] that provides compact encoding and multiplicity information.

The update operation replaces the oldest fingerprint in the window with that of the newly arriving item. To do so, it updates both the cyclic fingerprint buffer ($CFB$) and TinyTable. In CFB, the fingerprint at location $curr$ is replaced with the newly arriving fingerprint. In TinyTable, we remove one occurrence of the oldest fingerprint and add the newly arriving fingerprint. The update method also updates the variable $Z$, which measures the number of distinct fingerprints in the window. $Z$ is incremented every time that a new unique fingerprint is added to TinyTable, i.e., $FREQUENCY(y)$ changes.
(a) Window size is $2^{16}$ and varying $\varepsilon$.  
(b) $\varepsilon = 2^{-10}$ and varying window sizes.

Figure 7.3: Memory consumption of sliding Bloom filters as a function of $W$ and $\varepsilon$.

from 0 to 1, where the method $FREQUENCY(y)$ receives a fingerprint and returns its frequency as provided by TinyTable. Similarly, denote $x$ the item whose fingerprint is removed; if $FREQUENCY(x)$ changes to 0, we decrement $Z$.

SWAMP has two methods to estimate the number of distinct flows. DISTINCTLB simply returns $Z$, which yields a conservative estimator of $D$ while DISTINCTMLE is an approximation of its Maximum Likelihood Estimator. Clearly, DISTINCTMLE is more accurate than DISTINCTLB, but its estimation error is two sided. A pseudo code is provided in Algorithm 7.3.1 and an illustration is given by Figure 7.1.

7.4 Empirical Evaluation

7.4.1 Overview

We evaluate SWAMP’s various functionalities, each against its known solutions. We start with the $(W, \varepsilon)$-APPROXIMATE SET MEMBERSHIP problem where we compare SWAMP to SWBF \cite{18} and Timing Bloom filter (TBF) \cite{106}, which only solve $(W, \varepsilon)$-APPROXIMATE SET MEMBERSHIP.

For counting, we compare SWAMP to Window Compact Space Saving (WCSS) \cite{14} that solves heavy hitters identification on a sliding window. We evaluate their empirical accuracy when both algorithms are given the same amount of space.

For the distinct elements problem, we compare SWAMP against Sliding Hyper Log Log \cite{113, 114}, denoted SWHLL, who proposed running HyperLogLog and LogLog on a sliding window. Small range correction is due for the tested configurations, thus HyperLogLog and LogLog devolve into the same algorithm. Additionally, as we know
that small range correction is active, we only allocate single bit counters. This option (SWLC) slightly improves the space/accuracy ratio.

In all measurements, we use a window size of $W = 2^{16}$ unless specified otherwise. In addition, the underlying TinyTable uses $\alpha = 0.2$ as recommended by its authors [63].

Our evaluation includes six Internet packet traces consisting of backbone routers and data center traffic. The backbone traces contain a mix of 1 billion UDP, TCP and ICMP packets collected from two major routers in Chicago [87] and San Jose [70] during the years 2013-2016. The dataset Chicago 16 refers to data collected from the Chicago router in 2016, San Jose 14 to data collected from the San Jose router in 2014, etc. The datacenter packet traces are taken from two university datacenters consisting of up to 1,000 servers [89]. These traces are denoted DC1 and DC2.

### 7.4.2 Evaluation of analytical guarantees

Figure 7.2 evaluates the accuracy of our analysis from Section 7.5 on random inputs. As can be observed, the analysis of sliding Bloom filter (Figure 7.2(a)) and count distinct (Figure 7.2(b)) is accurate. For entropy (Figure 7.2(c)) the accuracy is better than...
Figure 7.5: Accuracy of SWAMP’s count distinct functionality compared to alternatives.

anticipated indicating that our analysis here is just an upper bound, but the trend line is nearly identical.

7.4.3 Set membership on sliding windows

We now compare SWAMP to TBF [106] and SWBF [118]. Our evaluation focuses on two aspects, fixing $\epsilon$ and changing the window size (Figure 7.3(b)) as well as fixing the window size and changing $\epsilon$ (Figure 7.3(a)).

As can be observed, SWAMP is considerably more space efficient than the alternatives in both cases for a wide range of window sizes and for a wide range of error probabilities. In the tested range, it is 25-40% smaller than the best alternative.

7.4.4 Per-flow counting on sliding windows

Next, we evaluate SWAMP for its per-flow counting functionality. We compare SWAMP to WCSS [114] that solves heavy hitters on a sliding window. Our evaluation uses the On Arrival model, which was used to evaluate WCSS. In that model, we perform a query for each incoming packet. Then, we calculate the Root Mean Square Error. We repeated each experiment 25 times with different seeds and computed 95% confidence intervals for SWAMP. Note that WCSS is a deterministic algorithm and as such was run only once.
Figure 7.6: Accuracy of SWAMP’s entropy functionality compared to the theoretical guarantee.

The results appear in Figure 7.4. Note that SWAMP’s space requirement is proportional to $W$, and that therefore it is only feasible for some of the range. Yet, when it is feasible, SWAMP’s error is lower on average than that of WCSS. Additionally, in many of the configurations we are able to show statistical significance to this improvement. Note that SWAMP becomes accurate with high probability using about 300KB of memory while WCSS requires about 8.3MB to provide the same accuracy. That is, an improvement of x27.

7.4.5 Count distinct on sliding windows

Next, we evaluate the count distinct functionality in terms of accuracy vs. space on the different datasets. Each data point is the average of 25 runs and evaluates two functionalities: SWAMP-LB corresponding to the function DISTINCTLB in Algorithm 7.3.1 and SWAMP-MLE corresponding to the function DISTINCTMLE in Algorithm 7.3.1. Our results are in Figure 7.5 show that SWAMP-MLE is up to x1000 more accurate than its alternatives. Moreover, SWAMP-LB provides a one sided error guarantee and is still superior to the alternatives for part of the range.
7.4.6 Entropy estimation on sliding window

Figure 7.6 shows results for Entropy estimation. As shown, SWAMP provides a very accurate entropy estimation in its entire operational range. Moreover, our analysis in Section 7.5.3 is conservative and SWAMP is much more accurate in practice.

7.5 Analysis of SWAMP

This section is dedicated for the analysis of SWAMP. Our analysis is partitioned into three subsections. Section 7.5.1 shows that SWAMP operates in constant time and solves the \((W, \epsilon)\)-APPROXIMATE SET MEMBERSHIP and the \((W, \epsilon)\)-APPROXIMATE SET MULTIPLICITY problems. Next, Section 7.5.2 shows that SWAMP solves the \((W, \epsilon, \delta)\)-APPROXIMATE COUNT DISTINCT problem using DISTINCTLB and that DISTINCTMLE approximates its Maximum Likelihood Estimator. Finally, Section 7.5.3 shows that SWAMP solves \((W, \epsilon, \delta)\)-ENTROPY ESTIMATION.

7.5.1 Analysis of per flow counting

**Theorem 7.5.1.** SWAMP’s runtime is \(O(1)\) with high probability.

*Proof.* Update - Updating the cyclic buffer requires two TinyTable operations - add and remove - both performed in constant time (with high probability) for any constant \(\alpha\). The manipulations to \(\hat{H}\) and \(Z\) are also done in constant time.

ISMEMBER - Is satisfied by TinyTable in constant time.

FREQUENCY - Is satisfied by TinyTable in constant time.

DISTINCTLB - Is satisfied by returning an integer.

DISTINCTMLE - Is satisfied with a simple calculation.

ENTROPY - Is satisfied by returning a floating point.

Next, we prove that SWAMP solves the \((W, \epsilon)\)-APPROXIMATE SET MULTIPLICITY problem with regard to the function FREQUENCY (in Algorithm 7.3.1).

**Theorem 7.5.2.** SWAMP solves the \((W, \epsilon)\)-APPROXIMATE SET MULTIPLICITY problem. That is, given an ID \(y\), FREQUENCY \((y)\) provides an estimation \(\hat{f}_y^W\) such that: \(\hat{f}_y^W \geq f_y^W\) and \(\Pr\left[f_y^W = \hat{f}_y^W\right] \geq 1 - \epsilon\).

*Proof.* SWAMP’s CFB variable stores \(W\) different fingerprints, each of length \(L = \log_2 (W\epsilon^{-1})\). For a fingerprint of size \(L\), the probability that two fingerprints collide is
There are \( W - 1 \) fingerprints that \( h(y) \) may collide with, and \( \hat{f}_y^W > f_y^W \) only if it collides with one of the fingerprints in CFB. Next, we use the Union Bound to get:

\[
\Pr \left[ f_y^W \neq \hat{f}_y^W \right] \leq W \cdot 2^{-\log_2 (W\epsilon^{-1})} = \varepsilon.
\]

Thus, given an item \( y \), with probability \( 1 - \varepsilon \), its fingerprint \( h(y) \) is unique and TinyTable accurately measures \( f_y^W \). Any collision of \( h(y) \) with other fingerprints only increases \( \hat{f}_y^W \), thus \( \hat{f}_y^W \geq f_y^W \) in all cases.

Theorem 7.5.2 shows that SWAMP solves the \((W, \epsilon)\)-APPROXIMATE SET MULTIPLICITY, which includes per flow counting and sliding Bloom filter functionalities.

Our next step is to analyze the space consumption of SWAMP. This enables us to show that SWAMP is memory optimal to the \((W, \epsilon)\)-APPROXIMATE SET MULTIPLICITY, or more precisely, that it is succinct (according to Definition 7.5.1).

**Definition** Given an information theoretic lower bound \( B \), an algorithm is called *succinct* if it uses \( B(1 + o(1)) \) bits.

**Theorem 7.5.3.** Given a window size \( W \) and an accuracy parameter \( \epsilon \), the number of bits required by SWAMP is:

\[
W \left( \lceil \log_2 (W\epsilon^{-1}) \rceil + (1 + \alpha) (\log_2 \epsilon^{-1} + 3) \right) + o(W).
\]

**Proof.** Our cyclic buffer (CFB) stores \( W \) fingerprints, each of size \( \lceil \log_2 (W\epsilon^{-1}) \rceil \), for an overall space of \( W\lceil \log_2 (W\epsilon^{-1}) \rceil \) bits. Additionally, TinyTable requires: \( (1 + \alpha) W (\log_2 (W\epsilon^{-1}) - \log_2 (W) + 3) + o(W) \)

\[
= (1 + \alpha) W (\log_2 \epsilon^{-1} + 3) + o(W)
\]

Each one of the variables \textit{curr}, \textit{Z} and \textit{H} require \( \log(W) = o(W) \) and thus SWAMP’s memory consumption is

\[
W \left( \lceil \log_2 (W\epsilon^{-1}) \rceil + (1 + \alpha) (\log_2 \epsilon^{-1} + 3) \right) + o(W).
\]

The work of [19] implies a lower bound for the \((W, \epsilon)\)-APPROXIMATE SET MEMBERSHIP problem of

\[
B \triangleq W \left( \log_2 \epsilon^{-1} + \max \left( \log_2 \log_2 \epsilon^{-1}, \log_2 (W) \right) \right)
\]

bits. This lower bound also applies to our case, as SWAMP solves the \((W, \epsilon)\)-APPROXIMATE SET MULTIPLICITY and the \((W, \epsilon)\)-APPROXIMATE SET MEMBERSHIP by returning whether or not \( \hat{f}_y^W > 0 \).

**Corollary 7.5.4.** SWAMP solves the \((W, \epsilon)\)-APPROXIMATE SET MEMBERSHIP problem.

We now show that SWAMP is succinct for \( \epsilon = W^{-o(1)} \).
Theorem 7.5.5. Let \( \epsilon = W^{-o(1)} \). Then SWAMP is succinct for the \((W, \epsilon)\)-Approximate Set Membership and \((W, \epsilon)\)-Approximate Set Multiplicity problems.

Proof. Theorem 7.5.3 shows that the required space is:

\[
S \leq W \log_2 W + W(2 + \alpha) \log_2 \epsilon^{-1} + 4W + 3W \alpha + o(W) = \\
W \log_2 W \cdot \left(1 + \frac{1}{\log_2 W} \left[(2 + \alpha) \log_2 \epsilon^{-1} + 4 + 3\alpha + o(1)\right]\right) = \\
W \log_2 W \cdot \left(1 + (2 + \alpha) \log_2 \epsilon^{-1} \right) + o(1) = \\
W \log_2 W \cdot \left(1 + (2 + \alpha) \log_2 \epsilon^{-1} \right).
\]

We show that this implies succinctness according to Definition 7.5.1. Under our assumption that \( \epsilon = W^{-o(1)} \), we get that:

\[
\log_2 \epsilon^{-1} = \log_2(W^{-o(1)}) = o(\log_2 W),
\]

hence SWAMP uses \( W \log_2 W(1 + o(1)) = B(1 + o(1)) \) bits and is succinct.

7.5.2 Analysis of count distinct functionality

We now move to analyze SWAMP’s count distinct functionality. Recall that SWAMP has two estimators; the one sided DISTINCTLB and the more accurate DISTINCTMLE. Also, recall that \( Z \) monitors the number of distinct fingerprints and that \( D \) is the real number of distinct flows (on the window).

Analysis of DISTINCTLB

We now present an analysis for DISTINCTLB, showing that it solves the \((W, \epsilon, \delta)\)-Approximate Count Distinct problem.

Theorem 7.5.6. \( Z \leq D \) and \( E(Z) \geq D \cdot (1 - \frac{\epsilon}{2}) \).

Proof. Clearly, \( Z \leq D \) always, since any two identical IDs map to the same fingerprint. Next, for any \( 0 \leq i \leq 2^L - 1 \), denote by \( Z_i \) the indicator random variable, indicating whether there is some item in the window whose fingerprint is \( i \). Then \( Z = \sum_{i=0}^{2^L} Z_i \), hence \( E(Z) = \sum_{i=0}^{2^L} E(Z_i) \). However, for any \( i \),

\[
E(Z_i) = \Pr(Z_i = 1) = 1 - (1 - 2^{-L})^D.
\]

The probability that a fingerprint is exactly \( i \) is \( 2^{-L} \), thus:

\[
E(Z) = 2^L \cdot \left(1 - (1 - 2^{-L})^D\right).
\]

Since \( 0 < 2^{-L} \leq 1 \), we have:

\[
(1 - 2^{-L})^D < 1 - D \cdot 2^{-L} \leq D \frac{D - 1}{2} \cdot 2^{-L} \] which, in turn, implies \( E(Z) > D - \frac{D}{2} \cdot 2^{-L} \). Finally, we note that \( 2^{-L} = \frac{\epsilon}{W} \) and that \( D \leq W \). Hence, \( E(Z) > D - \frac{D(D - 1)\epsilon}{2W} > D \cdot (1 - \frac{\epsilon}{2}) \).

Our next step is a probabilistic bound on the error of \( Z \) (\( X = D - Z \)). This is required for Theorem 7.5.9, which is the main result and shows that \( Z \) provides an \( \epsilon, \delta \) approximation for the distinct elements problem.
We model the problem as a balls and bins problem where \( D \) balls are placed into \( 2L = \frac{W}{\varepsilon} \) bins. The variable \( X \) is the number of bins with at least 2 balls.

For any \( 0 \leq i \leq 2^L - 1 \), denote by \( X_i \) the random variable denoting the number of balls in the \( i \)-th bin. Note that the variables \( X_i \) are dependent of each other and are difficult to reason about directly. Luckily, \( X \) is monotonically increasing and we can use a Poisson approximation. To do so, we denote by \( Y_i \sim \text{Poisson} \left( \frac{D}{2^L} \right) \) the corresponding independent Poisson variables and by \( Y \) the Poisson approximation of \( X \), that is \( Y \) is the sum of \( Y_i \)'s with value 2 or more.

Our goal is to apply Lemma 7.5.7 which links the Poisson approximation to the exact case. In our case, it allows us to derive insight about \( X \) by analyzing \( Y \).

**Lemma 7.5.7 (Corollary 5.11, page 103 of [101])**. Let \( E \) be an event whose probability is either monotonically increasing or decreasing with the number of balls. If \( E \) has probability \( p \) in the Poisson case then \( E \) has probability at most \( 2p \) in the exact case. Here \( E \) is an event depending on \( X_0, \ldots, X_{2^L-1} \) (in the exact case), and the probability of the event in the Poisson case is obtained by computing \( E \) using \( Y_0, \ldots, Y_{2^L-1} \).

That is, we need to bound the probability \( \Pr(X_i \geq 2) \) and to do so we define a random variable \( Y \) and bound: \( \Pr(Y_i \geq 2) \) which can be at most \( 2 \cdot \Pr(X_i \geq 2) \).

We may now denote by \( Y_i = Y_i \geq 2 \) the indicator variables, indicating whether \( Y_i \) has value at least 2. By definition:

\[
Y = \sum_{i=0}^{2^L-1} Y_i.
\]

It follows that:

\[
\mathbb{E}(Y) = \sum_{i=0}^{2^L-1} \mathbb{E}(Y_i) = 2^L \cdot \Pr(Y_i \geq 2).
\]

\( Y_i \sim \text{Poisson} \left( \frac{D}{2^L} \right) \), hence:

\[
\Pr(Y_i \geq 2) = 1 - \Pr(Y_i = 0) - \Pr(Y_i = 1) = 1 - e^{-\frac{D}{2^L}} \cdot \left( 1 + \frac{D}{2^L+1} \right),
\]

and thus:

\[
\mathbb{E}(Y) = 2^L \cdot \left( 1 - e^{-\frac{D}{2^L}} \cdot \left( 1 + \frac{D}{2^L} \right) \right).
\]

Since \( e^{-x} \cdot (1+x) > 1 - \frac{x^2}{2} \) for \( 0 < x \leq 1 \), we get

\[
\mathbb{E}(Y) < 2^L \cdot \left( 1 - \left( 1 - \frac{D^2}{2^{2L+1}} \right) \right) = \frac{D^2}{2^{L+1}}.
\]

Recall that \( 2^L = W\varepsilon^{-1} \) and \( D \leq W \) and thus \( \mathbb{E}(Y) < \frac{D\varepsilon}{2} \).

Note also that the \( \{Y_i\} \) are independent, since the \( \{Y_i\} \) are independent, and as they are indicator (Bernoulli) variables, in particular they are Poisson trials. Therefore, we may use a Chernoff bound on \( Y \) as it is the sum of independent Poisson trials. We use the following Chernoff bound to continue:
Lemma 7.5.8 (Theorem 4.4, page 64 of [101]). Let $X_1, \ldots, X_n$ be independent Poisson trials such that: $\Pr(X_i) = p_i$, and let $X = \sum_{i=1}^{n} X_i$, then for $R \geq 6\mathbb{E}(X)$, $\Pr(X \geq R) \leq 2^{-R}$.

Lemma 7.5.8 allows us to approach Theorem 7.5.9, which is the main result for DISTINCTLB and shows that it provides an $\epsilon, \delta$ approximation of $D$, which we can then use in Corollary 7.5.10 to show that it solves the count distinct problem.

Theorem 7.5.9. Let $\delta \leq \frac{1}{128}$. Then:

$$\Pr\left(D - Z \geq \frac{1}{2} \epsilon D \cdot \log\left(\frac{2}{\delta}\right)\right) \leq \delta.$$  

Proof. For $\delta \leq \frac{1}{128}$, $\log\left(\frac{2}{\delta}\right) \geq 6$ and thus, as $\log_2\left(\frac{2}{\delta}\right) \cdot \frac{1}{2} \epsilon D \geq 6\mathbb{E}(Y)$ we can use Lemma 7.5.8 to get that:

$$\Pr\left(Y \geq \frac{1}{2} \epsilon D \cdot \log_2\left(\frac{2}{\delta}\right)\right) \leq 2^{-\epsilon D \log_2\left(\frac{2}{\delta}\right)} \leq 2^{-\log_2\left(\frac{2}{\delta}\right)} \leq \delta.$$

As $X$ monotonically increases with $D$, we use Lemma 7.5.7 to conclude that

$$\Pr\left(X \geq \frac{1}{2} \epsilon D \cdot \log_2\left(\frac{2}{\delta}\right)\right) \leq \delta.$$  

The following readily follows, as claimed.

Corollary 7.5.10. DISTINCTLB solves the (W, $\epsilon, \delta$)-APPROXIMATE COUNT DISTINCT problem, for $\epsilon_D = \frac{1}{2} \epsilon \cdot \log_2\left(\frac{2}{\delta}\right)$, and any $\delta \leq \frac{1}{128}$. That is, for such $\delta$ we get: $\Pr(Z \geq (1 - \epsilon_D) \cdot D) \geq 1 - \delta$.

Analysis of DISTINCTMLE

We now provide an analysis of the DISTINCTMLE estimation method, which is more accurate but offers two sided error. Our goal is to derive confidence intervals around DISTINCTMLE and for this we require a better analysis of $Z$, which is provided by Lemma 7.5.11.

Lemma 7.5.11.

$$D - \left(\frac{D}{2}\right) \cdot 2^{-L} \leq \mathbb{E}(Z) \leq D - \left(\frac{D}{2}\right) \cdot 2^{-L} + \left(\frac{D}{3}\right) \cdot 2^{-2L}. \quad (7.2)$$

Proof. This follows simply from 7.1 by expanding the binomial.  

We also need to analyze the second moment of $Z$. Lemma 7.5.12 does just that.

Lemma 7.5.12.

$$\mathbb{E}(Z^2) = 2^{2L} - \left(2^{2L+1} - 2^L\right)\left(1 - \frac{1}{2^L}\right)^D + \left(2^L - 2^L\right)\left(1 - \frac{1}{2^{L-1}}\right)^D.$$  

125
Proof. As in Theorem 7.5.6 we write $Z$ as a sum of indicator variables $Z = \sum_{i=0}^{2^L-1} Z_i$. Then

$$Z^2 = \sum_{i=0}^{2^L-1} Z_i^2 + \sum_{i \neq j} Z_i Z_j.$$  

By linearity of the expectation, it implies

$$\mathbb{E}(Z^2) = \sum_{i=0}^{2^L-1} \mathbb{E}(Z_i^2) + \sum_{i \neq j} \mathbb{E}(Z_i Z_j). \quad (7.3)$$

We note that for any $i \neq j$, $(1 - Z_i)(1 - Z_j)$ is also an indicator variable, attaining the value 1 with probability $(1 - \frac{2}{2^L})^D$. Therefore, for any $i \neq j$, we have by linearity of the expectation, that

$$\left(1 - \frac{2}{2^L}\right)^D = \mathbb{E}(1 - Z_i)(1 - Z_j) = 1 - \mathbb{E}(Z_i) - \mathbb{E}(Z_j) + \mathbb{E}(Z_i Z_j).$$

Since $\mathbb{E}Z_i = 1 - (1 - \frac{1}{2^L})^D$, it follows that

$$\mathbb{E}(Z_i Z_j) = \left(1 - \frac{2}{2^L}\right)^D + 2 \cdot \left(1 - \left(1 - \frac{1}{2^L}\right)^D\right) - 1 = 1 + \left(1 - \frac{2}{2^L}\right)^D - 2 \left(1 - \frac{1}{2^L}\right)^D.$$

Plugging it back into Equation (7.3) and using $Z_i^2 = Z_i$, we obtain

$$\mathbb{E}(Z^2) = 2^L \cdot \left(1 - \left(1 - \frac{1}{2^L}\right)^D\right) + 2^L (2^L - 1) \left(1 + \left(1 - \frac{2}{2^L}\right)^D - 2 \left(1 - \frac{1}{2^L}\right)^D\right).$$

Finally, expanding and rearranging we obtain the claim of this lemma. \qed 

As our interest lies in the confidence intervals, we shall only need an approximation, described in the following simple corollary

**Corollary 7.5.13.** $D^2 - \frac{4D^3}{32^L} < \mathbb{E}(Z^2) < D^2 + \frac{D^3}{32^L}.$

**Proof.** First, note that binomial expansion yields

$$1 - \frac{D}{2^L} + \left(\frac{D}{2^L}\right)^3 - \frac{D}{2^{3L}} \leq \left(1 - \frac{1}{2^L}\right)^D \leq 1 - \frac{D}{2^L} + \left(\frac{D}{2^L}\right)^3.$$

Plugging it back into Lemma 7.5.12 and expanding we get on the one hand

$$\mathbb{E}(Z^2) > 2^L - (2^{2L+1} - 2^L) \left(1 - \frac{D}{2^L} + \left(\frac{D}{2^L}\right)^3\right) + (2^{2L} - 2^L) \left(1 - \frac{D}{2^{L-1}} + \left(\frac{D}{2^{2L-2}} - \frac{D}{2^{3L-3}}\right)\right) = D + 2 \cdot \left(\frac{D}{2}\right) - 2^{-L} \cdot \left(\frac{3D(D - 1)}{2} + \frac{8D(D - 1)(D - 2)}{6}\right) = D^2 - \frac{D(D - 1)(8D - 7)}{6 \cdot 2^L},$$
which yields the lower bound. On the other hand, we have

\[
\mathbb{E}(Z^2) < 2^{2L} - (2^{2L+1} - 2^L) \left( 1 - \frac{D}{2^L} + \frac{(D)^2}{2^{2L}} - \frac{(D)^3}{2^{3L}} \right) + \\
+ (2^{2L} - 2^L) \left( 1 - \frac{D}{2^{L-1}} + \frac{(D)^2}{2^{2L-2}} \right) =
\]

\[
D + 2 \cdot \left( \frac{D}{2} \right) + 2^{-L} \cdot \left( -\frac{3D(D-1)}{2} + \frac{D(D-1)(D-2)}{3} \right) =
\]

\[
D^2 + \frac{D(D-1)(2D-13)}{6 \cdot 2^L},
\]

which yields the upper bound. Thus, we have established the corollary.

Using Corollary 7.5.13 and Lemma 7.5.11 we can finally get an estimate for \(\mathbb{E}(\hat{D})\), as described in the following theorem. This shows that \(\hat{D}\) is unbiased up to an \(O(\varepsilon^2)\) additive factor.

**Theorem 7.5.14.**

\[
-D \cdot \varepsilon^2 \cdot \left( \frac{2}{3} + \frac{\varepsilon}{2W^2} \right) < \mathbb{E}(\hat{D}) < D \cdot \varepsilon^2 \cdot \left( \frac{2}{3(1-\varepsilon)^3} \right).
\]

**Proof.** We first note that for \(x \in [0, 1)\) one has

\[
x + \frac{x^2}{2} \leq -\ln(1-x) \leq x + \frac{x^2}{2} + \frac{x^3}{3(1-x)^3}.
\]

Therefore, we have

\[
\frac{Z}{2^L} + \frac{Z^2}{2^{2L+1}} \leq -\ln \left( 1 - \frac{Z}{2^L} \right) \leq \frac{Z}{2^L} + \frac{Z^2}{2^{2L+1}} + \frac{Z^3}{3 \cdot (2^L-Z)^3}.
\]

Since always \(Z \leq D\), and \(\frac{D}{2^L} \leq \frac{W}{2^L} = \varepsilon\), we can take expectation and obtain, by linearity and monotonicity of the expectation, that

\[
\frac{\mathbb{E}(Z)}{2^L} + \frac{\mathbb{E}(Z^2)}{2^{2L+1}} \leq \mathbb{E} \left( -\ln \left( 1 - \frac{Z}{2^L} \right) \right) \leq \frac{\mathbb{E}(Z)}{2^L} + \frac{\mathbb{E}(Z^2)}{2^{2L+1}} + \frac{D^3}{3 \cdot (2^L-D)^3}.
\]

We can now substitute Lemma 7.5.11 and Corollary 7.5.13 to obtain

\[
-\mathbb{E} \left( \ln \left( 1 - \frac{Z}{2^L} \right) \right) \leq \frac{D}{2^L} + \frac{D}{2^{2L+1}} + \frac{D^3}{6 \cdot 2^L} + \\
+ \frac{(D)^2}{2^{3L}} + \frac{D^3}{3 \cdot (2^L-D)^3} \leq \frac{D}{2^L} + \frac{D}{2^{2L+1}} + \\
+ D \cdot \left( \frac{2W^2}{2^{3L}} + \frac{W^2}{3(1-\varepsilon)^3 \cdot 2^L} \right).
\]
Recalling (7.5) we see also that $-\ln \left( 1 - \frac{1}{2^x} \right) \geq \frac{1}{2^x} + \frac{1}{2^{2x+1}}$. Combining both inequalities, we get

$$E(\hat{D}) \leq D + D \cdot \left( \frac{2W^2}{2^2L + 2^{L-1}} + \frac{W^2}{3(1 - \varepsilon)^3 \cdot (2^{2L} + 2^{L-1})} \right) \leq D + D \cdot \left( \frac{2W^2}{2^2L} + \frac{W^2}{3(1 - \varepsilon)^3 \cdot 2^{2L}} \right).$$

Since $2^L = W \varepsilon^{-1}$, this gives us the upper bound. On the other hand, substituting Lemma 7.5.11 and Corollary 7.5.13 in the left inequality given in (7.6) we have also

$$-E\left( \ln \left( 1 - \frac{Z}{2^x} \right) \right) \geq \frac{D}{2^L} + \frac{D}{2^{2L+1}} - \frac{4D^3}{6 \cdot 2^{3L}}.$$

From (7.5) we see also that $-\ln \left( 1 - \frac{1}{2^x} \right) \leq \frac{1}{2^x} + \frac{1}{2^{2x+1}} + \frac{1}{3(2^x-1)^2}$. Combining both inequalities, we get

$$E(\hat{D}) \geq D - \frac{4D^3}{6 \cdot (2^L + 2^{L-1} + 3)} - \frac{8D}{3 \cdot 2^3L} \geq D - D \cdot \left( \frac{2D^2}{3 \cdot 2^2L} + \frac{8}{3 \cdot 2^3L} \right).$$

Since $D \leq W$ and $2^L = W \varepsilon^{-1}$, this yields the lower bound, as claimed. \(\square\)

Next, we bound the error probability. This is done in the following theorem.

**Theorem 7.5.15.** Let $\delta \leq \frac{1}{128}$ and denote $\varepsilon_D = \frac{1}{2} \varepsilon \cdot \log_2 \left( \frac{2}{3} \right)$. Then

$$\Pr\left( \hat{D} \leq D \cdot (1 - \varepsilon_D) \right) \leq \delta.$$

**Proof.** We first note that

$$\Pr\left( \frac{\ln \left( 1 - \frac{Z}{2^x} \right)}{\ln \left( 1 - \frac{1}{2^x} \right)} \leq a \right) = \Pr\left( Z \leq 2^L \left( 1 - \left( 1 - \frac{1}{2^L} \right)^a \right) \right).$$

Now, using Corollary 7.5.10 and the fact that

$$2^L \left( 1 - \left( 1 - \frac{1}{2^L} \right)^a \right) \leq a,$$

the result immediately follows. \(\square\)

Theorem 7.5.15 shows the soundness of the DISTINCTMLE estimator by proving that it is at least as accurate as the DISTINCTLB estimator.
7.5.3 Analysis of entropy estimation functionality

We now turn to analyze the entropy estimation. Recall that SWAMP uses the estimator $\hat{H}$. If we denote by $F$ the set of distinct fingerprints in the last $W$ elements, it is given by

$$\hat{H} = -\sum_{h \in F} \frac{n_h}{W} \log\left(\frac{n_h}{W}\right),$$

where $n_h$ is the number of occurrences of fingerprint $h$ in the window of last $W$ elements.

We begin by showing that $E(\hat{H})$ approximates $H$, where $H = -\sum_{y \in D} \frac{f^W_y}{W} \log\left(\frac{f^W_y}{W}\right)$ is the entropy in the window.

**Theorem 7.5.16.** $\hat{H} \leq H$ and $E(\hat{H})$ is at least $H - \varepsilon$.

**Proof.** For any $y \in D$, let $h(y)$ be its fingerprint. Recall that $\hat{f}^W_y$ is the frequency $n_h(y)$ of $h(y)$, hence we have:

$$\hat{f}^W_y = \sum_{y': h(y') = h(y)} f^W_{y'}.$$

For ease of notation, we denote $p_y = \frac{f^W_y}{W}$, and $p_h = \frac{n_h}{W}$. Thus

$$p_h = \sum_{y \in D : h(y) = h} p_y.$$

It follows that:

$$\hat{H} = -\sum_{h \in F} p_h \log(p_h) = -\sum_{h \in F} \left(\sum_{y \in D : h(y) = h} p_y\right) \log\left(\sum_{y \in D : h(y) = h} p_y\right)$$

$$= -\sum_{y \in D} p_y \cdot \log\left(\sum_{y' \in D : h(y') = h(y)} p_{y'}\right).$$

Now, since $p_{y'} \geq 0$ for all $y'$, it follows that for any $y$, $\sum_{y' \in D : h(y') = h(y)} p_{y'} \geq p_y$.

Hence, by the monotonicity of the logarithm, $\hat{H} \leq -\sum_{y \in D} p_y \log(p_y) = H$ proving the first part of our claim. Conversely, denote for any $y \in D$ and any $h \in F$, by $I_{y,h}$ the Bernoulli random variable, attaining the value 1 if $h(y) = h$. Then we see that:

$$\hat{H} = -\sum_{y \in D} p_y \cdot \log\left(\sum_{y' \in D} p_{y'} \cdot I_{y',h(y)}\right).$$
We see, by using Jensen’s inequality, the concavity of the logarithm and the linearity of expectation, that for any $y$:

$$\mathbb{E} \log \left( \sum_{y' \in D} p_{y'} \cdot I_{y', h(y)} \right) \leq \log \left( \sum_{y' \in D} p_{y'} \cdot \mathbb{E}(I_{y', h(y)}) \right)$$

Since for any $y' \neq y$, we have $\mathbb{E}(I_{y', h(y)}) = 2^{-L}$, we see that

$$\mathbb{E} \log \left( \sum_{y' \in D} p_{y'} \cdot I_{y', h(y)} \right) \leq \log \left( 2^{-L} \cdot \sum_{y' \neq y \in D} p_{y'} + p_y \right) = \log(p_y + 2^{-L}(1 - p_y))$$

Summing this over $y$, and using the linearity of expectation once more, we obtain:

$$\mathbb{E}(\hat{H}) \geq -\sum_{y \in D} p_y \log(p_y + 2^{-L}(1 - p_y)).$$

Subtracting $H$ yields:

$$\mathbb{E}(\hat{H}) - H \geq -\sum_{y \in D} p_y \left( \log(p_y + 2^{-L}(1 - p_y)) - \log(p_y) \right)$$

$$\geq -\sum_{y \in D} p_y \left( \log(1 + 2^{-L} \left( \frac{1}{p_y} - 1 \right)) \right).$$

Note that $2^{-L} = \frac{\epsilon}{W}$ and $\frac{1}{W} \leq p_y < 1$. Hence, $0 < 2^{-L} \left( \frac{1}{p_y} - 1 \right) < \epsilon$. Since for any $0 < x$, $\log(1 + x) < x$, we get:

$$\mathbb{E}(\hat{H}) - H \geq -\sum_{y \in D} 2^{-L} \cdot (1 - p_y) = -\frac{\epsilon}{W} \cdot (D - 1).$$

As $D \leq W$, we get that $\mathbb{E}(\hat{H}) > H - \epsilon$, as claimed.

We proceed with a confidence interval derivation:

**Theorem 7.5.17.** Let $\epsilon_H = \epsilon \delta^{-1}$ then $\hat{H}$ solves $(W, \epsilon_H, \delta)$-Entropy Estimation. That is: $\Pr \left( H - \hat{H} \geq \epsilon_H \right) \leq \delta$.

**Proof.** Denote by $X \triangleq H - \hat{H}$ the random variable of the estimation error. According to Theorem 7.5.16, $X$ is non-negative and $E(X) \leq \epsilon$. Therefore, according to Markov’s theorem we have

$$\Pr \left( H \leq \hat{H} + \epsilon \delta^{-1} \right) = 1 - \Pr \left( X \geq \epsilon \delta^{-1} \right) \geq 1 - \frac{E(X)}{\epsilon \delta^{-1}} \geq 1 - \delta. \square$$

Setting $\epsilon_H = \epsilon \delta^{-1}$ and rearranging the expression completes the proof.
7.6 Discussion

In modern networks, operators are likely to require multiple measurement types. To that end, this work suggests SWAMP, a unified algorithm that monitors four common measurement metrics in constant time and compact space. Specifically, SWAMP approximates the following metrics on a sliding window: Bloom filters, per-flow counting, count distinct and entropy estimation. For all problems, we proved formal accuracy guarantees and demonstrated them on real Internet traces.

Despite being a general algorithm, SWAMP advances the state of the art for all these problems. For sliding Bloom filters, we showed that SWAMP is memory succinct for constant false positive rates and that it reduces the required space by 25%-40% compared to previous approaches [19]. In per-flow counting, our algorithm outperforms WCSS [14] – a state of the art window algorithm. When compared with $1 + \epsilon$ approximation count distinct algorithms [113, 114], SWAMP asymptotically improves the query time from $O(\epsilon^{-2})$ to a constant. It is also up to x1000 times more accurate on real packet traces. For the entropy estimation on a sliding window [130], SWAMP reduces the update time to a constant.

While SWAMP benefits from the compactness of TinyTable [63], most of its space reductions inherently come from using fingerprints rather than sketches. For example, all existing count distinct and entropy algorithms require $\Omega(\epsilon^{-2})$ space for computing a $1 + \epsilon$ approximation. SWAMP can compute the exact answers using $O(W \log W)$ bits. Thus, for a small $\epsilon$ value, we get an asymptotic reduction.

Finally, while our analysis of SWAMP is mathematically involved, the actual code is short and simple to implement. This facilitates its adoption in network devices and SDN. In particular, OpenBox [131] demonstrated that sharing common measurement results across multiple network functionalities is feasible and efficient. Our work fits into this trend.
Chapter 8

Conclusions

In this work, we studied techniques for efficient network measurements. We introduced algorithms and lower bounds for a variety of measurement tasks over both streams and sliding windows. Our algorithms aim to close the gap between the theoretical methods proposed in the literature and the actual performance when running on a networking device. The solutions are rigorously analyzed and evaluated using both simulations and virtual switch implementations.

8.1 Summing over Sliding Windows

The first project we considered was the average/total throughput of a link in a given time window. While existing solutions that compute a $(1+\epsilon)$ multiplicative approximations were known, they seem too costly both in terms of memory usage and update time. We have given, as a project, the task of implementing both our solutions and that of [2] to five undergraduate students. The results were overwhelming; while the space requirement for the same error was 3-4 times higher (in [2]), the main improvement came in the running time. The per-packet processing time of our algorithm was 100x-1000x times faster due to its simplicity.

We thoroughly analyze our algorithms and provide tight lower bounds. In a recent follow-up work, we prove that our algorithms are succinct – optimal up to a $(1+o(1))$-factor in their memory consumption.

8.2 Randomized Admission Policies

The second work of this thesis addresses the frequent flows problem. Our solution provides frequency estimations and finds the top-$k$ flows better than previous approaches. Intuitively, we utilize randomness to filter out tail items. This significantly increases the accuracy of the algorithm for heavy-tailed data, which is frequent in networking workloads.
Indeed, our solutions that are tailored for such data exhibits superior precision and provide the same error rates with while reducing the space by a factor of x2-x32.

8.3 Weighted Heavy Hitters

An important generalization of the frequent flows problem is its weighted variant that allows one to find the flows that consume the most bandwidth. In our third project, we explored methods for identifying flows with a large traffic volume over both streams and sliding windows. We further experimented with applying our technique to the Hierarchical Heavy Hitters problem in which we identify heavy networks and not just flows. In all cases, we reduced the update time to a constant and showed a speedup of up to x7. For sliding windows, our approach also reduces the space required by previous solutions by a staggering x1000 factor.

8.4 Hierarchical Heavy Hitters

The problem of Hierarchical Heavy Hitters, in which we wish to find networks that send an excessive amount of traffic, is an essential tool for distributed denial of service (DDoS) detection tools. Alas, existing solutions seem to slow for practical implementation, especially on virtual switches. Given the recent rise in software switching usage, designing a monitoring solution that can run on the switch without crashing the throughput was long overdue.

Our solution is based on a simple change to an existing algorithm, but one proved as extremely useful. When processing a packet, current solutions compute all its prefixes and count them individually. Instead, we only compute at most one prefix. This can be seen as a sampling technique, but one that allows for constant worst-case update time. As we show both analytically and empirically, this sampling procedure requires convergence time of a few seconds but improves the update time by a factor of up to x62. Most importantly, we evaluate our solution on a real virtual switch and shows that it can handle the current line rates. On the other hand, incorporating previous techniques in the switch reduced its throughput by a 2.5x factor.

8.5 All in One Sliding Bloom Filter

The last work of this thesis studies the problem of identifying the active flows; specifically, those that have at least one packet in the recent time window. This problem is known as Sliding Bloom Filter. Compared to previous approaches [18], we reduce the memory requirement by 25-40% and present a succinct algorithm. Further, we discovered that our data structure supports additional functionalities that existing sliding Bloom filters
do not. Namely, our solution allows estimating per-flow frequency estimation (over sliding windows), counting distinct elements, and finding the window’s entropy. All four measurement types are known to be useful for security purposes, and here we present a single algorithm that addresses all. Interestingly, the space overhead for computing the extra functionalities is negligible compared with the memory needed for a sliding Bloom filter. Finally, we compared our solution against state of the art algorithms that were each designed for solving just one measurement type. We show that in all functions, the unified construction we present outperforms the existing dedicated sketches given the same memory.
Bibliography


[86] P. Hick, “CAIDA Anonymized Internet Trace, equinix-chicago 2015-12-17 13:00-13:05 UTC, Direction A.”

[87] ——, “CAIDA Anonymized Internet Trace, equinix-chicago 2016-02-18 13:00-13:05 UTC, Direction A.”

[88] ——, “CAIDA Anonymized 2013 Internet Trace, equinix-sanjose 2013-12-19 13:00-13:05 UTC, Direction B.”


In conclusion, the work presented in this thesis shows that the developed algorithms and systems are effective in detecting and mitigating attacks and anomalies, and that the use of the same data structure allows for the extension of the algorithms to solve other measurement problems. The results are promising and demonstrate the potential of the methods developed for efficient and accurate measurement of network traffic.
We developed a new algorithm to improve the scalability of the federated learning system, which reduces the update time exponentially and also provides significant practical benefits. In detail, we present a processing rate up to 2.4x in utilizing only 1-1% of the competitors' memory. Finally, we show how the method allows identifying deep networks (an efficient way of known), which is also the topic of the next chapter.

Chapter 6 considers the problem of network identification. This problem is a further generalization of identifying frequent connections, where the goal is to identify networks that generate a large number of packets. The motivation for this problem is to prevent service denial attacks. In a denial-of-service attack, the attacker uses a large number of devices (which can be a computer, a laptop, a smartphone, a web camera, etc.) to overload the service to levels where it cannot serve real clients. During the attack, each device generates a small number of requests and generates a communication pattern that appears legitimate, but the total number of requests does not allow the service to function properly. Since the communication pattern of each attacker component appears normal, there is no choice but to focus on analyzing the communication pattern of groups of components. In this work, we propose a method that tries to identify the networks from which the attack comes by identifying multiple packets from them.

For this problem, the existing solutions are considered slow and not feasible in practice. We developed the first method that can process packets at a constant and empirically shown to improve by $62\%-70\%$. In this way, we developed a method that allows OVS to maintain the required throughput while reducing the network performance of methods by $60\%$.

This project is the last in this chapter, as described in Chapter 7, focuses on measuring on virtual windows. In this work, we consider a variety of measurement types for the purpose of identifying attacks. The first measurement type is identifying active connections. In the problem of identifying active connections, the system needs to answer questions in the form of "was a packet from window X seen?". In recent times, the challenge in this type of measurement is to try to evaluate the answer in real-time without explicitly maintaining a list of all the windows. In the solutions to this problem, we are willing to accept a type I error (False Positive) but not type II error (False Negative). The idea is that we are not willing to miss an attack, but we are checking a small number of false positives, which allows efficient algorithms. For this problem, we developed an algorithm that reduces by $25\%-40\%$ the memory usage of existing methods. In addition, the main advantage of this method is that 25%-40% of the total system cost is spent on the client's machine.
כפי שתיארנו מעלה. במקום זאת, אנו מציעים אלגוריתם אשר מאפשר לקרב את הסכום ועל ידי שיטות דחיסה מוריד את צריכת הזכרון להיות לוגריתמית ב -W. כדוגמה, אם נניח שברצוננו להעריך את ניצול קו של 100 ג'יגה ביט לשניה (100Gbps) עבור היממה האחרונה (86400 שניות),igth את המסה האופטימלית מaura 16 קילו ביט של זכרון בערך 3MB. לעומת זאת, אם נאפשר שגיאה של 16 קילו ביט (ואכן היא מילונית מהסכום האפשרי ב exemplo זה), השיטה המוצגת מספקת חישוב בערך 120 קילו ביט של זכרוןолько 120 קילו ביט של זכרוןgetLocation. בצריכת הזכרון נוספת, אנו מראים חסמים תחתונים על אלגוריתמים לבעיה זו ומוכיחים כי השיטה שלנו קרובה לופטימיות.

העבודה הבאה, שמוצגת בפרק 4, מדברת עלthèse גורמי השכיחים (heavy hitters) בזרם חבילות (packets). במקרה זה, הנתב רואה רצף של חבילות כאשר כל אחת מזוהה על ידי מזהה חיבור (flow identifier). חיבור שכיח הוא חיבור שמופיע כמות גדולה של פעמים בשטף המידע, והמטרה היא ל Unidos את החיבורים השכיחים ביותר. באופן פורמלי, אנו מנסים לזהות את k החיבורים השכיחים תוך מזעור כמות הזכרון. השיטה המוצעת מנצלת רנדומיות על מנת לשפר את דיוק המדידה באופן חדשני. אינטואיטיבית, מטרת האלגוריתם היא לזהות את החיבורים השכיחים. אז, אם מגיעה חבילה בעלת מזהה שאריה במבנה הנתונים, הרי שהמזהה שלה אינו שכיח. בהתאם, אנו מציעים להטיל מטבע עבור חיבורים שלא מונעים. כלומר, חבילות אשר המזהה שלהם מנוטר בזמן ההגעה ימדדו במדויק ואילו חבילות другой יגרמו לאלגוריתם להתחיל לנתר את החיבור ולהתחיל לנתר את החיבור השכיחות שהולכות ודועכת ככל שהאלגוריתם מתקדם. בעת הערכה (evaluation) האלגוריתם והשוואה לשוואות המובילות בתחום, הראינו כי גישת ההכנסה ההסתברותית שפיתחנו מאפשרת לקבל דיוק דומה למתחרים תוך שיפור של 4x-32x בכמות הזכרון.

הפריקט השישי, אוشر מופעי בפרק 5, מדבר עלchè חיבורים בשכיחים (jumbo frames) במטרה להגדיל את יכולת השיגור ב걥ג. ב HDCP, שולח מתודת את hè גורמי השכיחים הרב וحسب 영, אלא מಸקיפיווד את השיגור. למין, שולח מתודת את hè גורמי השכיחים הרב וحسب את hè גורמי השכיחים. ספציפית, יונק hè גורמי השכיחים מسقط קומ של hè גורמי השכיחים. נוספים,ève שורח את hè גורמי השכיחים, אך גם יונק hè גורמי השכיחים לפי thresholds. הגדרה המסתכל על hè גורמי הכוחות. בנסוף, שב贿ו וירח את hè גורמי השכיחיםuml. וימושיות פתרון אשר מומן את hè גורמי השכיחים על חלון 2, כדי שהטיהטר בפרק 3, משותם.
מדידות רשת הן כלי בסיסי והכרחי למגוון רחב של ישומיים כגון איזון עומסים, זיהוי חדירות, והבטחת איכות השירות. ביסודן, על מדידות הרשת להתבצע על ידי רכיבי הניתוב על מנת لنתח את התעבורה בפועל. עם זאת, הקצבים הגבוהים בהם נדרשים הנתבים앞יון מהווים אתגר משמעותי צווכן המדידות. למשל, בניתוב חומרה, לנתב יש זכרון מהיר שמאפשר גישה בזמן אמת בשם SRAM, אך SRAM הוא יקר וקטן יחסית (נכון ל-2017, מדובר בעשרות מגה ביט). מכאן, והיות ומספר החיבורים (flows) עשוי להיות ענק, אין לנתב האפשרות לנטר כל החיבור בנפרד במדויק. הפתרון הטיפוסי לבעיה זו הוא בשימוש באלגוריתמים ידידים מצומצמים המאושפים למשרורי ליניות לכל תזמור בודד בحاد למצב

בעית. כך, הנתב (הוירטואלי) יentario זכרון (מסוג DRAM) אך על מנת לתמוך בקצב הקו המ۾יד המידתי חייבות להיות קלות-משקל מבית כמות החישוב. גם ב случае תוכנה, בקצב הקהמה המידיום הייחודי לקוות-مشاكل מבית זכרון נבדליםコンardi (cache) המעבד, דר שחרור בזירה למבית ולתפסק (throughput) את האפשרות录ים הירטואליים של

הפרוייקט הא흐ואש במבנה מתקדם במדידת ניצול של קִוּ תקשורת. הארגוגה 우리는 את המודל מדידות רשת בגרסת טכנולוגיה סמארטפון (Big Data) שאור מקסימום של (1) ס…theלונה(SRAM)icides פוקס, 1-2) ס…theלונה של גיוס המודל. בכל הנוגע לקבוצת מהגרים, ניתן במדידת ניצול זה בשימוש בשיטות שלậm ביבית ואור מקסימום במדידת הוא האזורית של המודול לבקל בתפקידים בגרסת טכנולוגיה סמארטפון. האזורית של המודול לבקל בתפקידים בגרסת טכנולוגיה סמארטפון. האזורית של המודול לבקל בתפקידים בגרסת טכנולוגיה סמארטפון. האזורית של המודול לבקל בתפקידים בגרסת טכנולוגיה סמארטפון. האזורית של המודול לבקל בתפקידים בגרסת טכנולוגיה סמארטפון. האזורית של המודול לבקל בתפקידים בגרסת טכנולוגיה סמארטפון. האזורית של המודול לבקל לתוך שחלמה.

 applause
תודות

המחקר בוצע בהנחייתו של פרופ' רועי פרידמן, בפקולטה למגעים והמהנדס
בטכניון. אני מודה לטכניון ול-HPI על התמיכה הנדיבה במהלך השתלמותי.

אנני מודה ל-CRC ול-HPI על התמיכה והנידונה במושך השיתוף הלאומית.
אלגוריתמי מידע מסיבי למדידות רשת

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר דוקטור לפילוסופיה

רן בן בשט

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
תמוז התשע״ח חיפה יולי 2018

רותם למט גנוביץ – מחלקת טכנולוגיה ילישראלי

ת ViewGroup תبشرת חמחמ Danielle 2018
אלגוריתמי מידע מסיבי למדידות רשת

רן בן בשט

1